This is the Chainer documentation.
CHAPTER 1

Installation Guide

Recommended Environments

We recommend these Linux distributions.

- Ubuntu 14.04/16.04 LTS 64bit
- CentOS 7 64bit

The following versions of Python can be used: 2.7.6+, 3.4.3+, 3.5.1+, and 3.6.0+.

Note: We are testing Chainer automatically with Jenkins, where all the above recommended environments are tested. We cannot guarantee that Chainer works on other environments including Windows and macOS (especially with CUDA support), even if Chainer looks running correctly.

Dependencies

Before installing Chainer, we recommend to upgrade setuptools if you are using an old one:

```
$ pip install -U setuptools
```

The following Python packages are required to install Chainer. The latest version of each package will automatically be installed if missing.

- NumPy 1.9, 1.10, 1.11, 1.12
- Six 1.9+

The following packages are optional dependencies. Chainer can be installed without them, in which case the corresponding features are not available.

- CUDA/cuDNN support
Install Chainer

Install Chainer via pip

We recommend to install Chainer via pip:

```bash
$ pip install chainer
```

**Note:** Any optional dependencies (including CuPy) can be added after installing Chainer. Chainer automatically detects the available packages and enables/disables the optional features appropriately.

Install Chainer from source

The tarball of the source tree is available via `pip download chainer` or from the release notes page. You can use `setup.py` to install Chainer from the tarball:

```bash
$ tar zxf chainer-x.x.x.tar.gz
$ cd chainer-x.x.x
$ python setup.py install
```

You can also install the development version of Chainer from a cloned Git repository:

```bash
$ git clone https://github.com/chainer/chainer.git
$ cd chainer
$ python setup.py install
```

When an error occurs...

Use `-vvvv` option with `pip` command. That shows all logs of installation. It may help you:

```bash
$ pip install chainer -vvvv
```

Enable CUDA/cuDNN support

In order to enable CUDA support, you have to install CuPy manually. If you also want to use cuDNN, you have to install CuPy with cuDNN support. See CuPy’s installation guide to install CuPy. Once CuPy is correctly set up, Chainer will automatically enable CUDA support.
You can refer to the following flags to confirm if CUDA/cuDNN support is actually available.

\texttt{chainer.cuda.available} True iff Chainer successfully imports \texttt{cupy}.
\texttt{chainer.cuda.cudnn_enabled} True iff cuDNN support is available.

**Support image dataset**

Install Pillow manually to activate image dataset feature:

\begin{verbatim}
$ pip install pillow
\end{verbatim}

Note that this feature is optional.

**Support HDF5 serialization**

Install h5py manually to activate HDF5 serialization:

\begin{verbatim}
$ pip install h5py
\end{verbatim}

Before installing h5py, you need to install libhdf5. The way to install it depends on your environment:

\begin{verbatim}
# Ubuntu 14.04/16.04
$ apt-get install libhdf5-dev

# CentOS 7
$ yum -y install epel-release
$ yum install hdf5-devel
\end{verbatim}

Note that this feature is optional.

**Uninstall Chainer**

Use pip to uninstall Chainer:

\begin{verbatim}
$ pip uninstall chainer
\end{verbatim}

Note: When you upgrade Chainer, \texttt{pip} sometimes install the new version without removing the old one in \texttt{site-packages}. In this case, \texttt{pip uninstall} only removes the latest one. To ensure that Chainer is completely removed, run the above command repeatedly until \texttt{pip} returns an error.

**Upgrade Chainer**

Just use \texttt{pip} with \texttt{--upgrade} option:

\begin{verbatim}
$ pip install --upgrade chainer
\end{verbatim}
Reinstall Chainer

If you want to reinstall Chainer, please uninstall Chainer and then install it. We recommend to use `--no-cache-dir` option as pip sometimes uses cache:

```
$ pip uninstall chainer
$ pip install chainer --no-cache-dir
```

Run Chainer with Docker

We are providing the official Docker image. Use `nvidia-docker` command to run Chainer image with GPU. You can login to the environment with bash, and run the Python interpreter:

```
$ nvidia-docker run -it chainer/chainer /bin/bash
```

Or run the interpreter directly:

```
$ nvidia-docker run -it chainer/chainer /usr/bin/python
```

FAQ

The installer says “hdf5.h is not found”

You don’t have libhdf5. Please install it first. See Support HDF5 serialization.

Examples say “cuDNN is not enabled”

You failed to build CuPy with cuDNN. If you don’t need cuDNN, ignore this message. Otherwise, retry to install CuPy with cuDNN, `--v v v v` option helps you. There is no need of re-installing Chainer itself. See CuPy’s installation guide for more details.
Introduction to Chainer

This is the first section of the Chainer Tutorial. In this section, you will learn about the following things:

- Pros and cons of existing frameworks and why we are developing Chainer
- Simple example of forward and backward computation
- Usage of links and their gradient computation
- Construction of chains (a.k.a. “model” in most frameworks)
- Parameter optimization
- Serialization of links and optimizers

After reading this section, you will be able to:

- Compute gradients of some arithmetics
- Write a multi-layer perceptron with Chainer

Core Concept

As mentioned on the front page, Chainer is a flexible framework for neural networks. One major goal is flexibility, so it must enable us to write complex architectures simply and intuitively.

Most existing deep learning frameworks are based on the “Define-and-Run” scheme. That is, first a network is defined and fixed, and then the user periodically feeds it with mini-batches. Since the network is statically defined before any forward/backward computation, all the logic must be embedded into the network architecture as data. Consequently, defining a network architecture in such systems (e.g. Caffe) follows a declarative approach. Note that one can still produce such a static network definition using imperative languages (e.g. torch.nn, Theano-based frameworks, and TensorFlow).

In contrast, Chainer adopts a “Define-by-Run” scheme, i.e., the network is defined on-the-fly via the actual forward computation. More precisely, Chainer stores the history of computation instead of programming logic. This strategy
enables us to fully leverage the power of programming logic in Python. For example, Chainer does not need any magic to introduce conditionals and loops into the network definitions. The Define-by-Run scheme is the core concept of Chainer. We will show in this tutorial how to define networks dynamically.

This strategy also makes it easy to write multi-GPU parallelization, since logic comes closer to network manipulation. We will review such amenities in later sections of this tutorial.

Note: In the example code of this tutorial, we assume for simplicity that the following symbols are already imported:

```python
import numpy as np
import chainer
from chainer import cuda, Function, gradient_check, report, training, utils, Variable
from chainer import datasets, iterators, optimizers, serializers
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
from chainer.training import extensions
```

These imports appear widely in Chainer code and examples. For simplicity, we omit these imports in this tutorial.

### Forward/Backward Computation

As described above, Chainer uses the “Define-by-Run” scheme, so forward computation itself defines the network. In order to start forward computation, we have to set the input array to a `Variable` object. Here we start with a simple `ndarray` with only one element:

```python
>>> x_data = np.array([5], dtype=np.float32)
>>> x = Variable(x_data)
```

A `Variable` object has basic arithmetic operators. In order to compute \( y = x^2 - 2x + 1 \), just write:

```python
>>> y = x**2 - 2 * x + 1
```

The resulting `y` is also a `Variable` object, whose value can be extracted by accessing the `data` attribute:

```python
>>> y.data
array([16.], dtype=float32)
```

What `y` holds is not only the result value. It also holds the history of computation (or computational graph), which enables us to compute its differentiation. This is done by calling its `backward()` method:

```python
>>> y.backward()
```

This runs error backpropagation (a.k.a. backprop or reverse-mode automatic differentiation). Then, the gradient is computed and stored in the `grad` attribute of the input variable `x`:

```python
>>> x.grad
array([8.], dtype=float32)
```

Also we can compute gradients of intermediate variables. Note that Chainer, by default, releases the gradient arrays of intermediate variables for memory efficiency. In order to preserve gradient information, pass the `retain_grad` argument to the backward method:
>>> z = 2*x
>>> y = x**2 - z + 1
>>> y.backward(retain_grad=True)
>>> z.grad
array([-1.], dtype=float32)

All these computations are easily generalized to a multi-element array input. Note that if we want to start backward computation from a variable holding a multi-element array, we must set the initial error manually. This is done simply by setting the grad attribute of the output variable:

```python
>>> x = Variable(np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float32))
>>> y = x**2 - 2*x + 1
>>> y.grad = np.ones((2, 3), dtype=np.float32)
>>> y.backward()
>>> x.grad
array([[ 0.,  2.,  4.],
       [ 6.,  8., 10.]], dtype=float32)
```

Note: Many functions taking Variable object(s) are defined in the functions module. You can combine them to realize complicated functions with automatic backward computation.

## Links

In order to write neural networks, we have to combine functions with parameters and optimize the parameters. You can use links to do this. A link is an object that holds parameters (i.e. optimization targets).

The most fundamental ones are links that behave like regular functions while replacing some arguments by their parameters. We will introduce higher level links, but here think of links as simply functions with parameters.

One of the most frequently used links is the Linear link (a.k.a. fully-connected layer or affine transformation). It represents a mathematical function \( f(x) = Wx + b \), where the matrix \( W \) and the vector \( b \) are parameters. This link corresponds to its pure counterpart `linear()`, which accepts \( x, W, b \) as arguments. A linear link from three-dimensional space to two-dimensional space is defined by the following line:

```python
>>> f = L.Linear(3, 2)
```

Note: Most functions and links only accept mini-batch input, where the first dimension of the input array is considered as the batch dimension. In the above Linear link case, input must have shape of \( (N, 3) \), where \( N \) is the mini-batch size.

The parameters of a link are stored as attributes. Each parameter is an instance of Variable. In the case of the Linear link, two parameters, \( W \) and \( b \), are stored. By default, the matrix \( W \) is initialized randomly, while the vector \( b \) is initialized with zeros.

```python
>>> f.W.data
array([[ 1.01847613, 0.23103087, 0.56507462],
       [ 1.29378033, 1.07823515, -0.56423163]], dtype=float32)
>>> f.b.data
array([ 0.,  0.], dtype=float32)
```

An instance of the Linear link acts like a usual function:
>>> x = Variable(np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float32))
>>> y = f(x)
>>> y.data
array([[ 3.1757617 ,  1.75755572],
       [ 8.61950684,  7.18090773]], dtype=float32)

Note: Sometimes it is cumbersome to compute the dimension of the input space. The linear link and some of (de)convolution links can omit the input dimension in their instantiation and infer it from the first mini-batch. For example, the following line creates a linear link whose output dimension is two:

f = L.Linear(2)

If we feed a mini-batch of shape \((N, M)\), the input dimension will be inferred as \(M\), which means \(f.W\) will be a \(2 \times M\) matrix. Note that its parameters are initialized in a lazy manner at the first mini-batch. Therefore, \(f\) does not have \(W\) attribute if no data is put to the link.

Gradients of parameters are computed by the \texttt{backward()} method. Note that gradients are \texttt{accumulated} by the method rather than overwritten. So first you must clear gradients to renew the computation. It can be done by calling the \texttt{cleargrads()} method.

```python
>>> f.cleargrads()
```

Note: \texttt{cleargrads()} is introduced in v1.15 to replace \texttt{zerograds()} for efficiency. \texttt{zerograds()} is left only for backward compatibility.

Now we can compute the gradients of parameters by simply calling the backward method.

```python
>>> y.grad = np.ones((2, 2), dtype=np.float32)
>>> y.backward()
>>> f.W.grad
array([[ 5.,  7.,  9.],
       [ 5.,  7.,  9.]], dtype=float32)
>>> f.b.grad
array([ 2.,  2.], dtype=float32)
```

Write a model as a chain

Most neural network architectures contain multiple links. For example, a multi-layer perceptron consists of multiple linear layers. We can write complex procedures with parameters by combining multiple links like this:

```python
>>> l1 = L.Linear(4, 3)
>>> l2 = L.Linear(3, 2)
>>> def my_forward(x):
...     h = l1(x)
...     return l2(h)
```

Here the \texttt{L} indicates the \texttt{links} module. A procedure with parameters defined in this way is hard to reuse. More Pythonic way is combining the links and procedures into a class:

```python
>>> class MyProc(object):
...     def __init__(self):
...         ...
```
In order to make it more reusable, we want to support parameter management, CPU/GPU migration, robust and flexible save/load features, etc. These features are all supported by the `Chain` class in Chainer. Then, what we have to do here is just define the above class as a subclass of Chain:

```python
>>> class MyChain(Chain):
...     def __init__(self):
...         super(MyChain, self).__init__()
...         with self.init_scope():
...             self.l1 = L.Linear(4, 3)
...             self.l2 = L.Linear(3, 2)
...     def __call__(self, x):
...         h = self.l1(x)
...         return self.l2(h)
```

It shows how a complex chain is constructed by simpler links. Links like `l1` and `l2` are called child links of `MyChain`. Note that `Chain` itself inherits `Link`. It means we can define more complex chains that hold `MyChain` objects as their child links.

**Note:** We often define a single forward method of a link by the `__call__` operator. Such links and chains are callable and behave like regular functions of Variables.

**Note:** In Chainer v1, we could also register the trainable layers (i.e., `Link`s) to the model by putting them to the `__init__()` of `Chain` or registering them via `add_link()`. But as these ways are deprecated in Chainer v2, users are recommended to use the way explained above.

Another way to define a chain is using the `ChainList` class, which behaves like a list of links:

```python
>>> class MyChain2(ChainList):
...     def __init__(self):
...         super(MyChain2, self).__init__(
...             L.Linear(4, 3),
...             L.Linear(3, 2),
...         )
...     def __call__(self, x):
...         h = self[0](x)
...         return self[1](h)
```

`ChainList` can conveniently use an arbitrary number of links, however if the number of links is fixed like in the above case, the `Chain` class is recommended as a base class.

**Optimizer**

In order to get good values for parameters, we have to optimize them by the `Optimizer` class. It runs a numerical optimization algorithm on a given link. Many algorithms are implemented in the `optimizers` module. Here we use
the simplest one, called Stochastic Gradient Descent (SGD):

```python
>>> model = MyChain()
>>> optimizer = optimizers.SGD()
>>> optimizer.setup(model)
```

The method `setup()` prepares for the optimization given a link.

Some parameter/gradient manipulations, e.g. weight decay and gradient clipping, can be done by setting hook functions to the optimizer. Hook functions are called after the gradient computation and right before the actual update of parameters. For example, we can set weight decay regularization by running the next line beforehand:

```python
>>> optimizer.add_hook(chainer.optimizer.WeightDecay(0.0005))
```

Of course, you can write your own hook functions. It should be a function or a callable object, taking the optimizer as the argument.

There are two ways to use the optimizer. One is using it via `Trainer`, which we will see in the following sections. The other way is using it directly. We here review the latter case. If you are interested in getting able to use the optimizer in a simple way, skip this section and go to the next one.

There are two further ways to use the optimizer directly. One is manually computing gradients and then calling the `update()` method with no arguments. Do not forget to clear the gradients beforehand!

```python
>>> x = np.random.uniform(-1, 1, (2, 4)).astype('f')
>>> model.cleargrads()
>>> # compute gradient here...
>>> loss = F.sum(model(chainer.Variable(x)))
>>> loss.backward()
>>> optimizer.update()
```

The other way is just passing a loss function to the `update()` method. In this case, `cleargrads()` is automatically called by the update method, so the user does not have to call it manually.

```python
>>> def lossfun(arg1, arg2):
"""... # calculate loss
...     loss = F.sum(model(arg1 - arg2))
...     return loss"
>>> arg1 = np.random.uniform(-1, 1, (2, 4)).astype('f')
>>> arg2 = np.random.uniform(-1, 1, (2, 4)).astype('f')
>>> optimizer.update(lossfun, chainer.Variable(arg1), chainer.Variable(arg2))
```

See `Optimizer.update()` for the full specification.

## Trainer

When we want to train neural networks, we have to run `training loops` that update the parameters many times. A typical training loop consists of the following procedures:

1. Iterations over training datasets
2. Preprocessing of extracted mini-batches
3. Forward/backward computations of the neural networks
4. Parameter updates
5. Evaluations of the current parameters on validation datasets
6. Logging and printing of the intermediate results
Chainer provides a simple yet powerful way to make it easy to write such training processes. The training loop abstraction mainly consists of two components:

- **Dataset abstraction.** It implements 1 and 2 in the above list. The core components are defined in the `dataset` module. There are also many implementations of datasets and iterators in `datasets` and `iterators` modules, respectively.

- **Trainer.** It implements 3, 4, 5, and 6 in the above list. The whole procedure is implemented by `Trainer`. The way to update parameters (3 and 4) is defined by `Updater`, which can be freely customized. 5 and 6 are implemented by instances of `Extension`, which appends an extra procedure to the training loop. Users can freely customize the training procedure by adding extensions. Users can also implement their own extensions.

We will see how to use Trainer in the example section below.

### Serializer

Before proceeding to the first example, we introduce Serializer, which is the last core feature described in this page. Serializer is a simple interface to serialize or deserialize an object. `Link`, `Optimizer`, and `Trainer` supports serialization.

Concrete serializers are defined in the `serializers` module. It supports NumPy NPZ and HDF5 formats.

For example, we can serialize a link object into NPZ file by the `serializers.save_npz()` function:

```
>>> serializers.save_npz('my.model', model)
```

It saves the parameters of `model` into the file `my.model` in NPZ format. The saved model can be read by the `serializers.load_npz()` function:

```
>>> serializers.load_npz('my.model', model)
```

**Note:** Note that only the parameters and the **persistent values** are serialized by this serialization code. Other attributes are not saved automatically. You can register arrays, scalars, or any serializable objects as persistent values by the `Link.add_persistent()` method. The registered values can be accessed by attributes of the name passed to the `add_persistent` method.

The state of an optimizer can also be saved by the same functions:

```
>>> serializers.save_npz('my.state', optimizer)
>>> serializers.load_npz('my.state', optimizer)
```

**Note:** Note that serialization of optimizer only saves its internal states including number of iterations, momentum vectors of MomentumSGD, etc. It does not save the parameters and persistent values of the target link. We have to explicitly save the target link with the optimizer to resume the optimization from saved states.

Support of the HDF5 format is enabled if the h5py package is installed. Serialization and deserialization with the HDF5 format are almost identical to those with the NPZ format; just replace `save_npz()` and `load_npz()` by `save_hdf5()` and `load_hdf5()` respectively.

### Example: Multi-layer Perceptron on MNIST

Now you can solve a multiclass classification task using a multi-layer perceptron (MLP). We use a hand-written digits dataset called MNIST, which is one of the long-standing de facto “hello world” examples used in machine learning.
This MNIST example is also found in the examples/mnist directory of the official repository. We show how to use Trainer to construct and run the training loop in this section.

We first have to prepare the MNIST dataset. The MNIST dataset consists of 70,000 greyscale images of size 28x28 (i.e. 784 pixels) and corresponding digit labels. The dataset is divided into 60,000 training images and 10,000 test images by default. We can obtain the vectorized version (i.e., a set of 784 dimensional vectors) by datasets.get_mnist().

```python
>>> train, test = datasets.get_mnist()
...
```

This code automatically downloads the MNIST dataset and saves the NumPy arrays to the $(HOME)/.chainer directory. The returned train and test can be seen as lists of image-label pairs (strictly speaking, they are instances of TupleDataset).

We also have to define how to iterate over these datasets. We want to shuffle the training dataset for every epoch, i.e. at the beginning of every sweep over the dataset. In this case, we can use iterators.SerialIterator.

```python
>>> train_iter = iterators.SerialIterator(train, batch_size=100, shuffle=True)
```

On the other hand, we do not have to shuffle the test dataset. In this case, we can pass shuffle=False argument to disable the shuffling. It makes the iteration faster when the underlying dataset supports fast slicing.

```python
>>> test_iter = iterators.SerialIterator(test, batch_size=100, repeat=False, shuffle=False)
```

We also pass repeat=False, which means we stop iteration when all examples are visited. This option is usually required for the test/validation datasets; without this option, the iteration enters an infinite loop.

Next, we define the architecture. We use a simple three-layer rectifier network with 100 units per layer as an example.

```python
>>> class MLP(Chain):
...     def __init__(self, n_units, n_out):
...         super(MLP, self).__init__()
...         with self.init_scope():
...             # the size of the inputs to each layer will be inferred
...             self.l1 = L.Linear(None, n_units)  # n_in -> n_units
...             self.l2 = L.Linear(None, n_units)  # n_units -> n_units
...             self.l3 = L.Linear(None, n_out)   # n_units -> n_out
...             ...  
...         def __call__(self, x):
...             h1 = F.relu(self.l1(x))
...             h2 = F.relu(self.l2(h1))
...             y = self.l3(h2)
...             return y
```

This link uses relu() as an activation function. Note that the 'l3' link is the final linear layer whose output corresponds to scores for the ten digits.

In order to compute loss values or evaluate the accuracy of the predictions, we define a classifier chain on top of the above MLP chain:

```python
>>> class Classifier(Chain):
...     def __init__(self, predictor):
...         super(Classifier, self).__init__()
...         with self.init_scope():
...             self.predictor = predictor
...             ...  
...     def __call__(self, x, t):
```

Chapter 2. Chainer Tutorial
This Classifier class computes accuracy and loss, and returns the loss value. The pair of arguments \( x \) and \( t \) corresponds to each example in the datasets (a tuple of an image and a label). \texttt{softmax_cross_entropy()} computes the loss value given prediction and ground truth labels. \texttt{accuracy()} computes the prediction accuracy. We can set an arbitrary predictor link to an instance of the classifier.

The \texttt{report()} function reports the loss and accuracy values to the trainer. For the detailed mechanism of collecting training statistics, see \texttt{Reporter}. You can also collect other types of observations like activation statistics in a similar ways.

Note that a class similar to the Classifier above is defined as \texttt{chainer.links.Classifier}. So instead of using the above example, we will use this predefined Classifier chain.

```python
>>> model = L.Classifier(MLP(100, 10))  # the input size, 784, is inferred
>>> optimizer = optimizers.SGD()
>>> optimizer.setup(model)
```

Now we can build a trainer object.

```python
>>> updater = training.StandardUpdater(train_iter, optimizer)
>>> trainer = training.Trainer(updater, (20, 'epoch'), out='result')
```

The second argument \( (20, \text{'epoch'}) \) represents the duration of training. We can use either \texttt{epoch} or \texttt{iteration} as the unit. In this case, we train the multi-layer perceptron by iterating over the training set 20 times.

In order to invoke the training loop, we just call the \texttt{run()} method.

```python
>>> trainer.run()
```

This method executes the whole training sequence.

The above code just optimizes the parameters. In most cases, we want to see how the training proceeds, where we can use extensions inserted before calling the \texttt{run()} method.

```python
>>> trainer.extend(extensions.Evaluator(test_iter, model))
>>> trainer.extend(extensions.LogReport())
>>> trainer.extend(extensions.PrintReport(['epoch', 'main/accuracy', 'validation/main/accuracy']))
>>> trainer.extend(extensions.ProgressBar())
>>> trainer.run()
```

These extensions perform the following tasks:

- **Evaluator** Evaluates the current model on the test dataset at the end of every epoch. It automatically switches to the test mode (see \texttt{Configuring Chainer} for details), and so we do not have to take any special function for functions that behave differently in training/test modes (e.g. \texttt{dropout()}, \texttt{BatchNormalization}).

- **LogReport** Accumulates the reported values and emits them to the log file in the output directory.

- **PrintReport** Prints the selected items in the LogReport.

- **ProgressBar** Shows the progress bar.
There are many extensions implemented in the `chainer.training.extensions` module. The most important one that is not included above is `snapshot()`, which saves the snapshot of the training procedure (i.e., the Trainer object) to a file in the output directory.

The example code in the `examples/mnist` directory additionally contains GPU support, though the essential part is the same as the code in this tutorial. We will review in later sections how to use GPU(s).

## How to Write a New Network

### Convolutional Network for Visual Recognition Tasks

In this section, you will learn how to write

- A small convolutional network with a model class that is inherited from `Chain`,
- A large convolutional network that has several building block networks with `ChainList`.

After reading this section, you will be able to:

- Write your own original convolutional network in Chainer

A convolutional network (ConvNet) is mainly comprised of convolutional layers. This type of network is commonly used for various visual recognition tasks, e.g., classifying hand-written digits or natural images into given object classes, detecting objects from an image, and labeling all pixels of an image with the object classes (semantic segmentation), and so on.

In such tasks, a typical ConvNet takes a set of images whose shape is \((N, C, H, W)\), where

- \(N\) denotes the number of images in a mini-batch,
- \(C\) denotes the number of channels of those images,
- \(H\) and \(W\) denote the height and width of those images,

respectively. Then, it typically outputs a fixed-sized vector as membership probabilities over the target object classes. It also can output a set of feature maps that have the corresponding size to the input image for a pixel labeling task, etc.

---

**Note:** The below example code assumes that some packages are already imported. Please see the details here: *Introduction to Chainer.*

---

### LeNet5

Here, let’s start by defining LeNet5 [LeCun98] in Chainer. This is a ConvNet model that has 5 layers comprised of 3 convolutional layers and 2 fully-connected layers. This was proposed to classify hand-written digit images in 1998. In Chainer, the model can be written as follows:

```python
class LeNet5(Chain):
    def __init__(self):
        super(LeNet5, self).__init__()
        with self.init_scope():
            self.conv1 = L.Convolution2D(
                in_channels=1, out_channels=6, ksize=5, stride=1)
            self.conv2 = L.Convolution2D(
                in_channels=6, out_channels=16, ksize=5, stride=1)
            self.conv3 = L.Convolution2D(
```
A typical way to write your network is creating a new class inherited from `Chain` class. When defining your model in this way, typically, all the layers which have trainable parameters are registered to the model by assigning the objects of `Link` as an attribute.

The model class is instantiated before the forward and backward computations. To give input images and label vectors simply by calling the model object like a function, `__call__()` is usually defined in the model class. This method performs the forward computation of the model. Chainer uses the powerful autograd system for any computational graphs written with `Function`s and `Link`s (actually a `Link` calls a corresponding `Function` inside of it), so that you don’t need to explicitly write the code for backward computations in the model. Just prepare the data, then give it to the model. The way this works is the resulting output `Variable` from the forward computation has a `backward()` method to perform autograd. In the above model, `__call__()` has a if statement at the end to switch its behavior by the Chainer’s running mode, i.e., training mode or not. Chainer presents the running mode as a global variable `chainer.config.train`. When it’s in training mode, `__call__()` returns the output value of the last layer as is to compute the loss later on, otherwise it returns a prediction result by calculating `softmax()`.

**Note:** In Chainer v1, if a function or link behaved differently in training and other modes, it was common that it held an attribute that represented its running mode or was provided with the mode from outside as an argument. In Chainer v2, it is recommended to use the global configuration `chainer.config.train` to switch the running mode.

If you don’t want to write `conv1` and the other layers more than once, you can also write the model like in this way:

```python
class LeNet5(Chain):
    def __init__(self):
        super(LeNet5, self).__init__()
        net = [['conv1', L.Convolution2D(1, 6, 5, 1)]
               + [('sigm1', F.Sigmoid())
                 + ['mpool1', F.MaxPooling2D(2, 2)]
                 + ['conv2', L.Convolution2D(6, 16, 5, 1)]
                 + [('sigm2', F.Sigmoid())
                 + ['mpool2', F.MaxPooling2D(2, 2)]
                 + ['conv3', L.Convolution2D(16, 120, 4, 1)]
                 + [('sigm3', F.Sigmoid())
                 + ['mpool3', F.MaxPooling2D(2, 2)]
                 + ['fc4', L.Linear(None, 84)]
                 + [('sigm4', F.Sigmoid())
                 + ['fc5', L.Linear(84, 10)]
                 with self.init_scope():
                     for n in net:
                         if not n[0].startswith('_):
                             setattr(self, n[0], n[1])
```
This code creates a list of all Link s and Function s after calling its superclass’s constructor. Then the elements of the list are registered to this model as trainable layers when the name of an element doesn’t start with _ character. This operation can be freely replaced with many other ways because those names are just designed to select Link s only from the list net easily. Function doesn’t have any trainable parameters, so that we can’t register it to the model, but we want to use Function s for constructing a forward path. The list net is stored as an attribute forward to refer it in __call__(). In __call__(), it retrieves all layers in the network from self.forward sequentially regardless of what types of object (Link or Function) it is, and gives the input variable or the intermediate output from the previous layer to the current layer. The last part of the __call__() to switch its behavior by the training/inference mode is the same as the former way.

**Ways to calculate loss**

When you train the model with label vector t, the loss should be calculated using the output from the model. There also are several ways to calculate the loss:

```python
model = LeNet5()

# Input data and label
x = np.random.rand(32, 1, 28, 28).astype(np.float32)
t = np.random.randint(0, 10, size=(32,)).astype(np.int32)

# Forward computation
y = model(x)

# Loss calculation
loss = F.softmax_cross_entropy(y, t)
```

This is a primitive way to calculate a loss value from the output of the model. On the other hand, the loss computation can be included in the model itself by wrapping the model object (Chain or ChainList object) with a class inherited from Chain. The outer Chain should take the model defined above and register it with init_scope(). Chain is actually inherited from Link, so that Chain itself can also be registered as a trainable Link to another Chain. Actually, Classifier class to wrap the model and add the loss computation to the model already exists. Actually, there is already a Classifier class that can be used to wrap the model and include the loss computation as well. It can be used like this:

```python
model = L.Classifier(LeNet5())

# Forward & Loss calculation
loss = model(x, t)
```

This class takes a model object as an input argument and registers it to a predictor property as a trained parameter. As shown above, the returned object can then be called like a function in which we pass x and t as the input arguments and the resulting loss value (which we recall is a Variable) is returned.
See the detailed implementation of `Classifier` from here: `chainer.links.Classifier` and check the implementation by looking at the source.

From the above examples, we can see that Chainer provides the flexibility to write our original network in many different ways. Such flexibility intends to make it intuitive for users to design new and complex models.

**VGG16**

Next, let’s write some larger models in Chainer. When you write a large network consisting of several building block networks, `ChainList` is useful. First, let’s see how to write a VGG16 [Simonyan14] model.

```python
class VGG16(chainer.ChainList):
    def __init__(self):
        w = chainer.initializers.HeNormal()
        super(VGG16, self).__init__(
            VGGBlock(64),
            VGGBlock(128),
            VGGBlock(256, 3),
            VGGBlock(512, 3),
            VGGBlock(512, 3, True))

    def __call__(self, x):
        for f in self.children():
            x = f(x)
        if chainer.config.train:
            return x
        return F.softmax(x)

class VGGBlock(chainer.Chain):
    def __init__(self, n_channels, n_convs=2, fc=False):
        w = chainer.initializers.HeNormal()
        super(VGGBlock, self).__init__()
        with self.init_scope():
            self.conv1 = L.Convolution2D(None, n_channels, 3, 1, 1, initialW=w)
            self.conv2 = L.Convolution2D(n_channels, n_channels, 3, 1, 1, initialW=w)
            if n_convs == 3:
                self.conv3 = L.Convolution2D(n_channels, n_channels, 3, 1, 1, initialW=w)
            if fc:
                self.fc4 = L.Linear(None, 4096, initialW=w)
                self.fc5 = L.Linear(4096, 4096, initialW=w)
                self.fc6 = L.Linear(4096, 1000, initialW=w)
        self.n_convs = n_convs
        self.fc = fc

    def __call__(self, x):
        h = F.relu(self.conv1(x))
        h = F.relu(self.conv2(h))
        if self.n_convs == 3:
            h = F.relu(self.conv3(h))
        h = F.max_pooling_2d(h, 2, 2)
        if self.fc:
            h = F.dropout(F.relu(self.fc4(h)))
            h = F.dropout(F.relu(self.fc5(h)))
```

2.2. How to Write a New Network 19
That’s it. VGG16 is a model which won the 1st place in classification + localization task at ILSVRC 2014, and since then, has become one of the standard models for many different tasks as a pre-trained model. This has 16-layers, so it’s called “VGG-16”, but we can write this model without writing all layers independently. Since this model consists of several building blocks that have the same architecture, we can build the whole network by re-using the building block definition. Each part of the network is consisted of 2 or 3 convolutional layers and activation function (relu()) following them, and max_pooling_2d() operations. This block is written as VGGBlock in the above example code. And the whole network just calls this block one by one in sequential manner.

ResNet152

How about ResNet? ResNet [He16] came in the following year’s ILSVRC. It is a much deeper model than VGG16, having up to 152 layers. This sounds super laborious to build, but it can be implemented in almost same manner as VGG16. In the other words, it’s easy. One possible way to write ResNet-152 is:

```python
class ResNet152(chainer.Chain):
    def __init__(self, n_blocks=[3, 8, 36, 3]):
        w = chainer.initializers.HeNormal()
        super(ResNet152, self).__init__()
        conv1=L.Convolution2D(None, 64, 7, 2, 3, initialW=w, nobias=True),
        bn1=L.BatchNormalization(64),
        res2=ResBlock(n_blocks[0], 64, 64, 256, 1),
        res3=ResBlock(n_blocks[1], 256, 128, 512),
        res4=ResBlock(n_blocks[2], 512, 256, 1024),
        res5=ResBlock(n_blocks[3], 1024, 512, 2048),
        fc6=L.Linear(2048, 1000)
    def __call__(self, x):
        h = self.bn1(self.conv1(x))
        h = F.max_pooling_2d(F.relu(h), 2, 2)
        h = self.res2(h)
        h = self.res3(h)
        h = self.res4(h)
        h = self.res5(h)
        h = F.average_pooling_2d(h, h.shape[2:], stride=1)
        h = self.fc6(h)
        if chainer.config.train:
            return h
        return F.softmax(h)

class ResBlock(chainer.ChainList):
    def __init__(self, n_layers, n_in, n_mid, n_out, stride=2):
        w = chainer.initializers.HeNormal()
        super(ResBlock, self).__init__()
        self.add_link(BottleNeck(n_in, n_mid, n_out, stride, True))
        for _ in range(n_layers - 1):
            self.add_link(BottleNeck(n_out, n_mid, n_out))
    def __call__(self, x):
        for f in self.children():
            x = f(x)
        return x
```
class BottleNeck(chainer.Chain):
    def __init__(self, n_in, n_mid, n_out, stride=1, proj=False):
        w = chainer.initializers.HeNormal()
        super(BottleNeck, self).__init__()
        with self.init_scope():
            self.conv1x1a = L.Convolution2D(n_in, n_mid, 1, stride, 0, initialW=w, nobias=True)
            self.conv3x3b = L.Convolution2D(n_mid, n_mid, 3, 1, 1, initialW=w, nobias=True)
            self.conv1x1c = L.Convolution2D(n_mid, n_out, 1, 1, 0, initialW=w, nobias=True)
            self.bn_a = L.BatchNormalization(n_mid)
            self.bn_b = L.BatchNormalization(n_mid)
            self.bn_c = L.BatchNormalization(n_out)
            if proj:
                self.conv1x1r = L.Convolution2D(n_in, n_out, 1, stride, 0, initialW=w, nobias=True)
                self.bn_r = L.BatchNormalization(n_out)
        self.proj = proj

    def __call__(self, x):
        h = F.relu(self.bn_a(self.conv1x1a(x)))
        h = F.relu(self.bn_b(self.conv3x3b(h)))
        h = self.bn_c(self.conv1x1c(h))
        if self.proj:
            x = self.bn_r(self.conv1x1r(x))
        return F.relu(h + x)

In the BottleNeck class, depending on the value of the proj argument supplied to the initializer, it will conditionally compute a convolutional layer conv1x1r which will extend the number of channels of the input x to be equal to the number of channels of the output of conv1x1c, and followed by a batch normalization layer before the final ReLU layer. Writing the building block in this way improves the re-usability of a class. It switches not only the behavior in __class__() by flags but also the parameter registration. In this case, when proj is False, the BottleNeck doesn't have conv1x1r and bn_r layers, so the memory usage would be efficient compared to the case when it registers both anyway and just ignore them if proj is False.

Using nested Chains and ChainList for sequential part enables us to write complex and very deep models easily.

Use Pre-trained Models

Various ways to write your models were described above. It turns out that VGG16 and ResNet are very useful as general feature extractors for many kinds of tasks, including but not limited to image classification. So, Chainer provides you with the pre-trained VGG16 and ResNet-50/101/152 models with a simple API. You can use these models as follows:

```python
from chainer.links import VGG16Layers
model = VGG16Layers()
```

When VGG16Layers is instantiated, the pre-trained parameters are automatically downloaded from the author’s server. So you can immediately start to use VGG16 with pre-trained weight as a good image feature extractor. See the details of this model here: chainer.links.VGG16Layers.

In the case of ResNet models, there are three variations differing in the number of layers. We have chainer.links.ResNet50, chainer.links.ResNet101, and chainer.links.ResNet152 models with easy parameter
loading feature. ResNet’s pre-trained parameters are not available for direct downloading, so you need to download
the weight from the author’s web page first, and then place it into the dir `$CHAINER_DATSET_ROOT/pfnet/
chainer/models` or your favorite place. Once the preparation is finished, the usage is the same as VGG16:

```python
from chainer.links import ResNet152Layers
model = ResNet152Layers()
```

Please see the details of usage and how to prepare the pre-trained weights for ResNet here: chainer.links.
ResNet50

References

Recurrent Nets and their Computational Graph

In this section, you will learn how to write

- recurrent nets with full backprop,
- recurrent nets with truncated backprop,
- evaluation of networks with few memory.

After reading this section, you will be able to:

- Handle input sequences of variable length
- Truncate upper stream of the network during forward computation
- Use no-backprop mode to prevent network construction

Recurrent Nets

Recurrent nets are neural networks with loops. They are often used to learn from sequential input/output. Given
an input stream $x_1, x_2, \ldots, x_t, \ldots$ and the initial state $h_0$, a recurrent net iteratively updates its state
by $h_t = f(x_t, h_{t-1})$, and at some or every point in time $t$, it outputs $y_t = g(h_t)$. If we expand the procedure along the time axis, it looks
like a regular feed-forward network except that same parameters are repeatedly used within the network.

Here we learn how to write a simple one-layer recurrent net. The task is language modeling: given a finite sequence
of words, we want to predict the next word at each position without peeking the successive words. Suppose there
are 1,000 different word types, and that we use 100 dimensional real vectors to represent each word (a.k.a. word
embedding).

Let’s start from defining the recurrent neural net language model (RNNLM) as a chain. We can use the chainer.
links.LSTM link that implements a fully-connected stateful LSTM layer. This link looks like an ordinary fully-
connected layer. On construction, you pass the input and output size to the constructor:

```python
>>> l = L.LSTM(100, 50)
```

Then, call on this instance `l(x)` executes `one step of LSTM layer`:

```python
>>> l.reset_state()
>>> x = Variable(np.random.randn(10, 100).astype(np.float32))
>>> y = l(x)
```

Do not forget to reset the internal state of the LSTM layer before the forward computation! Every recurrent layer holds
its internal state (i.e. the output of the previous call). At the first application of the recurrent layer, you must reset the
internal state. Then, the next input can be directly fed to the LSTM instance:
Based on this LSTM link, let’s write our recurrent network as a new chain:

```python
class RNN(Chain):
    def __init__(self):
        super(RNN, self).__init__()
        with self.init_scope():
            self.embed = L.EmbedID(1000, 100)  # word embedding
            self.mid = L.LSTM(100, 50)  # the first LSTM layer
            self.out = L.Linear(50, 1000)  # the feed-forward output layer

    def reset_state(self):
        self.mid.reset_state()

    def __call__(self, cur_word):
        # Given the current word ID, predict the next word.
        x = self.embed(cur_word)
        h = self.mid(x)
        y = self.out(h)
        return y
```

Here `EmbedID` is a link for word embedding. It converts input integers into corresponding fixed-dimensional embedding vectors. The last linear link `out` represents the feed-forward output layer.

The `RNN` chain implements a one-step-forward computation. It does not handle sequences by itself, but we can use it to process sequences by just feeding items in a sequence straight to the chain.

Suppose we have a list of word variables `x_list`. Then, we can compute loss values for the word sequence by simple for loop.

```python
def compute_loss(x_list):
    loss = 0
    for cur_word, next_word in zip(x_list, x_list[1:]):
        loss += model(cur_word, next_word)
    return loss
```

Of course, the accumulated loss is a Variable object with the full history of computation. So we can just call its `backward()` method to compute gradients of the total loss according to the model parameters:

```python
# Suppose we have a list of word variables x_list.
rnn.reset_state()
model.cleargrads()
loss = compute_loss(x_list)
loss.backward()
optimizer.update()
```

Or equivalently we can use the `compute_loss` as a loss function:

```python
rnn.reset_state()
optimizer.update(compute_loss, x_list)
```
Truncate the Graph by Unchaining

Learning from very long sequences is also a typical use case of recurrent nets. Suppose the input and state sequence is too long to fit into memory. In such cases, we often truncate the backpropagation into a short time range. This technique is called truncated backprop. It is heuristic, and it makes the gradients biased. However, this technique works well in practice if the time range is long enough.

How to implement truncated backprop in Chainer? Chainer has a smart mechanism to achieve truncation, called backward unchaining. It is implemented in the Variable.unchain_backward() method. Backward unchaining starts from the Variable object, and it chops the computation history backwards from the variable. The chopped variables are disposed automatically (if they are not referenced explicitly from any other user object). As a result, they are no longer a part of computation history, and are not involved in backprop anymore.

Let's write an example of truncated backprop. Here we use the same network as the one used in the previous subsection. Suppose we are given a very long sequence, and we want to run backprop truncated at every 30 time steps. We can write truncated backprop using the model defined above:

```python
loss = 0
count = 0
seqlen = len(x_list[1:])

rnn.reset_state()
for cur_word, next_word in zip(x_list, x_list[1:]):
    loss += model(cur_word, next_word)
    count += 1
    if count % 30 == 0 or count == seqlen:
        model.cleargrads()
        loss.backward()
        loss.unchain_backward()
        optimizer.update()
```

State is updated at `model()`, and the losses are accumulated to `loss` variable. At each 30 steps, backprop takes place at the accumulated loss. Then, the `unchain_backward()` method is called, which deletes the computation history backward from the accumulated loss. Note that the last state of `model` is not lost, since the RNN instance holds a reference to it.

The implementation of truncated backprop is simple, and since there is no complicated trick on it, we can generalize this method to different situations. For example, we can easily extend the above code to use different schedules between backprop timing and truncation length.

Network Evaluation without Storing the Computation History

On evaluation of recurrent nets, there is typically no need to store the computation history. While unchaining enables us to walk through unlimited length of sequences with limited memory, it is a bit of a work-around.

As an alternative, Chainer provides an evaluation mode of forward computation which does not store the computation history. This is enabled by just calling `no_backprop_mode()` context:

```python
with chainer.no_backprop_mode():
    x_list = [Variable(...) for _ in range(100)] # list of 100 words
    loss = compute_loss(x_list)
```

Note that we cannot call `loss.backward()` to compute the gradient here, since the variable created in the no-backprop context does not remember the computation history.

No-backprop context is also useful to evaluate feed-forward networks to reduce the memory footprint.
We can combine a fixed feature extractor network and a trainable predictor network using `no_backprop_mode()`. For example, suppose we want to train a feed-forward network `predictor_func`, which is located on top of another fixed pre-trained network `fixed_func`. We want to train `predictor_func` without storing the computation history for `fixed_func`. This is simply done by following code snippets (suppose `x_data` and `y_data` indicate input data and label, respectively):

```python
with chainer.no_backprop_mode():
    x = Variable(x_data)
    feat = fixed_func(x)
    y = predictor_func(feat)
y.backward()
```

At first, the input variable `x` is in no-backprop mode, so `fixed_func` does not memorize the computation history. Then `predictor_func` is executed in backprop mode, i.e., with memorizing the history of computation. Since the history of computation is only memorized between variables `feat` and `y`, the backward computation stops at the `feat` variable.

### Making it with Trainer

The above codes are written with plain Function/Variable APIs. When we write a training loop, it is better to use Trainer, since we can then easily add functionalities by extensions.

Before implementing it on Trainer, let’s clarify the training settings. We here use Penn Tree Bank dataset as a set of sentences. Each sentence is represented as a word sequence. We concatenate all sentences into one long word sequence, in which each sentence is separated by a special word `<eos>`, which stands for “End of Sequence”. This dataset is easily obtained by `chainer.datasets.get_ptb_words()`. This function returns train, validation, and test dataset, each of which is represented as a long array of integers. Each integer represents a word ID.

Our task is to learn a recurrent neural net language model from the long word sequence. We use words in different locations to form mini-batches. It means we maintain `B` indices pointing to different locations in the sequence, read from these indices at each iteration, and increment all indices after the read. Of course, when one index reaches the end of the whole sequence, we turn the index back to 0.

In order to implement this training procedure, we have to customize the following components of Trainer:

- Iterator. Built-in iterators do not support reading from different locations and aggregating them into a mini-batch.
- Update function. The default update function does not support truncated BPTT.

When we write a dataset iterator dedicated to the dataset, the dataset implementation can be arbitrary; even the interface is not fixed. On the other hand, the iterator must support the `Iterator` interface. The important methods and attributes to implement are `batch_size`, `epoch`, `epoch_detail`, `is_new_epoch`, `iteration`, `__next__`, and `serialize`. Following is a code from the official example in the `examples/ptb` directory.

```python
from __future__ import division

class ParallelSequentialIterator(chainer.dataset.Iterator):
    def __init__(self, dataset, batch_size, repeat=True):
        self.dataset = dataset
        self.batch_size = batch_size
        self.epoch = 0
        self.is_new_epoch = False
        self.repeat = repeat
        self.offsets = [i * len(dataset) // batch_size
                        for i in range(batch_size)]
        self.iteration = 0

    def __next__(self):
```
length = len(self.dataset)
if not self.repeat and self.iteration * self.batch_size >= length:
    raise StopIteration
cur_words = self.get_words()
self.iteration += 1
next_words = self.get_words()

epoch = self.iteration * self.batch_size // length
self.is_new_epoch = self.epoch < epoch
if self.is_new_epoch:
    self.epoch = epoch

return list(zip(cur_words, next_words))

@property
def epoch_detail(self):
    return self.iteration * self.batch_size / len(self.dataset)

def get_words(self):
    return [self.dataset[(offset + self.iteration) % len(self.dataset)]
            for offset in self.offsets]

def serialize(self, serializer):
    self.iteration = serializer('iteration', self.iteration)
    self.epoch = serializer('epoch', self.epoch)

train_iter = ParallelSequentialIterator(train, 20)
val_iter = ParallelSequentialIterator(val, 1, repeat=False)

Although the code is slightly long, the idea is simple. First, this iterator creates \texttt{offsets} pointing to positions equally spaced within the whole sequence. The \( i \)-th examples of mini-batches refer the sequence with the \( i \)-th offset. The iterator returns a list of tuples of the current words and the next words. Each mini-batch is converted to a tuple of integer arrays by the \texttt{concat_examples} function in the standard updater (see the previous tutorial).

Backprop Through Time is implemented as follows.

def update_bptt(updater):
    loss = 0
    for i in range(35):
        batch = train_iter.__next__()
        x, t = chainer.dataset.concat_examples(batch)
        loss += model(chainer.Variable(x), chainer.Variable(t))

    model.cleargrads()
    loss.backward()
    loss.unchain_backward()  # truncate
    optimizer.update()

updater = training.StandardUpdater(train_iter, optimizer, update_bptt)

In this case, we update the parameters on every 35 consecutive words. The call of \texttt{unchain_backward} cuts the history of computation accumulated to the LSTM links. The rest of the code for setting up Trainer is almost same as one given in the previous tutorial.

In this section we have demonstrated how to write recurrent nets in Chainer and some fundamental techniques to manage the history of computation (a.k.a. computational graph). The example in the \texttt{examples/ptb} directory
implement truncated backprop learning of a LSTM language model from the Penn Treebank corpus. In the next section, we will review how to use GPU(s) in Chainer.

**Using GPU(s) in Chainer**

In this section, you will learn about the following things:

- Relationship between Chainer and CuPy
- Basics of CuPy
- Single-GPU usage of Chainer
- Multi-GPU usage of model-parallel computing
- Multi-GPU usage of data-parallel computing

After reading this section, you will be able to:

- Use Chainer on a CUDA-enabled GPU
- Write model-parallel computing in Chainer
- Write data-parallel computing in Chainer

**Relationship between Chainer and CuPy**

**Note:** From v2.0.0, CuPy is turned into a separate package and repository. Even if you have CUDA installed in your environment, you have to install CuPy separately to use GPUs. See [Enable CUDA/cuDNN support](#) for the way to set up CUDA support.

Chainer uses CuPy as its backend for GPU computation. In particular, the `cupy.ndarray` class is the GPU array implementation for Chainer. CuPy supports a subset of features of NumPy with a compatible interface. It enables us to write a common code for CPU and GPU. It also supports PyCUDA-like user-defined kernel generation, which enables us to write fast implementations dedicated to GPU.

**Note:** The `chainer.cuda` module imports many important symbols from CuPy. For example, the cupy namespace is referred as `cuda.cupy` in the Chainer code. Note that the `chainer.cuda` module can be imported even if CUDA is not installed.

Chainer uses a memory pool for GPU memory allocation. As shown in the previous sections, Chainer constructs and destructs many arrays during learning and evaluating iterations. It is not well suited for CUDA architecture, since memory allocation and release in CUDA (i.e. `cudaMalloc` and `cudaFree` functions) synchronize CPU and GPU computations, which hurts performance. In order to avoid memory allocation and deallocation during the computation, Chainer uses CuPy’s memory pool as the standard memory allocator. Chainer changes the default allocator of CuPy to the memory pool, so user can use functions of CuPy directly without dealing with the memory allocator.

**Basics of `cupy.ndarray`**

See the document of CuPy for the basic usage of `cupy.ndarray`

CuPy is a GPU array backend that implements a subset of NumPy interface. The `cupy.ndarray` class is in its core, which is a compatible GPU alternative of `numpy.ndarray`. CuPy implements many functions on `cupy.ndarray`
objects. See the reference for the supported subset of NumPy API. Understanding NumPy might help utilizing most features of CuPy. See the NumPy documentation for learning it.

The main difference of `cupy.ndarray` from `numpy.ndarray` is that the content is allocated on the device memory. The allocation takes place on the current device by default. The current device can be changed by `cupy.cu.

device` object as follows:

```python
with cupy.cuda.Device(1):
    x_on_gpu1 = cupy.array([1, 2, 3, 4, 5])
```

Most operations of CuPy is done on the current device. Be careful that it causes an error to process an array on a non-current device.

Chainer provides some convenient functions to automatically switch and choose the device. For example, the `chainer.cuda.to_gpu()` function copies a `numpy.ndarray` object to a specified device:

```python
x_cpu = np.ones((5, 4, 3), dtype=np.float32)
x_gpu = cuda.to_gpu(x_cpu, device=1)
```

It is equivalent to the following code using CuPy:

```python
x_cpu = np.ones((5, 4, 3), dtype=np.float32)
with cupy.cuda.Device(1):
    x_gpu = cupy.array(x_cpu)
```

Moving a device array to the host can be done by `chainer.cuda.to_cpu()` as follows:

```python
x_cpu = cuda.to_cpu(x_gpu)
```

It is equivalent to the following code using CuPy:

```python
with x_gpu.device:
    x_cpu = x_gpu.get()
```

Note: The `with` statements in these codes are required to select the appropriate CUDA device. If user uses only one device, these device switching is not needed. `chainer.cuda.to_cpu()` and `chainer.cuda.to_gpu()` functions automatically switch the current device correctly.

Chainer also provides a convenient function `chainer.cuda.get_device_from_id()` and `chainer.cuda.get_device_from_array()` to select a device. The former function accepts an integer or None. When None is given, it returns a dummy device object. Otherwise, it returns a corresponding device object. The latter function accepts CuPy array or NumPy array. When a NumPy array is given, it returns a dummy device object. Otherwise, it returns a corresponding device object to the given CuPy array. The dummy device object also supports `with` statements like the above example but does nothing. Here are some other examples:

```python
cuda.get_device_from_id(1).use()
x_gpu1 = cupy.empty((4, 3), dtype='f')  # 'f' indicates float32
with cuda.get_device_from_id(1):
    x_gpu1 = cupy.empty((4, 3), dtype='f')
with cuda.get_device_from_array(x_gpu1):
    y_gpu1 = x_gpu + 1
```

Since it accepts NumPy arrays, we can write a function that accepts both NumPy and CuPy arrays with correct device switching:
The compatibility of CuPy with NumPy enables us to write CPU/GPU generic code. It can be made easy by the `chainer.cuda.get_array_module()` function. This function returns the `numpy` or `cupy` module based on arguments. A CPU/GPU generic function is defined using it like follows:

```python
# Stable implementation of log(1 + exp(x))
def softplus(x):
    xp = cuda.get_array_module(x)
    return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

### Run Neural Networks on a Single GPU

Single-GPU usage is very simple. What you have to do is transferring `Link` and input arrays to the GPU beforehand. In this subsection, the code is based on our first MNIST example in this tutorial.

A `Link` object can be transferred to the specified GPU using the `to_gpu()` method.

This time, we make the number of input, hidden, and output units configurable. The `to_gpu()` method also accepts a device ID like `model.to_gpu(0)`. In this case, the link object is transferred to the appropriate GPU device. The current device is used by default.

If we use `chainer.training.Trainer`, what we have to do is just let the updater know the device ID to send each mini-batch.

```python
updater = training.StandardUpdater(train_iter, optimizer, device=0)
trainer = training.Trainer(updater, (20, 'epoch'), out='result')
```

We also have to specify the device ID for an evaluator extension as well.

```python
trainer.extend(extensions.Evaluator(test_iter, model, device=0))
```

When we write down the training loop by hand, we have to transfer each mini-batch to the GPU manually:

```python
model.to_gpu()
batchsize = 100
datasize = len(x_train)
for epoch in range(20):
    print('epoch %d' % epoch)
    indexes = np.random.permutation(datasize)
    for i in range(0, datasize, batchsize):
        x = Variable(cuda.to_gpu(x_train[indexes[i : i + batchsize]]))
        t = Variable(cuda.to_gpu(y_train[indexes[i : i + batchsize]]))
        optimizer.update(model, x, t)
```

### Model-parallel Computation on Multiple GPUs

Parallelization of machine learning is roughly classified into two types called “model-parallel” and “data-parallel”. Model-parallel means parallelizations of the computations inside the model. In contrast, data-parallel means parallelizations using data sharding. In this subsection, we show how to use the model-parallel approach on multiple GPUs in Chainer.
Recall the MNIST example. Now suppose that we want to modify this example by expanding the network to 6 layers with 2000 units each using two GPUs. In order to make multi-GPU computation efficient, we only make the two GPUs communicate at the third and sixth layer. The overall architecture looks like the following diagram:

```
(GPU0) input --+--> l1 --> l2 --> l3 --+--> l4 --> l5 --> l6 --+--> output
        |                     |
(GPU1) +--> l1 --> l2 --> l3 --+--> l4 --> l5 --> l6 --+
```

We can use the above MLP chain as following diagram:

```
(GPU0) input --+--> mlp1 --+--> mlp2 --+--> output
        |                     |
(GPU1) +--> mlp1 --+--> mlp2 --+
```

Let's write a link for the whole network.

```python
class ParallelMLP(Chain):
    def __init__(self):
        super(ParallelMLP, self).__init__ Perform
        # the input size, 784, is inferred
        mlp1_gpu0=MLP(1000, 2000).to_gpu(0),
        mlp1_gpu1=MLP(1000, 2000).to_gpu(1),

        # the input size, 2000, is inferred
        mlp2_gpu0=MLP(1000, 10).to_gpu(0),
        mlp2_gpu1=MLP(1000, 10).to_gpu(1),
    
    def __call__(self, x):
        # assume x is on GPU 0
        z0 = self.mlp1_gpu0(x)
        z1 = self.mlp1_gpu1(F.copy(x, 1))

        # sync
        h0 = F.relu(z0 + F.copy(z1, 0))
        h1 = F.relu(z1 + F.copy(z0, 1))

        y0 = self.mlp2_gpu0(h0)
        y1 = self.mlp2_gpu1(h1)

        # sync
        y = y0 + F.copy(y1, 0)
        return y  # output is on GPU0
```

Recall that the `Link.to_gpu()` method returns the link itself. The `copy()` function copies an input variable to specified GPU device and returns a new variable on the device. The copy supports backprop, which just reversely transfers an output gradient to the input device.

**Note:** Above code is not parallelized on CPU, but is parallelized on GPU. This is because all the functions in the above code run asynchronously to the host CPU.

An almost identical example code can be found at `examples/mnist/train_mnist_model_parallel.py`. 

30 Chapter 2. Chainer Tutorial
Data-parallel Computation on Multiple GPUs with Trainer

Data-parallel computation is another strategy to parallelize online processing. In the context of neural networks, it means that a different device does computation on a different subset of the input data. In this subsection, we review the way to achieve data-parallel learning on two GPUs.

Suppose again our task is the MNIST example. This time we want to directly parallelize the three-layer network. The most simple form of data-parallelization is parallelizing the gradient computation for a distinct set of data. First, define a model and optimizer instances:

```python
model = L.Classifier(MLP(1000, 10))  # the input size, 784, is inferred
optimizer = optimizers.SGD()
optimizer.setup(model)
```

Recall that the MLP link implements the multi-layer perceptron, and the Classifier link wraps it to provide a classifier interface. We used StandardUpdater in the previous example. In order to enable data-parallel computation with multiple GPUs, we only have to replace it with ParallelUpdater.

```python
updater = training.ParallelUpdater(train_iter, optimizer,
                                 devices={'main': 0, 'second': 1})
```

The devices option specifies which devices to use in data-parallel learning. The device with name 'main' is used as the main device. The original model is sent to this device, so the optimization runs on the main device. In the above example, the model is also cloned and sent to GPU 1. Half of each mini-batch is fed to this cloned model. After every backward computation, the gradient is accumulated into the main device, the parameter update runs on it, and then the updated parameters are sent to GPU 1 again.

See also the example code in examples/mnist/train_mnist_data_parallel.py.

Data-parallel Computation on Multiple GPUs without Trainer

We here introduce a way to write data-parallel computation without the help of Trainer. Most users can skip this section. If you are interested in how to write a data-parallel computation by yourself, this section should be informative. It is also helpful to, e.g., customize the ParallelUpdater class.

We again start from the MNIST example. At this time, we use a suffix like _0 and _1 to distinguish objects on each device. First, we define a model.

```python
model_0 = L.Classifier(MLP(1000, 10))  # the input size, 784, is inferred
```

We want to make two copies of this instance on different GPUs. The Link.to_gpu() method runs in place, so we cannot use it to make a copy. In order to make a copy, we can use Link.copy() method.

```python
import copy
model_1 = copy.deepcopy(model_0)
model_0.to_gpu(0)
model_1.to_gpu(1)
```

The Link.copy() method copies the link into another instance. It just copies the link hierarchy, and does not copy the arrays it holds.

Then, set up an optimizer:

```python
optimizer = optimizers.SGD()
optimizer.setup(model_0)
```
Here we use the first copy of the model as the master model. Before its update, gradients of model_1 must be aggregated to those of model_0.

Then, we can write a data-parallel learning loop as follows:

```python
batchsize = 100
datasize = len(x_train)
for epoch in range(20):
    print('epoch %d' % epoch)
    indexes = np.random.permutation(datasize)
    for i in range(0, datasize, batchsize):
        x_batch = x_train[indexes[i : i + batchsize]]
        y_batch = y_train[indexes[i : i + batchsize]]
        x0 = Variable(cuda.to_gpu(x_batch[:batchsize//2], 0))
        t0 = Variable(cuda.to_gpu(y_batch[:batchsize//2], 0))
        x1 = Variable(cuda.to_gpu(x_batch[batchsize//2:], 1))
        t1 = Variable(cuda.to_gpu(y_batch[batchsize//2:], 1))
        loss_0 = model_0(x0, t0)
        loss_1 = model_1(x1, t1)
        model_0.cleargrads()
        model_1.cleargrads()
        loss_0.backward()
        loss_1.backward()
        model_0.addgrads(model_1)
        optimizer.update()
        model_1.copyparams(model_0)
```

Do not forget to clear the gradients of both model copies! One half of the mini-batch is forwarded to GPU 0, the other half to GPU 1. Then the gradients are accumulated by the `Link.addgrads()` method. This method adds the gradients of a given link to those of the self. After the gradients are prepared, we can update the optimizer in usual way. Note that the update only modifies the parameters of model_0. So we must manually copy them to model_1 using `Link.copyparams()` method.

**Note:** If the batch size used in one model remain the same, the scale of the gradient is roughly proportional to the number of models, when we aggregate gradients from all models by `chainer.Link.addgrads()`. So you need to adjust the batch size and/or learning rate of the optimizer accordingly.

Now you can use Chainer with GPUs. All examples in the examples directory support GPU computation, so please refer to them if you want to know more practices on using GPUs. In the next section, we will show how to define a differentiable (i.e. backpropable) function on Variable objects. We will also show there how to write a simple (elementwise) CUDA kernel using Chainer’s CUDA utilities.

**Define your own function**

In this section, you will learn about the following things:

- How to define a function on variables
- Useful tools to write a function using a GPU
- How to test the function definition

After reading this section, you will be able to:

- Write your own functions
- Define simple kernels in the function definition

### Differentiable Functions

Chainer provides a collection of functions in the `functions` module. It covers typical use cases in deep learning, so many existing works can be implemented with them. On the other hand, deep learning is evolving rapidly and we cannot cover all possible functions to define unseen architectures. So it is important to learn how to define your own functions.

First, suppose we want to define an elementwise function \( f(x, y, z) = x \times y + z \). While it is possible to implement this equation using a combination of the \( \times \) and \( + \) functions, defining it as a single function may reduce memory consumption, so it is not only a toy example. Here we call this function `MulAdd`.

Let’s start with defining `MulAdd` working on the CPU. Any function must inherit the `Function` class. The skeleton of a function looks like:

```python
class MulAdd(Function):
    def forward_cpu(self, inputs):
        # do forward computation on CPU
        return some_tuple

    def backward_cpu(self, inputs, grad_outputs):
        # do backward computation on CPU
        return some_tuple
```

We must implement `forward_cpu()` and `backward_cpu()` methods. The non-self arguments of these functions are tuples of array(s), and these functions must return a tuple of array(s).

**Warning:** Be careful to return a tuple of arrays even if you have just one array to return.

MulAdd is simple and implemented as follows:

```python
class MulAdd(Function):
    def forward_cpu(self, inputs):
        x, y, z = inputs
        w = x * y + z
        return w,

    def backward_cpu(self, inputs, grad_outputs):
        x, y, z = inputs
        gw, = grad_outputs
        gx = y * gw
        gy = x * gw
        gz = gw
        return gx, gy, gz
```

As per the warning above, the `forward_cpu()` method returns a tuple of single element. Note that all arrays appearing in CPU functions are `numpy.ndarray`. The forward function is straightforward: It unpacks the input tuple,
computes the output, and packs it into a tuple. The backward function is a bit more complicated. Recall the rule of differentiation of multiplication. This example just implements the rule. Look at the return values, the function just packs the gradient of each input in same order and returns them.

By just defining the core computation of forward and backward, Function class provides a chaining logic on it (i.e. storing the history of computation, etc.).

**Note:** Assuming we implement a (forward) function \( y = f(x) \) which takes as input the vector \( x \in \mathbb{R}^n \) and produces as output a vector \( y \in \mathbb{R}^m \). Then the backward method has to compute

\[
\lambda_i = \sum_{j=1}^{m} \frac{\partial y_j}{\partial x_i} \gamma_j \quad \text{for} \quad i = 1 \ldots n
\]

where \( \gamma \) is the \texttt{grad_outputs}. Note, that the resulting vector \( \lambda \) must have the same shape as the arguments of the forward method.

Now let’s define the corresponding GPU methods. You can easily predict that the methods we have to write are named \texttt{forward_gpu()} and \texttt{backward_gpu()}:

```python
class MulAdd(Function):
    def forward_cpu(self, inputs):
        ...

    def backward_cpu(self, inputs, grad_outputs):
        ...

    def forward_gpu(self, inputs):
        x, y, z = inputs
        w = x * y + z
        return w,

    def backward_gpu(self, inputs, grad_outputs):
        x, y, z = inputs
        gw, = grad_outputs
        gx = y * gw
        gy = x * gw
        gz = gw
        return gx, gy, gz
```

In GPU methods, arrays are of type \texttt{cupy.ndarray}. We use arithmetic operators defined for this class. These operators implement the basic elementwise arithmetics.

You may find that the definitions of GPU methods are exactly same as those of CPU methods. In that case, we can reduce them to \texttt{forward()} and \texttt{backward()} methods:

```python
class MulAdd(Function):
    def forward(self, inputs):
        x, y, z = inputs
        w = x * y + z
        return w,

    def backward(self, inputs, grad_outputs):
        x, y, z = inputs
        gw, = grad_outputs
        gx = y * gw
```

34 Chapter 2. Chainer Tutorial
Since the `cupy.ndarray` class implements many methods of `numpy.ndarray`, we can write these unified methods in most cases.

The MulAdd function is used as follows:

```python
def muladd(x, y, z):
    return MulAdd()(x, y, z)
```

It looks a bit ugly: we have to explicitly instantiate MulAdd before applying it to variables. We also have to be careful that one instance of MulAdd must not be used multiple times, since it acts as a node in the computational graph. In Chainer, we often define a thin wrapper Python function that hide the instantiation:

```python
def muladd(x, y, z):
    return MulAdd()(x, y, z)
```

### Unified forward/backward methods with NumPy/CuPy functions

CuPy also implements many functions that are compatible to those of NumPy. We can write unified forward/backward methods with them. Consider that we want to write a backprop-able function $f(x, y) = \exp(x) + \exp(y)$. We name it `ExpAdd` here. It can be written straight-forward as follows:

```python
class ExpAdd(Function):
    def forward_cpu(self, inputs):
        x, y = inputs
        z = np.exp(x) + np.exp(y)
        return z,

    def backward_cpu(self, inputs, grad_outputs):
        x, y = inputs
        gz, = grad_outputs
        gx = gz * np.exp(x)
        gy = gz * np.exp(y)
        return gx, gy

    def forward_gpu(self, inputs):
        cupy = cuda.cupy
        x, y = inputs
        z = cupy.exp(x) + cupy.exp(y)
        return z,

    def backward_gpu(self, inputs, grad_outputs):
        cupy = cuda.cupy
        x, y = inputs
        gz, = grad_outputs
        gx = gz * cupy.exp(x)
        gy = gz * cupy.exp(y)
```
```
return gx, gy

def expadd(x, y):
    return ExpAdd()(x, y)
```

**Note:** Here we used `cuda.cupy` instead of directly accessing `cupy`. This is because the `cupy` module cannot be imported if the CUDA is not installed. In order to keep the implementation valid in non-CUDA environment, we have to defer the access to the `cupy` module. Note that the `chainer.cuda` module can be imported even if the CUDA is not installed. Of course, the module in such environment is almost useless, but if the interpreter does not run through the code accessing CUDA-dedicated functions, the code is still valid.

The CPU and GPU implementations are almost same, except that `numpy` is replaced by `cupy` in GPU methods. We can unify these functions using the `cuda.get_array_module()` function. This function accepts arbitrary number of arrays, and returns an appropriate module for them. See the following code

```
class ExpAdd(Function):
    def forward(self, inputs):
        xp = cuda.get_array_module(*inputs)
        x, y = inputs
        z = xp.exp(x) + xp.exp(y)
        return z,

    def backward(self, inputs, grad_outputs):
        xp = cuda.get_array_module(*inputs)
        x, y = inputs
        gz, = grad_outputs
        gx = gz * xp.exp(x)
        gy = gz * xp.exp(y)
        return gx, gy

def expadd(x, y):
    return ExpAdd()(x, y)
```

Note that this code works correctly even if CUDA is not installed in the environment. If CUDA is not found, `get_array_module` function always returns `numpy`. We often use the name `xp` for the variadic module name, which is analogous to the abbreviation `np` for NumPy and `cp` for CuPy.

### Write an Elementwise Kernel Function

Let’s turn back to the MulAdd example.

The GPU implementation of MulAdd as shown above is already fast and parallelized on GPU cores. However, it invokes two kernels during each of forward and backward computations. It might hurt performance, since the intermediate temporary arrays are read and written by possibly different GPU cores, which consumes much bandwidth. We can reduce the number of invocations by defining our own kernel. It also reduce the memory consumption.

Most functions only require elementwise operations like MulAdd. CuPy provides a useful tool to define elementwise kernels, the `cupy.elementwise.ElementwiseKernel` class, and Chainer wraps it by `cuda.elementwise()` function. Our MulAdd implementation can be improved as follows:

```
class MulAdd(Function):
    def forward_cpu(self, inputs):
        ...
```
def backward_cpu(self, inputs, grad_outputs):
    ...

def forward_gpu(self, inputs):
    cupy = cuda.cupy
    x, y, z = inputs
    w = cuda.elementwise(
        'float32 x, float32 y, float32 z',
        'float32 w',
        'w = x * y + z',
        'muladd_fwd')(x, y, z)
    return w,

def backward_gpu(self, inputs, grad_outputs):
    x, y, z = inputs
    gw, = grad_outputs
    gx, gy = cuda.elementwise(
        'float32 x, float32 y, float32 gw',
        'float32 gx, float32 gy',
        '''
        gx = y * gw;
        gy = x * gw;
        ''',
        'muladd_bwd')(x, y, gw)
    gz = gw
    return gx, gy, gz

cuda.elementwise() function accepts the essential implementation of the kernel function, and returns a kernel invocation function (actually, it returns ElementwiseKernel object, which is callable). In typical usage, we pass four arguments to this function as follows:

1. Input argument list. This is a comma-separated string each entry of which consists of a type specification and an argument name.
2. Output argument list in the same format as the input argument list.
3. Body of parallel loop. We can use the input/output argument names as an element of these arrays.
4. Name of the kernel function, which is shown in debuggers and profilers.

Above code is not compiled on every forward/backward computation thanks to two caching mechanisms provided by cuda.elementwise().

The first one is binary caching: cuda.elementwise() function caches the compiled binary in the $(HOME)/.cupy/kernel_cache directory with a hash value of the CUDA code, and reuses it if the given code matches the hash value. This caching mechanism is actually implemented in CuPy.

The second one is upload caching: Given a compiled binary code, we have to upload it to the current GPU in order to execute it. cuda.elementwise() function memoizes the arguments and the current device, and if it is called with the same arguments for the same device, it reuses the previously uploaded kernel code.

The above MulAdd code only works for float32 arrays. The ElementwiseKernel also supports the type-variadic kernel definition. In order to define variadic kernel functions, you can use type placeholder by placing a single character as type specifier:

class MulAdd(Function):
    def forward_cpu(self, inputs):

2.4. Define your own function
...  

```python
def backward_cpu(self, inputs, grad_outputs):
    ...  

def forward_gpu(self, inputs):
    cupy = cuda.cupy
    x, y, z = inputs
    w = cupy.elementwise(
        'T x, T y, T z',
        'T w',
        'w = x * y + z',
        'muladd_fwd')(x, y, z)
    return w,

def backward_gpu(self, inputs, grad_outputs):
    x, y, z = inputs
    gw, = grad_outputs
    gx, gy = cupy.elementwise(
        'T x, T y, T gw',
        'T gx, T gy',
        'gx = y * gw;
        gy = x * gw;
        ',
        'muladd_bwd')(x, y, gw)
    gz = gw
    return gx, gy, gz
```

The type placeholder \( T \) indicates an arbitrary data type that CuPy supports.

There are more functionalities on user-defined kernels in CuPy. See the CuPy documentation on user-defined kernels for more details.

**Write a function with training/test mode**

We sometimes want to make a function behave differently in training and test modes. The training/test mode in Chainer is configured by `chainer.config`. This is a thread-local configuration object, and users can substitute True or False to its `train` attribute. You can refer to [Configuring Chainer](#) to see how to configure this flag as well as other configuration items.

Here, we just show how to use this flag to make a function support training/test mode. You will need to check the value of the boolean flag `chainer.config.train` and branch appropriately.

For example, consider the following simple dropout function:

```python
def dropout(x):
    xp = cuda.get_array_module(x.data)
    mask = 2 * (xp.random.rand(*x.shape) > 0.5).astype(x.dtype)
    return x * mask
```

This function applies dropout to each element and doubles survived elements to preserve the scale. The above implementation applies dropout even in test mode, but it is not a desired behavior. We can fix it as follows:
def dropout(x):
    if not chainer.config.train:
        return x

    xp = cuda.get_array_module(x.data)
    mask = 2 * (xp.random.rand(*x.shape) > 0.5).astype(x.dtype)
    return x * mask

The function now supports test mode. Note that you usually do not have to implement your own dropout function because dropout() is officially provided.

Links that wrap functions

Some functions are meant to be combined with parameters. In such case, it is useful to write a small link that wraps the function. We have already seen how to define a chain that wraps other links (by inheriting Chain class). Here we study how to define a link that does not hold any other links.

As the first example, suppose that we want to implement elementwise product function between the input array and the parameter array. It can be defined as follows:

class EltwiseParamProduct(Chain):
    def __init__(self, shape):
        super(EltwiseParamProduct, self).__init__()
        with self.init_scope():
            self.W = chainer.Parameter(initializers.Normal(scale=1.), shape)

    def __call__(self, x):
        return self.W * x

For another example, assume we want to define a simple linear layer. It is already defined as Linear, so this is an educational example. The linear layer is divided into two parts: a function and its wrapper link. First, we have to define a function on variables:

class LinearFunction(Function):
    def forward(self, inputs):
        x, W, b = inputs
        return x.dot(W.T) + b,

    def backward(self, inputs, grad_outputs):
        x, W, b = inputs
        gy, = grad_outputs
        gx = gy.dot(W)
        gW = gy.T.dot(x)
        gb = gy.sum(axis=0)
        return gx, gW, gb

def linear(x, W, b):
    return LinearFunction()(x, W, b)

This function takes three arguments: input, weight, and bias. It can be used as a part of model definition, though is inconvenient since the user have to manage the weight and bias parameters directly. In order to make a convenient module, let’s wrap it into a link:

class Linear(Chain):
    def __init__(self, in_size, out_size):
        super(Linear, self).__init__()
        with self.init_scope():
            self.W = chainer.Parameter(initializers.Normal(scale=1.), (in_size, out_size))
            self.b = chainer.Parameter(initializers.Zeros(), out_size)


    def __call__(self, x):
        x, W, b = x, self.W, self.b
        return x.dot(W.T) + b

2.4. Define your own function
super(Linear, self).__init__()
with self.init_scope():
    self.W = chainer.Parameter(
        initializers.Normal(1. / math.sqrt(in_size)),
        (out_size, in_size))
    self.b = chainer.Parameter(0, (out_size,))

def __call__(self, x):
    return linear(x, self.W, self.b)

This link hides the parameters of the linear layer.

**Note:** An advanced tip to implement functions: if you want to preserve some information between forward and backward computations (e.g. to cache some arrays), you can store it as attributes. Be careful that it might increase the memory consumption during the whole forward-backward computation. If you want to train very large networks on a GPU with limited memory, it is not recommended to cache arrays between forward and backward. There is one exception for this: caching the output arrays does not change the memory consumption, because they are also held by the output Variable objects.

**Warning:** You should not assume a one-to-one match of calls of forward and backward. Some users may call backward more than once after one forward call.

**Testing Function**

In order to isolate the cause of learning failure from implementation bugs, it is important to test function implementations. Chainer provides simple utilities to help writing unit tests. They are defined in the `gradient_check` module.

The most important test utility is the `numerical_grad()` function. This function computes the numerical gradient of given function using finite differences. It can be used as follows

```python
x = np.random.randn(4, 3).astype(np.float32)
gy = np.ones((4, 3), dtype=np.float32)
f = lambda: (x * x,)
gx = gradient_check.numerical_grad(f, (x,), (gy,))
```

f is a closure that returns a tuple of array(s) computed from input arrays. The second and third arguments of `numerical_grad()` are tuples of input arrays and output gradient arrays, respectively. The code above computes the numerical gradients of \( \text{sum}(f(x)) \), where \( \text{sum} \) indicates the summation over all elements. The summation can be weighted by changing `gy`. `numerical_grad()` function also accepts additional `eps` argument, which indicates the quantization width of finite differences.

**Note:** `numerical_grad()` function accepts both CPU and GPU arrays. Note that we cannot mix CPU and GPU arrays.

Another utility is `chainer.testing.assert_allclose()` function. This is similar to `numpy.testing.assert_allclose()` function. The difference is that Chainer’s version accepts CPU and GPU arrays as inputs. We can mix them in one invocation of `chainer.testing.assert_allclose()`. The default values of optional arguments are also different.

Here is a typical usage of gradient checking utilities. This is a test example of `functions.relu()` function.
The first four lines of the test code are simple forward and backward computation of ReLU function. The next two lines compute numerical gradient using the same forward function without backward routine. And at last, we compare these two results elementwise. Note that the above test code can be easily modified to test GPU version just by replacing CPU arrays to GPU arrays.

In most cases, we do not write the code like the above explicitly because Chainer offers a utility function `chainer.gradient_check.check_backward()` that follows this procedure.

You can find many examples of function tests under `tests/chainer_tests/function_tests` directory.

### Type check

In this section, you will learn about the following things:

- Basic usage of type check
- Detail of type information
- Internal mechanism of type check
- More complicated cases
- Call functions
- Typical type check example
After reading this section, you will be able to:

- Write a code to check types of input arguments of your own functions

**Basic usage of type check**

When you call a function with an invalid type of array, you sometimes receive no error, but get an unexpected result by broadcasting. When you use CUDA with an illegal type of array, it causes memory corruption, and you get a serious error. These bugs are hard to fix. Chainer can check preconditions of each function, and helps to prevent such problems. These conditions may help a user to understand specification of functions.

Each implementation of Function has a method for type check, check_type_forward(). This function is called just before the forward() method of the Function class. You can override this method to check the condition on types and shapes of arguments.

check_type_forward() gets an argument in_types:

```python
def check_type_forward(self, in_types):
    ...
```

in_types is an instance of TypeInfoTuple, which is a sub-class of tuple. To get type information about the first argument, use in_types[0]. If the function gets multiple arguments, we recommend to use new variables for readability:

```python
x_type, y_type = in_types
```

In this case, x_type represents the type of the first argument, and y_type represents the second one.

We describe usage of in_types with an example. When you want to check if the number of dimension of x_type equals to 2, write this code:

```python
utils.type_check.expect(x_type.ndim == 2)
```

When this condition is true, nothing happens. Otherwise this code throws an exception, and the user gets a message like this:

```
Traceback (most recent call last):
...
InvalidType: Expect: in_types[0].ndim == 2
Actual: 3 != 2
```

This error message means that “ndim of the first argument expected to be 2, but actually it is 3”.

**Detail of type information**

You can access three information of x_type.

- .shape is a tuple of ints. Each value is size of each dimension.
- .ndim is int value representing the number of dimensions. Note that ndim == len(shape)
- .dtype is numpy.dtype representing data type of the value.

You can check all members. For example, the size of the first dimension must be positive, you can write like this:

```python
utils.type_check.expect(x_type.shape[0] > 0)
```

You can also check data types with .dtype:
And an error is like this:

```
Traceback (most recent call last):
...  
InvalidType: Expect: in_types[0].dtype == <type 'numpy.float64'>
Actual: float32 != <type 'numpy.float64'>
```

You can also check kind of dtype. This code checks if the type is floating point

```
utils.type_check.expect(x_type.dtype.kind == 'f')
```

You can compare between variables. For example, the following code checks if the first argument and the second argument have the same length:

```
utils.type_check.expect(x_type.shape[1] == y_type.shape[1])
```

### Internal mechanism of type check

How does it show an error message like "in_types[0].ndim == 2"? If x_type is an object containing ndim member variable, we cannot show such an error message because this equation is evaluated as a boolean value by Python interpreter.

Actually x_type is a Expr objects, and doesn’t have a ndim member variable itself. Expr represents a syntax tree. x_type.ndim makes a Expr object representing (getattr, x_type, 'ndim').x_type.ndim == 2 makes an object like (eq, (getattr, x_type, 'ndim'), 2).type_check.expect() gets a Expr object and evaluates it. When it is True, it causes no error and shows nothing. Otherwise, this method shows a readable error message.

If you want to evaluate a Expr object, call eval() method:

```
actual_type = x_type.eval()
```

actual_type is an instance of TypeInfo, while x_type is an instance of Expr. In the same way, x_type.shape[0].eval() returns an int value.

### More powerful methods

Expr class is more powerful. It supports all mathematical operators such as + and *. You can write a condition that the first dimension of x_type is the first dimension of y_type times four:

```
utils.type_check.expect(x_type.shape[0] == y_type.shape[0] * 4)
```

When x_type.shape[0] == 3 and y_type.shape[0] == 1, users can get the error message below:

```
Traceback (most recent call last):
...
InvalidType: Expect: in_types[0].shape[0] == in_types[1].shape[0] * 4
Actual: 3 != 4
```

To compare a member variable of your function, wrap a value with Variable to show readable error message:
This code can check the equivalent condition below:

\[
x_{\text{type}.\text{shape}[0]} == \text{utils.type_check.Variable}(\text{self.in\_size}, \text{"in\_size"})
\]

However, the latter condition doesn’t know the meaning of this value. When this condition is not satisfied, the latter code shows unreadable error message:

```
InvalidType: Expect: \text{in\_types}[0].\text{shape}[0] == 4 \ # \text{what does ‘4’ mean?}
Actual: \ 3 \ != \ 4
```

Note that the second argument of `utils.type_check.Variable` is only for readability.

The former shows this message:

```
InvalidType: Expect: \text{in\_types}[0].\text{shape}[0] == \text{in\_size} \ # \text{OK, ‘in\_size’ is a value that is given to the constructor}
Actual: \ 3 \ != \ 4 \ # \text{You can also check actual value here}
```

**Call functions**

How to check summation of all values of shape? `Expr` also supports function call:

```python
sum = \text{utils.type_check.Variable}(\text{np.sum}, \text{‘sum’})
\text{utils.type_check.expect}(\text{sum}(x_{\text{type}.\text{shape}}) == 10)
```

Why do we need to wrap the function `numpy.sum` with `utils.type_check.Variable`? `x_{\text{type}.\text{shape}}` is not a tuple but an object of `Expr` as we have seen before. Therefore, `numpy.sum(x_{\text{type}.\text{shape}})` fails. We need to evaluate this function lazily.

The above example produces an error message like this:

```
Traceback (most recent call last):
...  
InvalidType: Expect: \text{sum(in\_types}[0].\text{shape}) == 10
Actual: \ 7 \ != \ 10
```

**More complicated cases**

How to write a more complicated condition that can’t be written with these operators? You can evaluate `Expr` and get its result value with `eval()` method. Then check the condition and show warning message by hand:

```python
x_{\text{shape}} = x_{\text{type}.\text{shape}.\text{eval}}() \ # \text{get actual shape (int tuple)}
\text{if not more\_complicated\_condition}(x_{\text{shape}}):
    \text{expect\_msg} = \text{‘Shape is expected to be …’}
    \text{actual\_msg} = \text{‘Shape is …’}
    \text{raise \text{utils.type_check.InvalidType}(expect\_msg, actual\_msg)}
```

Please write a readable error message. This code generates the following error message:

```
Traceback (most recent call last):
...  
InvalidType: Expect: Shape is expected to be ...
Actual: Shape is ...
```
Typical type check example

We show a typical type check for a function.

First check the number of arguments:

```python
utils.type_check.expect(in_types.size() == 2)
```

`in_types.size()` returns a `Expr` object representing the number of arguments. You can check it in the same way.

And then, get each type:

```python
x_type, y_type = in_types
```

Don’t get each value before checking `in_types.size()`. When the number of argument is illegal, `type_check.expect` might output unuseful error messages. For example, this code doesn’t work when the size of `in_types` is 0:

```python
utils.type_check.expect(
    in_types.size() == 2,
    in_types[0].ndim == 3,
)
```

After that, check each type:

```python
utils.type_check.expect(
    x_type.dtype == np.float32,
    x_type.ndim == 3,
    x_type.shape[1] == 2,
)
```

The above example works correctly even when `x_type.ndim == 0` as all conditions are evaluated lazily.
CHAPTER 3

Chainer Reference Manual

Core functionalities

Variable and Parameter

<table>
<thead>
<tr>
<th>chainer.Variable</th>
<th>Array with a structure to keep track of computation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.Parameter</td>
<td>Parameter variable that can be registered to a link.</td>
</tr>
<tr>
<td>chainer.variable.VariableNode</td>
<td>Node in the backward computational graph representing a variable.</td>
</tr>
</tbody>
</table>

**chainer.Variable**

class chainer.Variable(data=None, *, name=None, grad=None, requires_grad=True)

Array with a structure to keep track of computation.

Every variable holds a data array of type either `numpy.ndarray` or `cupy.ndarray`.

A variable object holds a data array and a `VariableNode` object of a computational graph. If the variable is constructed by the user, the node is `_root_` and does not hold any parent. If the variable is constructed by a `Function` object, the node holds a reference to its parent called `creator`. This reference is used in backpropagation to backtrack the graph.

Users can disable (resp. enable) this chaining behavior by calling `no_backprop_mode()` (resp. `force_backprop_mode()`). In the former context, a variable never creates a computational graph, whereas in the latter context, it is forced to create.

**Warning:** `volatile` argument is not supported anymore since v2. Instead, use `chainer.no_backprop_mode()`.

Parameters
Variables

- **data** (*numpy.ndarray or cupy.ndarray*) – Initial data array.
- **name** (*str*) – Name of the variable.
- **grad** (*numpy.ndarray or cupy.ndarray*) – Initial gradient array.
- **requires_grad** (*bool*) – Boolean indicating whether grad will be set in backward calculation.

Methods

**_getitem_(x, slices)**

Extract elements from array with specified shape, axes and offsets.

**Parameters**

- **x** (*Variable*) – A variable to be sliced.
- **slices** (*int, slice, Ellipsis, None, integer array-like, boolean array-like or tuple of them*) – It is an integer, a slice, an ellipsis, a numpy.newaxis, an integer array-like, a boolean array-like or tuple of them.

**Returns**

*Variable object* which contains sliced array of x.

**Return type** *Variable*  

**Note:** It only supports types that are supported by CUDA’s atomicAdd when an integer array is included in slices. The supported types are *numpy.float32, numpy.int32, numpy.uint32, numpy.uint64* and *numpy.ulonglong*.

**Note:** It does not support slices that contains multiple boolean arrays.

**Note:** See NumPy document for details of indexing.

**_len_()**

Returns the first dimension of the data array.

**Returns** Number of the first dimension of the data array.

**Return type** *int*

**_copy_()**

**addgrad**(var)

Accumulates the gradient array from given source variable.
This method adds the gradient of a given variable to the gradient of this variable. The accumulation is even done across the host and different devices. If this variable has uninitialized data/grad arrays, this method initializes it with the shape of the given variable and then accumulates the gradient.

**Parameters**

**var** *(Variable)* – Source variable.

**backward** *(retain_grad=False)*

Runs error backpropagation (a.k.a. backprop) from this variable.

On backprop, *Function.backward()* is called on each *Function* object appearing in the backward graph starting from this variable. The backward graph is represented by backward references from variable nodes to their creators, and from functions to their input variable nodes. The backprop stops at all root nodes. Some functions set *None* as gradients of some inputs, where further backprop does not take place at such inputs.

This method uses *grad* as the initial error array. User can manually set a gradient array before calling this method. If *data* contains only one element (i.e., it is scalar) and *grad* is *None*, then this method automatically complements 1.0 as the initial error. This is useful on starting backprop from some scalar loss value.

**Parameters**

**retain_grad** *(bool)* – If *True*, the gradient arrays of all intermediate variables are kept. Otherwise, *grad* of the intermediate variables are set to *None* on appropriate timing, which may reduce the maximum memory consumption.

In most cases of training some models, the purpose of backprop is to compute gradients of parameters, not of all variables, and therefore it is recommended to set this flag *False*.

**cleargrad()**

Clears the gradient array.

**copydata** *(var)*

Copies the data array from given source variable.

This method copies the data array from given variable to this variable. The copy is done even if the arrays reside on different devices, including across the host and a GPU device. If this variable has an uninitialized data array, this method initializes it by the data array of the given variable. Similarly, if the given variable has an uninitialized data array, this method initializes it by the data array of this variable (*self*). If both are uninitialized, this method does nothing.

**Parameters**

**var** *(Variable)* – Source variable.

**debug_print()**

Display a summary of the stored data and location of the Variable

**reshape** *(shape)*

Returns a variable of a different shape and the same content.

See also:

*chainer.functions.reshape()* for full documentation,

**retain_data()**

Lets the corresponding variable node keep the underlying array.

**set_creator** *(gen_func)*

Notifies the variable that the given function is its creator.

**Parameters**

**gen_func** *(Function)* – Function object that creates this variable as one of its outputs.

**summary()**
**to_cpu()**
Copies the data and gradient arrays to CPU.

**to_gpu**(device=None)
Copies the data and gradient arrays to specified GPU.

**Parameters**
*device* — Target device specifier. If omitted, the current device is used.

**transpose**(*axes*)
Permute the dimensions of an input variable without copy.

**See also:**
*chainer.functions.transpose() for full documentation.*

**unchain()**
Deletes the reference to the creator of this variable.

This method deletes the reference to the creator from the corresponding variable node. Unlike
*unchain_backward()*, it does not backtrack the graph.

This method is equivalent to *self.creator = None*.

**unchain_backward()**
Deletes references between variable nodes and functions backward.

After this method completes, intermediate variable nodes and functions that are not referenced from any-
where are deallocated by reference count GC. Also this variable itself deletes the reference to its creator
function from the node, i.e. the node becomes root in the computation graph. It indicates that backprop
after unchaining stops at this variable. This behavior is useful to implement truncated BPTT.

**zerograd()**
Initializes the gradient array by zeros.

Depreciated since version v1.15: Use *cleargrad()* instead.

**Attributes**

**creator**
*Function()* object that created this variable.

This property has a setter to which *None* can be set. Setting *None* to this property is equivalent to call
*unchain()*: it purges the variable from the function that created this variable.

The setter also accepts the original *Function()* object that created this variable. For example, you can
once set *None* to this property and then set the original value again.
Note: Setting an irrelevant `Function()` object does not emit any error immediately, whereas the behavior is undefined. Do not set a `Function()` object that did not create this variable object.

**data**

**dtype**

**grad**

**label**
Short text that represents the variable.

**name**

**ndim**

**node**

**rank**

**requires_grad**
It indicates that `grad` will be set in backward calculation.

**shape**

**size**

---

**chainer.Parameter**

```python
class chainer.Parameter(initializer=None, shape=None, name=None)
```

Parameter variable that can be registered to a link.

Parameter is a subclass of `Variable`. It almost behaves as same as a usual variable except that a parameter can be registered to a `Link` object just by assigning it to an attribute of the link within an `init_scope()` context.

Parameter also supports an initialization by an initializer. It can have two initializers: one for the data array, and the other for the gradient array. The initializer only specifies the way of filling the elements of these arrays, and the shape information is specified at the initialization point.

When a link that the parameter has been registered to is passed to a `GradientMethod`, an update rule is set to the parameter. This update rule specifies how to update the data array of the parameter using its gradient array.

**Parameters**

- **initializer** (`Initializer or numpy.ndarray or cupy.ndarray`) – Initializer of the data array. If `shape` is given, this initializer is immediately used to initialize the data array. Otherwise, if it is an array, it is immediately used as the data array, and otherwise the data array is left uninitialized and will be initialized by this initializer in `initialize()`. It can also be a scalar, in which case the data array will be filled by this scalar. Note that float32 is used in this case.

- **shape** (`int or tuple of int or None`) – Shape of the parameter. If it is `None`, the initialization is deferred to the call of `initialize()`.

- **name** (`str`) – Name of the parameter.

**Variables**

3.1. Core functionalities
• **initializer** – Initializer of the data array. It is used for initializing the data array of an uninitialized variable.

• **update_rule** – UpdateRule instance that updates this variable as a parameter. This argument is set to `update_rule`.

**Methods**

```python
__getitem__(x, slices)
```

Extract elements from array with specified shape, axes and offsets.

**Parameters**

• **x** (*Variable*) – A variable to be sliced.

• **slices** (*int, slice, Ellipsis, None, integer array-like, boolean array-like or tuple of them*) – It is an integer, a slice, an ellipsis, a numpy.newaxis, an integer array-like, a boolean array-like or tuple of them.

**Returns**

*Variable* object which contains sliced array of *x*.

**Return type** *Variable*

---

**Note:** It only supports types that are supported by CUDA’s atomicAdd when an integer array is included in `slices`. The supported types are `numpy.float32`, `numpy.int32`, `numpy.uint32`, `numpy.uint64` and `numpy.ulonglong`.

---

**Note:** It does not support `slices` that contains multiple boolean arrays.

---

**Note:** See NumPy document for details of indexing.

```python
__len__()
```

Returns the first dimension of the data array.

**Returns** Number of the first dimension of the data array.

**Return type** *int*

```python
__copy__()
```

Accumulates the gradient array from given source variable.

This method adds the gradient of a given variable to the gradient of this variable. The accumulation is even done across the host and different devices. If this variable has uninitialized data/grad arrays, this method initializes it with the shape of the given variable and then accumulates the gradient.

**Parameters** *var* (*Variable*) – Source variable.

```python
backward(retain_grad=False)
```

Runs error backpropagation (a.k.a. backprop) from this variable.

On backprop, `Function.backward()` is called on each `Function` object appearing in the backward graph starting from this variable. The backward graph is represented by backward references from variable
nodes to their creators, and from functions to their input variable nodes. The backprop stops at all root nodes. Some functions set `None` as gradients of some inputs, where further backprop does not take place at such inputs.

This method uses `grad` as the initial error array. User can manually set a gradient array before calling this method. If `data` contains only one element (i.e., it is scalar) and `grad` is `None`, then this method automatically complements 1.0 as the initial error. This is useful on starting backprop from some scalar loss value.

**Parameters**

- `retain_grad (bool)` — If `True`, the gradient arrays of all intermediate variables are kept. Otherwise, `grad` of the intermediate variables are set to `None` on appropriate timing, which may reduce the maximum memory consumption.

In most cases of training some models, the purpose of backprop is to compute gradients of parameters, not of all variables, and therefore it is recommended to set this flag `False`.

`cleargrad()`

`copydata (var)`

Copies the data array from given source variable.

This method copies the data array from given variable to this variable. The copy is done even if the arrays reside on different devices, including across the host and a GPU device. If this variable has an uninitialized data array, this method initializes it by the data array of the given variable. Similarly, if the given variable has an uninitialized data array, this method initializes it by the data array of this variable (`self`). If both are uninitialized, this method does nothing.

**Parameters**

- `var (Variable)` — Source variable.

`debug_print ()`

Display a summary of the stored data and location of the Variable.

`initialize (shape)`

Initializes the uninitialized variable.

Uninitialized variable is a variable created with the data array set to `None`. This method creates and initializes the data array. The shape of the variable can be left unknown until this method is called.

**Parameters**

- `shape (tuple of int)` — Shape of the data array.

`reshape (*shape)`

Returns a variable of a different shape and the same content.

See also:

- `chainer.functions.reshape()` for full documentation,

`retain_data ()`

Lets the corresponding variable node keep the underlying array.

`set_creator (gen_func)`

Notifies the variable that the given function is its creator.

**Parameters**

- `gen_func (Function)` — Function object that creates this variable as one of its outputs.

`summary ()`

`to_cpu ()`

`to_gpu (device=None)`

`transpose (*axes)`

Permute the dimensions of an input variable without copy.
See also:

\texttt{chainer.functions.transpose()} for full documentation.

\texttt{unchain()}
Deletes the reference to the creator of this variable.

This method deletes the reference to the creator from the corresponding variable node. Unlike \texttt{unchain_backward()}, it does not backtrack the graph.

This method is equivalent to \texttt{self.creator = None}.

\texttt{unchain_backward()}
Deletes references between variable nodes and functions backward.

After this method completes, intermediate variable nodes and functions that are not referenced from anywhere are deallocated by reference count GC. Also this variable itself deletes the reference to its creator function from the node, i.e. the node becomes root in the computation graph. It indicates that backprop after unchaining stops at this variable. This behavior is useful to implement truncated BPTT.

\texttt{update()}
Updates the data array using the gradient and the update rule.

This method updates the parameter using the attached update rule.

\texttt{zerograd()}

\texttt{\_eq\_}(\texttt{other})

\texttt{\_ne\_}(\texttt{other})

\texttt{\_lt\_}(\texttt{other})

\texttt{\_le\_}(\texttt{other})

\texttt{\_gt\_}(\texttt{other})

\texttt{\_ge\_}(\texttt{other})

\texttt{\_nonzero\_}()

\texttt{\_bool\_}()

Attributes

\texttt{creator}
\texttt{Function()} object that created this variable.

This property has a setter to which \texttt{None} can be set. Setting \texttt{None} to this property is equivalent to call \texttt{unchain()}; it purges the variable from the function that created this variable.

The setter also accepts the original \texttt{Function()} object that created this variable. For example, you can once set \texttt{None} to this property and then set the original value again.

\textbf{Note:} Setting an irrelevant \texttt{Function()} object does not emit any error immediately, whereas the behavior is undefined. Do not set a \texttt{Function()} object that did not create this variable object.

data
dtype
grad
initializer = None

label
Short text that represents the variable.

name

ndim

node

rank

requires_grad
It indicates that grad will be set in backward calculation.

shape

size

chainer.variable.VariableNode
class chainer.variable.VariableNode(variable, name, grad=None)
Node in the backward computational graph representing a variable.

This object represents a variable node in a computational graph. The node is used in error backpropagation (a.k.a. backprop) to determine which gradient to be passed to each function.

A variable node is held by the corresponding Variable object, which is managed by users. Function objects that take the variable as an input also hold references to the variable node.

Note that the node does not hold a reference to the corresponding data array in general. The data array is actually accessible by the node in the following cases.

1. If there exists a Variable object that holds a reference to the variable node, the variable node holds a weak reference to the variable object, and thus the data array is accessible via the weak reference.

2. If retain_data() is called, the node holds a reference to the data array. It is mainly called by a function that needs the input or output data array in its backprop procedure. See Function.retain_inputs() and Function.retain_outputs() for more details.

Users usually do not need to touch this variable node object. The computational graph is automatically managed by Chainer, and any interface that is beneficial for users is also provided by Variable.

Parameters

• variable (Variable) – The corresponding variable object.

• name (str) – Name of the variable node.

Variables

• dtype – Data type of the data array.

• shape – Shape of the data array.

• name (str) – Name of the variable node.

Methods

retain_data()
Lets the node hold a reference to the underlying data array.
This method gets the data array of the corresponding variable and keeps it. If the weak reference to the corresponding variable is dead, it raises an error.

**set_creator** (*creator*)
Sets a `Function` object that created this node.

This method is equivalent to `self.creator = creator`.

**Parameters**

*creator* (`Function`) – Function object that created this node.

**unchain** ()
Deletes the reference to the creator of this variable node.

This method is equivalent to `self.creator = None`.

**Attributes**

*creator*
Function node that created this variable node.

*data*
Data array of the corresponding variable.

If the data is not available, it returns `None`.

*grad*
Gradient array of the corresponding variable.

*label*
Short text that represents the variable node.

*rank*

*requires_grad*
It indicates that `grad` will be set in backward calculation.

**Function**

<table>
<thead>
<tr>
<th>chainer.Function</th>
<th>Function on variables with backpropagation ability.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.force_backprop_mode</td>
<td>Make a context manager which enables back-propagation.</td>
</tr>
<tr>
<td>chainer.no_backprop_mode</td>
<td>Make a context manager which disables back-propagation.</td>
</tr>
</tbody>
</table>

**chainer.Function**

**class chainer.Function**
Function on variables with backpropagation ability.

All function implementations defined in `chainer.functions` inherit this class.

The main feature of this class is keeping track of function applications as a backward graph. When a function is applied to `Variable` objects, its `forward()` method is called on `data` fields of input variables, and at the same time it chains references from output variable nodes to the function and from the function to its input nodes.

**Note:** As of v2.0, the input/output variables and their corresponding variable nodes in the graph are distinguished. Function acts like a function on `Variable` objects that returns `Variable` objects as outputs,
whereas these objects do not appear directly in the graph. Instead, their corresponding `VariableNode` objects are inserted to the graph.

**Note:** As of v1.5, a function instance cannot be used twice in any computational graphs. In order to reuse a function object multiple times, use `copy.copy()` before the function applications to make a copy of the instance.

This restriction also means that we cannot make a *stateful function* anymore. For example, it is now not allowed to let a function hold parameters. Define a function as a pure (stateless) procedure, and use `Link` to combine it with parameter variables.

### Example

Let \( x \) an instance of `Variable` and \( f \) an instance of `Function` taking only one argument. Then a line

```python
>>> import numpy, chainer, chainer.functions as F
>>> x = chainer.Variable(numpy.zeros(10))
>>> f = F.Identity()
>>> y = f(x)
```

computes a new variable \( y \) and creates backward references. Actually, backward references are set as per the following diagram:

\[
x\text{.node} \leftarrow f \leftarrow y\text{.node}
\]

If an application of another function \( g \) occurs as

```python
>>> g = F.Identity()
>>> z = g(x)
```

then the graph grows with a branch:

\[
\begin{array}{c}
|--- f \leftarrow y\text{.node} \\
\text{x.node} \leftarrow & \quad |--- g \leftarrow z\text{.node}
\end{array}
\]

Note that the branching is correctly managed on backward computation, i.e. the gradients from \( f \) and \( g \) are accumulated to the gradient of \( x \).

Every function implementation should provide `forward_cpu()`, `forward_gpu()`, `backward_cpu()` and `backward_gpu()`. Alternatively, one can provide `forward()` and `backward()` instead of separate methods. Backward methods have default implementations that just return `None`, which indicates that the function is non-differentiable.

For functions that do not need a part of inputs in backward computation, there is a way to possibly reduce the memory consumption by quickly releasing the input arrays after the forward propagation. This is done by calling `retain_inputs()` from inside of `forward()` (including `forward_cpu()` and `forward_gpu()`). See the documentation of `retain_inputs()` for details.

For functions that need a part of outputs in backward computation, it is strongly recommended to call `retain_outputs()` from inside of `forward()` (including `forward_cpu()` and `forward_gpu()`). It marks the specified output variable nodes to retain the data. The retained data can be accessed by `output_arrays` property.

### Variables

3.1. Core functionalities 57
• **inputs** – A tuple or list of input variables.
• **outputs** – A tuple or list of output variables.
• **output_data** – A tuple of retained output arrays. It has the same length as **outputs**.
  The data of variables that are not retained are set to **None**. See **retain_outputs()** for details.

### Methods

**__call__** (*inputs*)
Applies forward propagation with chaining backward references.
Basic behavior is expressed in documentation of **Function** class.

**Note:** If the **data** attribute of input variables exist on GPU device, then, before it calls **forward()**
method, the appropriate device is selected, so in most cases implementers do not need to take care of
device selection.

**Parameters**
• **inputs** – Tuple of input **Variable**, **numpy.ndarray** or **cupy.ndarray** objects. If the input is an **numpy.ndarray** or a **cupy.ndarray**, it is automatically
  wrapped with **Variable**.

**Returns** One **Variable** object or a tuple of multiple **Variable** objects.

**add_hook** (**hook**, **name**=**None**)
Registers the function hook.

**Parameters**
• **hook** (**FunctionHook**) – Function hook to be registered.
• **name** (**str**) – Name of the function hook. name must be unique among function hooks
  registered to the function. If **None**, default name of the function hook is used.

**backward** (**inputs**, **grad_outputs**)
Applies backprop to output gradient arrays.

It delegates the procedure to **backward_cpu()** or **backward_gpu()** by default. Which it selects is
determined by the type of input arrays and output gradient arrays. Implementations of **Function** must
implement either CPU/GPU methods or this method, if the function is intended to be backprop-ed.

**Parameters**
• **inputs** – Tuple of input arrays.
• **grad_outputs** – Tuple of output gradient arrays.

**Returns** Tuple of input gradient arrays. Some or all of them can be **None**, if the function is not
differentiable on inputs.

**Return type** **tuple**

**Warning:** Implementations of **Function** must take care that the return value must be a tuple even
if it returns only one array.
backward_cpu \((inputs, grad_outputs)\)
Applies backprop to output gradient arrays on CPU.

Parameters

- **inputs** – Tuple of input \texttt{numpy.ndarray} object(s).
- **grad_outputs** – Tuple of output gradient \texttt{numpy.ndarray} object(s).

Returns Tuple of input gradient \texttt{numpy.ndarray} object(s). Some or all of them can be None, if the function is not differentiable on corresponding inputs.

Return type \texttt{tuple}

**Warning:** Implementations of \textit{Function} must take care that the return value must be a tuple even if it returns only one array.

backward_gpu \((inputs, grad_outputs)\)
Applies backprop to output gradient arrays on GPU.

Parameters

- **inputs** – Tuple of input \texttt{cupy.ndarray} object(s).
- **grad_outputs** – Tuple of output gradient \texttt{cupy.ndarray} object(s).

Returns Tuple of input gradient \texttt{cupy.ndarray} object(s). Some or all of them can be None, if the function is not differentiable on corresponding inputs.

Return type \texttt{tuple}

**Warning:** Implementations of \textit{Function} must take care that the return value must be a tuple even if it returns only one array.

check_type_forward \((in_types)\)
Checks types of input data before forward propagation.

Before \texttt{forward()} is called, this function is called. You need to validate types of input data in this function using \texttt{the type checking utilities}.

Parameters **in_types** \((TypeInfoTuple)\) – The type information of input data for \texttt{forward()}.

delete_hook \((name)\)
Unregisters the function hook.

Parameters **name** \((str)\) – the name of the function hook to be unregistered.

forward \((inputs)\)
Applies forward propagation to input arrays.

It delegates the procedure to \texttt{forward_cpu()} or \texttt{forward_gpu()} by default. Which it selects is determined by the type of input arrays. Implementations of \textit{Function} must implement either CPU/GPU methods or this method.

Parameters **inputs** – Tuple of input array(s).

Returns Tuple of output array(s).
**Warning:** Implementations of *Function* must take care that the return value must be a tuple even if it returns only one array.

### forward_cpu(inputs)
Applies forward propagation to input arrays on CPU.

**Parameters** inputs – Tuple of `numpy.ndarray` object(s).

**Returns** Tuple of `numpy.ndarray` object(s).

**Return type** tuple

**Warning:** Implementations of *Function* must take care that the return value must be a tuple even if it returns only one array.

### forward_gpu(inputs)
Applies forward propagation to input arrays on GPU.

**Parameters** inputs – Tuple of `cupy.ndarray` object(s).

**Returns** Tuple of `cupy.ndarray` object(s).

**Return type** tuple

**Warning:** Implementations of *Function* must take care that the return value must be a tuple even if it returns only one array.

### retain_inputs(indexes)
Lets specified input variable nodes keep data arrays.

By calling this method from *forward()*, the function can specify which inputs are required for backprop. If this method is not called, the function keeps all input arrays. If you want to release all input arrays, call this method by passing an empty sequence.

Note that **this method must not be called from the outside of forward method.**

**Parameters**
- indexes *(iterable of int)* – Indexes of input variables that the function does not require for backprop.

### retain_outputs(indexes, retain_after_backward=False)
Lets specified output variable nodes keep data arrays.

By calling this method from *forward()*, the function can specify which outputs are required for backprop. If this method is not called, any output variables are not marked to keep the data array at the point of returning from `__call__()` . The retained arrays are stored to `output_data`.

**Note:** It is STRONGLY RECOMMENDED to use this method if the function requires some or all output arrays in backprop. The function can also use output arrays just by keeping references to them directly, whereas it might influence on the performance of later function applications to the output variables.

Note that **this method must not be called from the outside of forward method.**

**Parameters**
• **indexes** *(iterable of int)* – Indexes of input variables that the function does not require for backprop.

• **retain_after_backward** *(bool)* – If True, a reference to the outputs will remain after the backprop of the function is over. If False, the reference will be deleted.

`unchain()`

Purges in/out nodes and this function itself from the graph.

This method is called from `Variable.unchain_backward()` method.

**Attributes**

**label**

Short text that represents the function.

The default implementation returns its type name. Each function should override it to give more information.

**local_function_hooks**

Ordered Dictionary of registered function hooks.

Contrary to `chainer.thread_local.function_hooks`, which registers its elements to all functions, Function hooks in this property is specific to this function.

**rank** = 0

**stack**

### `chainer.force_backprop_mode`

**`chainer.force_backprop_mode()`**

Make a context manager which enables back-propagation.

When you want to enable back-propagation in `no_backprop_mode()`, call this method. *chainer.Variable* created in this context always has a computational graph. If you call this method outside of `no_backprop_mode()` context, it changes nothing.

In this example, `y` has a computational graph and `y.backward` computes gradients of variables in the graph.

```python
>>> with chainer.no_backprop_mode():
...   with chainer.force_backprop_mode():
...     y = x + 1
```

**See also:**

See `no_backprop_mode()` for details of back-prop mode.

### `chainer.no_backprop_mode`

**`chainer.no_backprop_mode()`**

Make a context manager which disables back-propagation.

In this context, Chainer does not make a computational graph. *Variable* created in this context does not have reference to the `Function` which created the variable. So, you cannot compute gradient with `backward()`. Instead memory consumption is reduced.

In this example, `y` is created in this context. So you cannot call `backward()`.

3.1. Core functionalities
>>> x = chainer.Variable(numpy.array([1,], 'f'))
>>> with chainer.no_backprop_mode():
...    y = x + 1

## Link and Chain

<table>
<thead>
<tr>
<th><strong>chainer.Link</strong></th>
<th>Building block of model definitions.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>chainer.Chain</strong></td>
<td>Composable link with object-like interface.</td>
</tr>
<tr>
<td><strong>chainer.ChainList</strong></td>
<td>Composable link with list-like interface.</td>
</tr>
</tbody>
</table>

### chainer.Link

```python
class chainer.Link(**params):
    Building block of model definitions.
```

Link is a building block of neural network models that support various features like handling parameters, defining network fragments, serialization, etc.

Link is the primitive structure for the model definitions. It supports management of parameter variables and persistent values that should be incorporated to serialization.

Parameter is an instance of Parameter registered to a link. A Parameter object can be registered as a parameter of the link by assigning it to an attribute within an initialization scope, which is a code surrounded by a `init_scope()` context manager using the `with` statement.

Persistent values are arrays, scalars, or any other serializable values registered via `register_persistent()` or `add_persistent()`.

**Note:** Whereas arbitrary serializable objects can be registered as persistent values, it is strongly recommended to just register values that should be treated as results of learning. A typical example of persistent values is ones computed during training and required for testing, e.g. running statistics for batch normalization.

Parameters and persistent values are referred by their names. They can be accessed as attributes of the links. Link class itself manages the lists of names of parameters and persistent values to distinguish parameters and persistent values from other attributes.

Link can be composed into more complex models. This composition feature is supported by child classes like `Chain` and `ChainList`. One can create a chain by combining one or more links. See the documents for these classes for details.

As noted above, Link supports the serialization protocol of the Serializer class. **Note that only parameters and persistent values are saved and loaded.** Other attributes are considered as a part of user program (i.e. a part of network definition). In order to construct a link from saved file, other attributes must be identically reconstructed by user codes.

**Example**

This is a simple example of custom link definition. Chainer itself also provides many links defined under the `links` module. They might serve as examples, too.

Consider we want to define a simple primitive link that implements a fully-connected layer based on the `linear()` function. Note that this function takes input units, a weight variable, and a bias variable as arguments. Then, the fully-connected layer can be defined as follows:
import chainer
import chainer.functions as F
from chainer import initializers
import numpy as np

class LinearLayer(chainer.Link):
    def __init__(self, n_in, n_out):
        super(LinearLayer, self).__init__()
        with self.init_scope():
            self.b = chainer.Parameter((n_out,), initializers.Zero())

    def __call__(self, x):
        return F.linear(x, self.W, self.b)

This example shows that a user can define arbitrary parameters and use them in any methods. Links typically implement the __call__ operator, although they can also provide other methods to implement the forward propagation.

Parameters

- **params** — *(deprecated since v2.0.0)* Names, shapes, and optional dtypes of initial parameters. The keywords are used as the parameter names and the corresponding values consist either of the shape or a tuple of shape and a dtype (shape, dtype). If only the shape is supplied, the default dtype will be used.

Variables

- **name** *(str)* — Name of this link, given by the parent chain (if exists).

Methods

- **add_param** *(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)* Registers a parameter to the link.
  
  Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

  can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

  The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- **name** *(str)* — Name of the parameter. This name is also used as the attribute name.
- **shape** *(int or tuple of ints)* — Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** — Data type of the parameter array.
• **initializer** – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

**add_persistent**(*name*, *value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name** *(str)* – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads**(*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

- **link** *(Link)* – Source link object.

**children**()

Returns a generator of all child links.

**Returns** A generator object that generates all child links.

**cleargrads**()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy**()

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns** Copied link object.

**Return type** *Link*

**copyparams**(*link*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

- **link** *(Link)* – Source link object.

**disable_update**()

Disables update rules of all parameters under the link hierarchy.

This method sets the `chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

**enable_update**()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`. 
init_scope (*args, **kwds)  
  Creates an initialization scope.

  This method returns a context manager object that enables registration of parameters (and links for Chain) 
  by an assignment. A Parameter object can be automatically registered by assigning it to an attribute 
  under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the init_scope 
method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (skipself=False)  
  Returns a generator of all links under the hierarchy.

  Parameters skipself (bool) – If True, then the generator skips this link and starts with the 
  first child link.

  Returns A generator object that generates all links.

namedlinks (skipself=False)  
  Returns a generator of all (path, link) pairs under the hierarchy.

  Parameters skipself (bool) – If True, then the generator skips this link and starts with the 
  first child link.

  Returns A generator object that generates all (path, link) pairs.

namedparams (include_uninit=True)  
  Returns a generator of all (path, param) pairs under the hierarchy.

  Parameters include_uninit (bool) – If True, it also generates uninitialized parameters.

  Returns A generator object that generates all (path, parameter) pairs. The paths are relative from 
  this link.

params (include_uninit=True)  
  Returns a generator of all parameters under the link hierarchy.

  Parameters include_uninit (bool) – If True, it also generates uninitialized parameters.

  Returns A generator object that generates all parameters.

register_persistent (name)  
  Registers an attribute of a given name as a persistent value.

  This is a convenient method to register an existing attribute as a persistent value. If name has been already 
  registered as a parameter, this method removes it from the list of parameter names and re-registers it as a 
  persistent value.

  Parameters name (str) – Name of the attribute to be registered.

serialize (serializer)  
  Serializes the link object.
Parameters **serializer** *(AbstractSerializer)* – Serializer object.

to_cpu()
Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

Returns: self
to_gpu(*device=None*)
Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

Parameters **device** – Target device specifier. If omitted, the current device is used.

Returns: self
zerograd()
Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use *cleargrads()* instead.

Attributes

**update_enabled**
True if at least one parameter has an update rule enabled.

**within_init_scope**
True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

**xp**
Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns *numpy* or *cupy*.

**chainer.Chain**

class *chainer.Chain(**links**)*
Composable link with object-like interface.

Composability is one of the most important features of neural nets. Neural net models consist of many reusable fragments, and each model itself might be embedded into a larger learnable system. Chain enables us to write a neural net based on composition, without bothering about routine works like collecting parameters, serialization, copying the structure with parameters shared, etc.

This class actually provides a way to compose one or more links into one structure. A chain can contain one or more *child links*. Child link is a link registered to the chain with its own name. The child link is stored to an attribute of the chain with the name. User can write a whole model or a fragment of neural nets as a child class of Chain.

Each chain itself is also a link. Therefore, one can combine chains into higher-level chains. In this way, links and chains construct a *link hierarchy*. Link hierarchy forms a tree structure, where each node is identified by the
path from the root. The path is represented by a string like a file path in UNIX, consisting of names of nodes on
the path, joined by slashes /.

A child link can be added just by assigning it to an attribute of the chain within an initialization
scope.

The registered child link is saved and loaded on serialization and deserialization, and involved in the optimiza-
tion. The registered link is called a child. The child link is accessible via children() generator, which
returns a generator running through the children in registered order.

On registration of a child link, its name attribute is also set (or overwritten if the link has already been registered
to another chain).

Example

This is a simple example of custom chain definition. Chainer itself also provides some chains defined under the
links module. They might serve as examples, too.

Consider we want to define a multi-layer perceptron consisting of two hidden layers with rectifiers as activation
functions. We can use the Linear link as a building block:

```python
import chainer
import chainer.functions as F
import chainer.links as L

class MultiLayerPerceptron(chainer.Chain):
    def __init__(self, n_in, n_hidden, n_out):
        super(MultiLayerPerceptron, self).__init__()
        with self.init_scope():
            self.layer1 = L.Linear(n_in, n_hidden)
            self.layer2 = L.Linear(n_hidden, n_hidden)
            self.layer3 = L.Linear(n_hidden, n_out)

    def __call__(self, x):
        # Forward propagation
        h1 = F.relu(self.layer1(x))
        h2 = F.relu(self.layer2(h1))
        return self.layer3(h2)
```

Child links are registered via the assignment within a with self.init_scope(): block. The forward
propagation is often implemented as the __call__ operator as the above example, though it is not mandatory.

Parameters

**links** – Child links. The keywords are used as their names. The names are also set to
the links.

Deprecated since version v2.0.0: Assign child links directly to attributes, instead.

Methods

**__getitem__** (name)

Equivalent to getattr.

**add_link** (name, link)

Registers a child link to this chain.
Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('l1', L.Linear(3, 5))
```

can be replaced by the following line.

```python
with self.init_scope():
    chain.l1 = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name (str)** – Name of the child link. This name is also used as the attribute name.
- **link (Link)** – The link object to be registered.

**add_param** *(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)*

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name (str)** – Name of the parameter. This name is also used as the attribute name.
- **shape (int or tuple of ints)** – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

**add_persistent** *(name, value)*

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name (str)** – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads** *(link)*

**children** ()
cleargrads()
Clears all gradient arrays.
This method should be called before the backward computation at every iteration of the optimization.

copy()
copyparams(link)
disable_update()
Disables update rules of all parameters under the link hierarchy.
This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.
enable_update()
Enables update rules of all parameters under the link hierarchy.
This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope(*args, **kwds)
Creates an initialization scope.
This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example
In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))

links(skipself=False)
namedlinks(skipself=False)
namedparams(include_uninit=True)
params(include_uninit=True)

register_persistent(name)
Registers an attribute of a given name as a persistent value.
This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters name (str) – Name of the attribute to be registered.

serialize(name)
to_cpu()
to_gpu(device=None)
zerograds()

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.

**Attributes**

**update_enabled**

True if at least one parameter has an update rule enabled.

**within_init_scope**

True if the current code is inside of an initialization scope.

See init_scope() for the details of the initialization scope.

**xp**

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

**chainer.ChainList**

class chainer.ChainList(*links)

Composable link with list-like interface.

This is another example of compositional link. Unlike Chain, this class can be used like a list of child links. Each child link is indexed by a non-negative integer, and it maintains the current number of registered child links. The add_link() method inserts a new link at the end of the list. It is useful to write a chain with arbitrary number of child links, e.g. an arbitrarily deep multi-layer perceptron.

Note that this class does not implement all methods of list.

**Parameters**

**links** – Initial child links.

**Methods**

**__getitem__(index)**

Returns the child at given index.

**Parameters**

**index** (*int*) – Index of the child in the list.

**Returns**

The index-th child link.

**Return type**

**Link**

**__len__()**

Returns the number of children.

**__iter__()**

add_link(link)

Registers a child link and adds it to the tail of the list.

**Parameters**

**link** (*Link*) – The link object to be registered.
add_param (name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)

Registers a parameter to the link.

Depreciated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- **name (str)** – Name of the parameter. This name is also used as the attribute name.
- **shape (int or tuple of ints)** – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent (name, value)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name (str)** – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (link)

append (link)

Registers a child link and adds it to the tail of the list.

This is equivalent to add_link(). This method has been added to emulate the list interface.

Parameters **link (Link)** – The link object to be registered.

children ()

cleargrads ()

Cleans all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy ()

copyparams (link)

disable_update ()

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.
enable_update()
   Enables update rules of all parameters under the link hierarchy.

   This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope(*args, **kwds)
   Creates an initialization scope.

   This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (skipself=False)

namedlinks (skipself=False)

namedparams (include_uninit=True)

params (include_uninit=True)

register_persistent (name)
   Registers an attribute of a given name as a persistent value.

   This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

   Parameters  name (str) – Name of the attribute to be registered.

serialize (serializer)

to_cpu ()

to_gpu (device=None)

zerograds ()
   Initializes all gradient arrays by zero.

   This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

   Deprecated since version v1.15: Use cleargrads() instead.

Attributes

update_enabled
   True if at least one parameter has an update rule enabled.
**within_init_scope**

True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

**xp**

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns *numpy* or *cupy*.

---

**Optimizer**

<table>
<thead>
<tr>
<th>chainer.Hyperparameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.UpdateRule</td>
</tr>
<tr>
<td>chainer.Optimizer</td>
</tr>
<tr>
<td>chainer.GradientMethod</td>
</tr>
</tbody>
</table>

**chainer.UpdateRule**

class chainer.UpdateRule(*parent_hyperparam=None*)

Base class of all update rules.

Update rule is an object that implements how to update one parameter variable using the gradient of a loss function. This class provides the interface and the common features of any update rules.

An update rule can be set to a *Variable* object that represents a parameter array of a model. An *Optimizer* instance defines which parameters to update, and the update rule instance of each parameter defines how to update it.

Hook functions can be set to any update rule instance. The hook function is called just before any updates in the order of registrations.

An implementation of update rule should override *update_core()* or its device-dependent variants (i.e., *update_core_cpu()* and *update_core_gpu()*).

The state (e.g. a moving average of the gradient) of the update rule is stored into the state dictionary. An implementation of update rule using state should also override *init_state()* to initialize the state at the first update. The values of the state dictionary are automatically copied to the appropriate device before the update based on the data and grad arrays.

**Parameters**

- **parent_hyperparam** (*Hyperparameter*) – Hyperparameter that provides the default values.

**Variables**

- **enabled** (*bool*) – Flag to configure if this update rule is active. If the update rule is not active (i.e., *enabled = False*), the *update()* method does not update the parameter.
- **hyperparam** (*Hyperparameter*) – Hyperparameter of the update rule.
- **t** (*int*) – Number of updates made by this update rule.

**Methods**

- **add_hook** (*hook, name=None*)

  Adds a hook function.

  The hook function is called before any updates.
Parameters

- **hook** *(callable)* – Hook function to be added. It takes two arguments: the update rule object and the parameter variable.
- **name** *(str)* – Name of the hook function. The name attribute of the hook function is used by default.

`init_state (param)`

Initializes the state.

Any implementations that use the state should override this method. This method is called at the first update.

**Parameters**

- **param** *(Variable)* – Parameter variable. It can be used to extract the shape and the data type of the parameter.

`remove_hook (name)`

Removes the specified hook function.

**Parameters**

- **name** *(str)* – Name of the hook function to be removed. The hook function registered with this name will be removed.

`serialize (serializer)`

Serializes the update rule state.

Be careful that this method only saves/loads the state of the update rule. The parameters of the target link is not saved/loaded by this method, and so you need to serialize the target link separately if you want to fully recover the training state including parameters.

**Parameters**

- **serializer** *(AbstractSerializer)* – Serializer object.

`update (param)`

Invokes hook functions and updates the parameter.

**Parameters**

- **param** *(Variable)* – Variable to be updated.

`update_core (param)`

Updates the parameter.

Implementation of UpdateRule should override this method or both of _update_core_cpu() and _update_core_gpu().

**Parameters**

- **param** *(Variable)* – Variable to be updated.

`update_core_cpu (param)`

Updates the parameter on CPU.

See `update_core()` for details.

**Parameters**

- **param** *(Variable)* – Variable to be updated.

`update_core_gpu (param)`

Updates the parameter on GPU.

See `update_core()` for details.

**Parameters**

- **param** *(Variable)* – Variable to be updated.

Attributes

**state**

State dictionary.
chainer.Optimizer

class chainer.Optimizer
Base class of all numerical optimizers.

This class provides basic features for all optimization methods. It optimizes parameters of a target link. The target link is registered via the setup() method, and then the update() method updates its parameters based on a given loss function.

Each optimizer implementation must be defined as a child class of Optimizer. It must override update() method.

If the optimizer is based on single gradient computation (like most first-order methods), then it should inherit GradientMethod, which adds some features dedicated for the first order methods, including the support of UpdateRule.

Optimizer instance also supports hook functions. Hook function is registered by the add_hook() method. Each hook function is called in registration order in advance of the actual parameter update. If the hook function has an attribute call_for_each_param and its value is True, the hook function is used as a hook function of all update rules (i.e., it is invoked for every parameter by passing the corresponding update rule and the parameter).

Variables

• target – Target link object. It is set by the setup() method.
• t – Number of update steps. It must be incremented by the update() method.
• epoch – Current epoch. It is incremented by the new_epoch() method.

Methods

add_hook (hook, name=None)
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method.

Parameters

• hook (function) – Hook function. If hook.call_for_each_param is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
• name (str) – Name of the registration. If omitted, hook.name is used by default.

call_hooks ()
Invokes hook functions in registration order.

new_epoch ()
Starts a new epoch.

This method increments the epoch count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

remove_hook (name)
Removes a hook function.

Parameters name (str) – Registered name of the hook function to remove.
serialize(serializer)
Serializes or deserializes the optimizer.
It only saves or loads the following things:
• Optimizer states
• Global states (t and epoch)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters serializer(AbstractSerializer) – Serializer or deserializer object.

setup(link)
Sets a target link and initializes the optimizer states.
Given link is set to the target attribute. It also prepares the optimizer state dictionaries corresponding to all parameters in the link hierarchy. The existing states are discarded.

Parameters link(Links) – Target link object.

update(lossfun=None, *args, **kwds)
Updates the parameters.
This method updates the parameters of the target link. The behavior of this method is different for the cases either lossfun is given or not.
If lossfun is given, this method typically clears the gradients, calls the loss function with given extra arguments, and calls the backward() method of its output to compute the gradients. The actual implementation might call lossfun more than once.
If lossfun is not given, then this method assumes that the gradients of all parameters are already computed. An implementation that requires multiple gradient computations might raise an error on this case.
In both cases, this method invokes the update procedure for all parameters.

Parameters
• lossfun(function) – Loss function. It accepts arbitrary arguments and returns one Variable object that represents the loss (or objective) value. This argument can be omitted for single gradient-based methods. In this case, this method assumes gradient arrays computed.
• kwds(args,) – Arguments for the loss function.

Chainer.GradientMethod

class chainer.GradientMethod
Base class of all single gradient-based optimizers.
This is an extension of the Optimizer class. Typical gradient methods that just require the gradient at the current parameter vector on an update can be implemented as its child class.
This class uses UpdateRule to manage the update rule of each parameter. A child class of GradientMethod should override create_update_rule() to create the default update rule of each parameter.
This class also provides hyperparam, which is the hyperparameter used as the default configuration of each update rule. All built-in gradient method implementations also provide proxy properties that act as aliases to the attributes of hyperparam. It is recommended to provide such an alias to each attribute. It can be done by only adding one line for each attribute using HyperparameterProxy.
Variables `hyperparam` (*Hyperparameter*) – The hyperparameter of the gradient method. It is used as the default configuration of each update rule (i.e., the hyperparameter of each update rule refers this hyperparameter as its parent).

Methods

`add_hook (hook, name=None)`
Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method.

Parameters

- `hook` (*function*) – Hook function. If `hook.call_for_each_param` is `true`, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.

- `name` (*str*) – Name of the registration. If omitted, `hook.name` is used by default.

`call_hooks ()`
Invokes hook functions in registration order.

`create_update_rule ()`
Creates a new update rule object.

This method creates an update rule object. It is called by `setup()` to set up an update rule of each parameter. Each implementation of the gradient method should override this method to provide the default update rule implementation.

Returns Update rule object.

Return type `UpdateRule`

`new_epoch ()`
Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

`reallocate_cleared_grads ()`
Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

`remove_hook (name)`
Removes a hook function.

Parameters `name` (*str*) – Registered name of the hook function to remove.

`serialize (serializer)`
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (`t` and `epoch`)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

3.1. Core functionalities
Parameters `serializer` (AbstractSerializer) – Serializer or deserializer object.

`setup(link)`

`update(lossfun=None, *args, **kwds)`

Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

`use_cleargrads(use=True)`

Enables or disables use of `cleargrads()` in `update`.

Parameters `use` (bool) – If True, this function enables use of `cleargrads`. If False, disables use of `cleargrads` (zerograds is used).

Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

Hook functions

<table>
<thead>
<tr>
<th>hook</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.optimizer.WeightDecay</td>
<td>Optimizer/UpdateRule hook function for weight decay regularization.</td>
</tr>
<tr>
<td>chainer.optimizer.Lasso</td>
<td>Optimizer/UpdateRule hook function for Lasso regularization.</td>
</tr>
<tr>
<td>chainer.optimizer.GradientClipping</td>
<td>Optimizer hook function for gradient clipping.</td>
</tr>
<tr>
<td>chainer.optimizer.GradientNoise</td>
<td>Optimizer/UpdateRule hook function for adding gradient noise.</td>
</tr>
</tbody>
</table>

`chainer.optimizer.WeightDecay`

Class `chainer.optimizer.WeightDecay(rate)`

Optimizer/UpdateRule hook function for weight decay regularization.

This hook function adds a scaled parameter to the corresponding gradient. It can be used as a regularization.

Parameters `rate` (float) – Coefficient for the weight decay.

Variables `rate` (float) – Coefficient for the weight decay.

Methods

`__call__(rule, param)`

Attributes

`call_for_each_param` = True

`name` = ‘WeightDecay’
### chainer.optimizer.Lasso

**class** `chainer.optimizer.Lasso(rate)`

Optimizer/UpdateRule hook function for Lasso regularization. This hook function adds a scaled parameter to the sign of each weight. It can be used as a regularization.

- **Parameters**
  - `rate (float)`: Coefficient for the weight decay.

- **Variables**
  - `rate (float)`: Coefficient for the weight decay.

**Methods**

- `__call__(rule, param)`

**Attributes**

- `call_for_each_param = True`
- `name = ‘Lasso’`

### chainer.optimizer.GradientClipping

**class** `chainer.optimizer.GradientClipping(threshold)`

Optimizer hook function for gradient clipping. This hook function scales all gradient arrays to fit to the defined L2 norm threshold.

- **Parameters**
  - `threshold (float)`: L2 norm threshold.

- **Variables**
  - `threshold (float)`: L2 norm threshold of gradient norm.

**Methods**

- `__call__(opt)`

**Attributes**

- `name = ‘GradientClipping’`

### chainer.optimizer.GradientNoise

**class** `chainer.optimizer.GradientNoise(eta, noise_func=<function exponential_decay_noise>)`

Optimizer/UpdateRule hook function for adding gradient noise. This hook function simply adds noise generated by the `noise_func` to the gradient. By default it adds time-dependent annealed Gaussian noise to the gradient at every training step:

\[ g_t \leftarrow g_t + N(0, \sigma_t^2) \]
where

\[ \sigma_i^2 = \frac{\eta}{(1 + t)^\gamma} \]

with \(\eta\) selected from \{0.01, 0.3, 1.0\} and \(\gamma = 0.55\).

**Parameters**

- \texttt{eta (float)} – Parameter that defines the scale of the noise, which for the default noise function is recommended to be either 0.01, 0.3 or 1.0.
- \texttt{noise_func (function)} – Noise generating function which by default is given by "Adding Gradient Noise Improves Learning for Very Deep Networks."

**Methods**

\texttt{__call__ (rule, param)}

**Attributes**

- \texttt{call_for_each_param = True}
- \texttt{name = ‘GradientNoise’}

**Serializer**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{chainer.AbstractSerializer}</td>
<td>Abstract base class of all serializers and deserializers.</td>
</tr>
<tr>
<td>\texttt{chainer.Serializer}</td>
<td>Base class of all serializers.</td>
</tr>
<tr>
<td>\texttt{chainer.Deserializer}</td>
<td>Base class of all deserializers.</td>
</tr>
</tbody>
</table>

**Methods**

\texttt{__call__ (key, value)}

Serializes or deserializes a value by given name.

This operator saves or loads a value by given name.

If this is a serializer, then the value is simply saved at the key. Note that some type information might be missed depending on the implementation (and the target file format).

If this is a deserializer, then the value is loaded by the key. The deserialization differently works on scalars and arrays. For scalars, the value argument is used just for determining the type of restored value to be converted, and the converted value is returned. For arrays, the restored elements are directly copied into the value argument. String values are treated like scalars.

**Note:** As of v2.0.0, serializers and deserializers are required to correctly handle the \texttt{None} value. When \texttt{value} is \texttt{None}, serializers save it in format-dependent ways, and deserializers just return the loaded
value. When the saved None value is loaded by a deserializer, it should quietly return the None value without modifying the value object.

Parameters

- **key** *(str)* – Name of the serialization entry.
- **value** *(scalar, array, None, or str)* – Object to be (de)serialized. None is only supported by deserializers.

Returns  Serialized or deserialized value.

__getitem__ *(key)*

Gets a child serializer.

This operator creates a _child_ serializer represented by the given key.

Parameters  **key** *(str)* – Name of the child serializer.

chainer.Serializer

class chainer.Serializer
Base class of all serializers.

Methods

__call__ *(key, value)*

Serializes or deserializes a value by given name.

This operator saves or loads a value by given name.

If this is a serializer, then the value is simply saved at the key. Note that some type information might be missed depending on the implementation (and the target file format).

If this is a deserializer, then the value is loaded by the key. The deserialization differently works on scalars and arrays. For scalars, the value argument is used just for determining the type of restored value to be converted, and the converted value is returned. For arrays, the restored elements are directly copied into the value argument. String values are treated like scalars.

Note:  As of v2.0.0, serializers and deserializers are required to correctly handle the None value. When value is None, serializers save it in format-dependent ways, and deserializers just return the loaded value. When the saved None value is loaded by a deserializer, it should quietly return the None value without modifying the value object.

Parameters

- **key** *(str)* – Name of the serialization entry.
- **value** *(scalar, array, None, or str)* – Object to be (de)serialized. None is only supported by deserializers.

Returns  Serialized or deserialized value.
__getitem__(key)

Gets a child serializer.

This operator creates a _child_ serializer represented by the given key.

Parameters

key (str) – Name of the child serializer.

dssave(obj)

Saves an object by this serializer.

This is equivalent to obj.serialize(self).

Parameters

obj – Target object to be serialized.

chainer.Deserializer

class chainer.Deserializer

Base class of all deserializers.

Methods

__call__(key, value)

Serializes or deserializes a value by given name.

This operator saves or loads a value by given name.

If this is a serializer, then the value is simply saved at the key. Note that some type information might be missed depending on the implementation (and the target file format).

If this is a deserializer, then the value is loaded by the key. The deserialization differently works on scalars and arrays. For scalars, the value argument is used just for determining the type of restored value to be converted, and the converted value is returned. For arrays, the restored elements are directly copied into the value argument. String values are treated like scalars.

Note: As of v2.0.0, serializers and deserializers are required to correctly handle the None value. When value is None, serializers save it in format-dependent ways, and deserializers just return the loaded value. When the saved None value is loaded by a deserializer, it should quietly return the None value without modifying the value object.

Parameters

- key (str) – Name of the serialization entry.
- value (scalar, array, None, or str) – Object to be (de)serialized. None is only supported by deserializers.

Returns

Serialized or deserialized value.

__getitem__(key)

Gets a child serializer.

This operator creates a _child_ serializer represented by the given key.

Parameters

key (str) – Name of the child serializer.
**load**(obj)

Loads an object from this deserializer.

This is equivalent to `obj.serialize(self)`.

**Parameters**

- **obj** – Target object to be serialized.

---

## Dataset abstraction

Chainer has a support of common interface of training and validation datasets. The dataset support consists of three components: datasets, iterators, and batch conversion functions.

**Dataset** represents a set of examples. The interface is only determined by combination with iterators you want to use on it. The built-in iterators of Chainer requires the dataset to support `__getitem__` and `__len__` method. In particular, the `__getitem__` method should support indexing by both an integer and a slice. We can easily support slice indexing by inheriting `DatasetMixin`, in which case users only have to implement `get_example()` method for indexing. Some iterators also restrict the type of each example. Basically, datasets are considered as *stateless* objects, so that we do not need to save the dataset as a checkpoint of the training procedure.

**Iterator** iterates over the dataset, and at each iteration, it yields a mini batch of examples as a list. Iterators should support the `Iterator` interface, which includes the standard iterator protocol of Python. Iterators manage where to read next, which means they are *stateful*.

**Batch conversion function** converts the mini batch into arrays to feed to the neural nets. They are also responsible to send each array to an appropriate device. Chainer currently provides `concat_examples()` as the only example of batch conversion functions.

These components are all customizable, and designed to have a minimum interface to restrict the types of datasets and ways to handle them. In most cases, though, implementations provided by Chainer itself are enough to cover the usages.

Chainer also has a light system to download, manage, and cache concrete examples of datasets. All datasets managed through the system are saved under *the dataset root directory*, which is determined by the `CHAINER_DATASET_ROOT` environment variable, and can also be set by the `set_dataset_root()` function.

### Dataset representation

See [Dataset examples](#) for dataset implementations.

```python
chainer.dataset.DatasetMixin
```

Default implementation of dataset indexing.

**chainer.dataset.DatasetMixin**

```python
class chainer.dataset.DatasetMixin
```

Default implementation of dataset indexing.

DatasetMixin provides the `__getitem__()` operator. The default implementation uses `get_example()` to extract each example, and combines the results into a list. This mixin makes it easy to implement a new dataset that does not support efficient slicing.

Dataset implementation using DatasetMixin still has to provide the `__len__()` operator explicitly.
Methods

__getitem__(index)
Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the
get_example() method by default, but it may be overridden by the implementation to, for example,
 improve the slicing performance.

Parameters
index (int, slice, list or numpy.ndarray) – An index of an example or indexes of examples.

Returns
If index is int, returns an example created by get_example. If index is either slice or
one-dimensional list or numpy.ndarray, returns a list of examples created by get_example.

Example

```python
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]  # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

__len__()
Returns the number of data points.

get_example(i)
Returns the i-th example.

Implementations should override it. It should raise IndexError if the index is invalid.

Parameters
i (int) – The index of the example.

Returns
The i-th example.

Iterator interface

See Iterator examples for dataset iterator implementations.
chainer.dataset.Iterator

class chainer.dataset.Iterator
Base class of all dataset iterators.

Iterator iterates over the dataset, yielding a minibatch at each iteration. Minibatch is a list of examples. Each implementation should implement an iterator protocol (e.g., the __next__() method).

Note that, even if the iterator supports setting the batch size, it does not guarantee that each batch always contains the same number of examples. For example, if you let the iterator to stop at the end of the sweep, the last batch may contain a fewer number of examples.

The interface between the iterator and the underlying dataset is not fixed, and up to the implementation.

Each implementation should provide the following attributes (not needed to be writable).

• batch_size: Number of examples within each minibatch.
• epoch: Number of completed sweeps over the dataset.
• epoch_detail: Floating point number version of the epoch. For example, if the iterator is at the middle of the dataset at the third epoch, then this value is 2.5.
• previous_epoch_detail: The value of epoch_detail at the previous iteration. This value is None before the first iteration.
• is_new_epoch: True if the epoch count was incremented at the last update.

Each implementation should also support serialization to resume/suspend the iteration.

Methods

__next__()  
Returns the next batch.

This is a part of the iterator protocol of Python. It may raise the StopIteration exception when it stops the iteration.

__iter__()  
Returns self.

finalize()  
Finalizes the iterator and possibly releases the resources.

This method does nothing by default. Implementation may override it to better handle the internal resources.

next()  
Python2 alternative of __next__.

It calls __next__() by default.

serialize(serializer)  
Serializes the internal state of the iterator.

This is a method to support serializer protocol of Chainer.

Note: It should only serialize the internal state that changes over the iteration. It should not serializes what is set manually by users such as the batch size.

3.1. Core functionalities 85
Batch conversion function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>Concatenates a list of examples into array(s).</td>
</tr>
<tr>
<td>Dataset</td>
<td>Sends an array to a given device.</td>
</tr>
</tbody>
</table>

chainer.dataset.concat_examples

chainer.dataset.concat_examples(batch, device=None, padding=None)
Concatenates a list of examples into array(s).

Dataset iterator yields a list of examples. If each example is an array, this function concatenates them along
the newly-inserted first axis (called batch dimension) into one array. The basic behavior is same for examples
consisting of multiple arrays, i.e., corresponding arrays of all examples are concatenated.

For instance, consider each example consists of two arrays \((x, y)\). Then, this function concatenates \(x\)'s into
one array, and \(y\)'s into another array, and returns a tuple of these two arrays. Another example: consider each
example is a dictionary of two entries whose keys are \('x'\) and \('y'\), respectively, and values are arrays. Then,
this function concatenates \(x\)'s into one array, and \(y\)'s into another array, and returns a dictionary with two
entries \(x\) and \(y\) whose values are the concatenated arrays.

When the arrays to concatenate have different shapes, the behavior depends on the padding value. If
padding is None (default), it raises an error. Otherwise, it builds an array of the minimum shape that the
contents of all arrays can be substituted to. The padding value is then used to the extra elements of the resulting
arrays.

TODO(beam2d): Add an example.

**Parameters**

- **batch** (list) – A list of examples. This is typically given by a dataset iterator.
- **device** (int) – Device ID to which each array is sent. Negative value indicates the host
  memory (CPU). If it is omitted, all arrays are left in the original device.
- **padding** – Scalar value for extra elements. If this is None (default), an error is raised on
  shape mismatch. Otherwise, an array of minimum dimensionalities that can accommodate
  all arrays is created, and elements outside of the examples are padded by this value.

**Returns** Array, a tuple of arrays, or a dictionary of arrays. The type depends on the type of each
example in the batch.

chainer.dataset.to_device

chainer.dataset.to_device(device, x)
Send an array to a given device.

This method sends a given array to a given device. This method is used in concat_examples().
You can also use this method in a custom converter method used in Updater and Extension such as
StandardUpdater and Evaluator.

See also chainer.dataset.concat_examples().

**Parameters**

- **device** (int or None) – Device ID to which an array is sent. If it is negative value,
an array is sent to CPU. If it is positive, an array is sent to GPU with the given ID. If it is
  None, an array is left in the original device.
* `x (numpy.ndarray or cupy.ndarray)` – An array to send.

Returns Converted array.

**Dataset management**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.dataset.get_dataset_root</code></td>
<td>Gets the path to the root directory to download and cache datasets.</td>
</tr>
<tr>
<td><code>chainer.dataset.set_dataset_root</code></td>
<td>Sets the root directory to download and cache datasets.</td>
</tr>
<tr>
<td><code>chainer.dataset.cached_download</code></td>
<td>Downloads a file and caches it.</td>
</tr>
<tr>
<td><code>chainer.dataset.cache_or_load_file</code></td>
<td>Caches a file if it does not exist, or loads it otherwise.</td>
</tr>
</tbody>
</table>

**chainer.dataset.get_dataset_root**

`chainer.dataset.get_dataset_root()`  
Gets the path to the root directory to download and cache datasets.  

Returns The path to the dataset root directory.  

Return type `str`

**chainer.dataset.set_dataset_root**

`chainer.dataset.set_dataset_root(path)`  
Sets the root directory to download and cache datasets.  

There are two ways to set the dataset root directory. One is by setting the environment variable `CHAINER_DATASET_ROOT`. The other is by using this function. If both are specified, one specified via this function is used. The default dataset root is `$HOME/.chainer/dataset`.

Parameters `path (str)` – Path to the new dataset root directory.

**chainer.dataset.cached_download**

`chainer.dataset.cached_download(url)`  
Downloads a file and caches it.  

It downloads a file from the URL if there is no corresponding cache. After the download, this function stores a cache to the directory under the dataset root (see `set_dataset_root()`). If there is already a cache for the given URL, it just returns the path to the cache without downloading the same file.

Parameters `url (str)` – URL to download from.  

Returns Path to the downloaded file.  

Return type `str`

**chainer.dataset.cache_or_load_file**

`chainer.dataset.cache_or_load_file(path, creator, loader)`  
Caches a file if it does not exist, or loads it otherwise.
This is a utility function used in dataset loading routines. The creator creates the file to given path, and returns the content. If the file already exists, the loader is called instead, and it loads the file and returns the content.

Note that the path passed to the creator is temporary one, and not same as the path given to this function. This function safely renames the file created by the creator to a given path, even if this function is called simultaneously by multiple threads or processes.

**Parameters**

- **path (str)** – Path to save the cached file.
- **creator** – Function to create the file and returns the content. It takes a path to temporary place as the argument. Before calling the creator, there is no file at the temporary path.
- **loader** – Function to load the cached file and returns the content.

**Returns** It returns the returned values by the creator or the loader.

### Training loop abstraction

Chainer provides a standard implementation of the training loops under the `chainer.training` module. It is built on top of many other core features of Chainer, including Variable and Function, Link/Chain/ChainList, Optimizer, Dataset, and Reporter/Summary. Compared to the training loop abstraction of other machine learning toolkits, Chainer’s training framework aims at maximal flexibility, while keeps the simplicity for the typical usages. Most components are pluggable, and users can overwrite the definition.

The core of the training loop abstraction is `Trainer`, which implements the training loop itself. The training loop consists of two parts: one is `Updater`, which actually updates the parameters to train, and the other is `Extension` for arbitrary functionalities other than the parameter update.

Updater and some extensions use `chainer.dataset` and `Iterator` to scan the datasets and load mini-batches. The trainer also uses `Reporter` to collect the observed values, and some extensions use `DictSummary` to accumulate them and computes the statistics.

You can find many examples for the usage of this training utilities from the official examples. You can also search the extension implementations from `Trainer extensions`.

### Trainer

**`chainer.training.Trainer`**

The standard training loop in Chainer.

**`chainer.training.Trainer`**

```python
class chainer.training.Trainer(updater, stop_trigger=None, out='result')
```

The standard training loop in Chainer.

Trainer is an implementation of a training loop. Users can invoke the training by calling the `run()` method.

Each iteration of the training loop proceeds as follows.

- Update of the parameters. It includes the mini-batch loading, forward and backward computations, and an execution of the update formula. These are all done by the update object held by the trainer.
- Invocation of trainer extensions in the descending order of their priorities. A trigger object is attached to each extension, and it decides at each iteration whether the extension should be executed. Trigger objects
are callable objects that take the trainer object as the argument and return a boolean value indicating whether the extension should be called or not.

Extensions are callable objects that take the trainer object as the argument. There are three ways to define custom extensions: inheriting the `Extension` class, decorating functions by `make_extension()`, and defining any callable including lambda functions. See `Extension` for more details on custom extensions and how to configure them.

Users can register extensions to the trainer by calling the `extend()` method, where some configurations can be added.

- Trigger object, which is also explained above. In most cases, `IntervalTrigger` is used, in which case users can simply specify a tuple of the interval length and its unit, like `(1000, 'iteration')` or `(1, 'epoch')`.

- The order of execution of extensions is determined by their priorities. Extensions of higher priorities are invoked earlier. There are three standard values for the priorities:
  - `PRIORITY_WRITER`. This is the priority for extensions that write some records to the `observation` dictionary. It includes cases that the extension directly adds values to the observation dictionary, or the extension uses the `chainer.report()` function to report values to the observation dictionary.
  - `PRIORITY_EDITOR`. This is the priority for extensions that edit the `observation` dictionary based on already reported values.
  - `PRIORITY_READER`. This is the priority for extensions that only read records from the `observation` dictionary. This is also suitable for extensions that do not use the `observation` dictionary at all.

- Extensions with `invoke_before_training` flag on are also invoked at the beginning of the training loop. Extensions that update the training status (e.g., changing learning rates) should have this flag to be `True` to ensure that resume of the training loop correctly recovers the training status.

The current state of the trainer object and objects handled by the trainer can be serialized through the standard serialization protocol of Chainer. It enables us to easily suspend and resume the training loop.

**Note:** The serialization does not recover everything of the training loop. It only recovers the states which change over the training (e.g. parameters, optimizer states, the batch iterator state, extension states, etc.). You must initialize the objects correctly before deserializing the states.

On the other hand, it means that users can change the settings on deserialization. For example, the exit condition can be changed on the deserialization, so users can train the model for some iterations, suspend it, and then resume it with larger number of total iterations.

During the training, it also creates a `Reporter` object to store observed values on each update. For each iteration, it creates a fresh observation dictionary and stores it in the `observation` attribute.

Links of the target model of each optimizer are registered to the reporter object as observers, where the name of each observer is constructed as the format `<optimizer name><link name>`. The link name is given by the `chainer.Link.namedlink()` method, which represents the path to each link in the hierarchy. Other observers can be registered by accessing the reporter object via the `reporter` attribute.

The default trainer is `plain`, i.e., it does not contain any extensions.

**Parameters**

- `updater` (`Updater`) – Updater object. It defines how to update the models.
Variables

• **updater** – The updater object for this trainer.
• **stop_trigger** – Trigger that determines when to stop the training loop. The training loop stops at the iteration on which this trigger returns True.
• **observation** – Observation of values made at the last update. See the Reporter class for details.
• **out** – Output directory.
• **reporter** – Reporter object to report observed values.

Methods

**extend**(extension, name=None, trigger=None, priority=None, invoke_before_training=None)

Registers an extension to the trainer.

*Extension* is a callable object which is called after each update unless the corresponding trigger object decides to skip the iteration. The order of execution is determined by priorities: extensions with higher priorities are called earlier in each iteration. Extensions with the same priority are invoked in the order of registrations.

If two or more extensions with the same name are registered, suffixes are added to the names of the second to last extensions. The suffix is _N where N is the ordinal of the extensions.

See *Extension* for the interface of extensions.

**Parameters**

• **extension** – Extension to register.
• **name** (*str*) – Name of the extension. If it is omitted, the default_name attribute of the extension is used instead. Note that the name would be suffixed by an ordinal in case of duplicated names as explained above.
• **trigger** (*tuple or Trigger*) – Trigger object that determines when to invoke the extension. If it is None, extension.trigger is used instead. If it is None and the extension does not have the trigger attribute, the extension is triggered at every iteration by default. If the trigger is not callable, it is passed to IntervalTrigger to build an interval trigger.
• **priority** (*int*) – Invocation priority of the extension. Extensions are invoked in the descending order of priorities in each iteration. If this is None, extension.priority is used instead.
• **invoke_before_training** (*bool or None*) – If True, the extension is also invoked just before entering the training loop. If this is None, extension.invoke_before_training is used instead. This option is mainly used for extensions that alter the training configuration (e.g., learning rates); in such a case, resuming from snapshots require the call of extension to recover the configuration before any updates.

**get_extension**(name)

Returns the extension of a given name.

**Parameters**

• **name** (*str*) – Name of the extension.
Returns Extension.

run() Executes the training loop.
      This method is the core of Trainer. It executes the whole loop of training the models.
      Note that this method cannot run multiple times for one trainer object.

serialize(serializer)

Attributes

elapsed_time Total time used for the training.
      The time is in seconds. If the training is resumed from snapshot, it includes the time of all the previous
      training to get the current state of the trainer.

Updater

<table>
<thead>
<tr>
<th>chainer.trainingUpdater</th>
<th>Interface of updater objects for trainers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.training.StandardUpdater</td>
<td>Standard implementation of Updater.</td>
</tr>
<tr>
<td>chainer.training.ParallelUpdater</td>
<td>Implementation of a parallel GPU Updater.</td>
</tr>
</tbody>
</table>

chainer.training.Updater
class chainer.training.Updater
      Interface of updater objects for trainers.

Updater implements a training iteration as update().
      Typically, the updating iteration proceeds as follows.
      • Fetch a minibatch from :module:`chainer.dataset` via Iterator.
      • Run forward and backward process of Chain.
      • Update parameters according to their UpdateRule.
      The first line is processed by chainer.dataset.Iterator.next(). The second and third are
      processed by update().
      Users can also implement original update() by overiding it.

Methods

connect_trainer(trainer)
      Connects the updater to the trainer that will call it.

      The typical usage of this method is to register additional links to the reporter of the trainer. This method is
      called at the end of the initialization of Trainer. The default implementation does nothing.

      Parameters trainer (Trainer) – Trainer object to which the updater is registered.
finalize()
Finalizes the updater object.
This method is called at the end of training loops. It should finalize each dataset iterator used in this updater.

get_all_optimizers()
Gets a dictionary of all optimizers for this updater.

Returns Dictionary that maps names to optimizers.
Return type dict
get_optimizer(name)
Gets the optimizer of given name.
Updater holds one or more optimizers with names. They can be retrieved by this method.

Parameters name (str) – Name of the optimizer.

Returns Optimizer of the name.
Return type Optimizer
serialize (serializer)
Serializes the current state of the updater object.
update()
Updates the parameters of the target model.
This method implements an update formula for the training task, including data loading, forward/backward computations, and actual updates of parameters.
This method is called once at each iteration of the training loop.

crainer.training.StandardUpdater

class crainer.training.StandardUpdater (iterator, optimizer, converter=<function concat_examples>, device=None, loss_func=None)
Standard implementation of Updater.

This is the standard implementation of Updater. It accepts one or more training datasets and one or more optimizers. The default update routine assumes that there is only one training dataset and one optimizer. Users can override this update routine by inheriting this class and overriding the update_core() method. Each batch is converted to input arrays by concat_examples() by default, which can also be manually set by converter argument.

Parameters

- iterator – Dataset iterator for the training dataset. It can also be a dictionary that maps strings to iterators. If this is just an iterator, then the iterator is registered by the name 'main'.
- optimizer – Optimizer to update parameters. It can also be a dictionary that maps strings to optimizers. If this is just an optimizer, then the optimizer is registered by the name 'main'.
- converter – Converter function to build input arrays. Each batch extracted by the main iterator and the device option are passed to this function. concat_examples() is used by default.
- device – Device to which the training data is sent. Negative value indicates the host memory (CPU).
Variables

- **converter** – Converter function.
- **loss_func** – Loss function. If it is `None`, the target link of the main optimizer is used instead.
- **device** – Device to which the training data is sent.
- **iteration** – Current number of completed updates.

Methods

**connect_trainer** (*trainer*)

Connects the updater to the trainer that will call it.

The typical usage of this method is to register additional links to the reporter of the trainer. This method is called at the end of the initialization of `Trainer`. The default implementation does nothing.

Parameters

- **trainer** (*Trainer*) – Trainer object to which the updater is registered.

**finalize** ()

Finalizes the updater object.

This method calls the `finalize` method of each iterator that this updater has. It is called at the end of training loops.

**get_all_optimizers** ()

Gets a dictionary of all optimizers for this updater.

Returns

- **Dictionary that maps names to optimizers.**

Return type

- **dict**

**get_iterator** (*name*)

Gets the dataset iterator of given name.

Parameters

- **name** (*str*) – Name of the dataset iterator.

Returns

- **Corresponding dataset iterator.**

Return type

- **Iterator**

**get_optimizer** (*name*)

Gets the optimizer of given name.

Parameters

- **name** (*str*) – Name of the optimizer.

Returns

- **Corresponding optimizer.**

Return type

- **Optimizer**

**serialize** (*serializer*)

Serializes the current state of the updater object.

**update** ()

Updates the parameters of the target model.

This method implements an update formula for the training task, including data loading, forward/backward computations, and actual updates of parameters.

This method is called once at each iteration of the training loop.
update_core()

Attributes

epoch
epoch_detail
is_new_epoch
previous_epoch_detail

chainer.training.ParallelUpdater

class chainer.training.ParallelUpdater(iterator, optimizer, converter=<function concat_examples>, models=None, devices=None, loss_func=None)

Implementation of a parallel GPU Updater.

This is an implementation of Updater that uses multiple GPUs. It behaves similarly to StandardUpdater. The update routine is modified to support data-parallel computation on multiple GPUs in one machine. It is based on synchronous parallel SGD: it parallelizes the gradient computation over a mini-batch, and updates the parameters only in the main device.

Parameters

• iterator – Dataset iterator for the training dataset. It can also be a dictionary that maps strings to iterators. If this is just an iterator, then the iterator is registered by the name 'main'.

• optimizer – Optimizer to update parameters. It can also be a dictionary that maps strings to optimizers. If this is just an optimizer, then the optimizer is registered by the name 'main'.

• converter – Converter function to build input arrays. Each batch extracted by the main iterator is split equally between the devices and then passed with corresponding device option to this function. concat_examples() is used by default.

• models – Dictionary of models. The main model should be the same model attached to the 'main' optimizer.

• devices – Dictionary of devices to which the training data is sent. The devices should be arranged in a dictionary with the same structure as models.

• loss_func – Loss function. The model is used as a loss function by default.

Methods

connect_trainer(trainer)

finalize()

Finalizes the updater object.

This method calls the finalize method of each iterator that this updater has. It is called at the end of training loops.

get_all_optimizers()

Gets a dictionary of all optimizers for this updater.
Returns Dictionary that maps names to optimizers.

Return type dict

def get_iterator(name)
    Gets the dataset iterator of given name.

    Parameters name (str) – Name of the dataset iterator.

    Returns Corresponding dataset iterator.

    Return type Iterator

def get_optimizer(name)
    Gets the optimizer of given name.

    Parameters name (str) – Name of the optimizer.

    Returns Corresponding optimizer.

    Return type Optimizer

def serialize(serializer)
    Serializes the current state of the updater object.

def update()
    Updates the parameters of the target model.

    This method implements an update formula for the training task, including data loading, forward/backward
    computations, and actual updates of parameters.

    This method is called once at each iteration of the training loop.

def update_core()

Attributes

epoch
epoch_detail
is_new_epoch
previous_epoch_detail

Implementation of a multiprocess parallel GPU Updater.

This is an implementation of Updater that uses multiple GPUs with multi-process data parallelism. It uses
Nvidia NCCL for communication between multiple GPUs.

It behaves similarly to StandardUpdater. The update routine is modified to support data-parallel compu-
tation on multiple GPUs in one machine. It is based on synchronous parallel SGD: it parallelizes the gradient
computation over a mini-batch, and updates the parameters only in the main device.

It does not transfer the values collected by Reporter in the sub devices to the main device. So you can only
see the reported values in the main device.
Parameters

- **iterators** – List of dataset iterator for the training dataset. The number of the iterators must be same to the number of GPUs you use.
- **optimizer** – Optimizer to update parameters. The model should be attached to the optimizer.
- **converter** – Converter function to build input arrays. Each batch extracted by the iterator is split equally between the devices and then passed with corresponding device option to this function. `concat_examples()` is used by default.
- **devices** – Dictionary or list of devices to which the training data is sent. The master device will be the first one in the list or the value attached to the key 'main'.

Methods

**static available()**

**connect_trainer**(trainer)

Connects the updater to the trainer that will call it.

The typical usage of this method is to register additional links to the reporter of the trainer. This method is called at the end of the initialization of Trainer. The default implementation does nothing.

**Parameters**

`trainer (Trainer) – Trainer object to which the updater is registered.`

**finalize()**

**get_all_optimizers()**

Gets a dictionary of all optimizers for this updater.

**Returns** Dictionary that maps names to optimizers.

**Return type** `dict`

**get_iterator**(name)

Gets the dataset iterator of given name.

**Parameters**

`name (str) – Name of the dataset iterator.`

**Returns** Corresponding dataset iterator.

**Return type** `Iterator`

**get_optimizer**(name)

Gets the optimizer of given name.

**Parameters**

`name (str) – Name of the optimizer.`

**Returns** Corresponding optimizer.

**Return type** `Optimizer`

**serialize**(serializer)

Serializes the current state of the updater object.

**setup_workers()**

**update()**

Updates the parameters of the target model.

This method implements an update formula for the training task, including data loading, forward/backward computations, and actual updates of parameters.
This method is called once at each iteration of the training loop.

\texttt{update\_core()}

**Attributes**

- \texttt{epoch}
- \texttt{epoch\_detail}
- \texttt{is\_new\_epoch}
- \texttt{previous\_epoch\_detail}

**Extension**

- \texttt{chainer.training.Extension} [Base class of trainer extensions.]
- \texttt{chainer.training.make_extension} [Decorator to make given functions into trainer extensions.]

**chainer.training.Extension**

**class** \texttt{chainer.training.Extension} [Base class of trainer extensions.]

Extension of \texttt{Trainer} is a callable object that takes the trainer object as the argument. It also provides some default configurations as its attributes, e.g. the default trigger and the default priority. This class provides a set of typical default values for these attributes.

There are three ways to define users’ own extensions: inheriting this class, decorating closures by \texttt{make\_extension()}, or using any callable including lambda functions as extensions. Decorator can slightly reduce the overhead and is much easier to use, while this class provides more flexibility (for example, it can have methods to configure the behavior). Using a lambda function allows one-line coding for simple purposes, but users have to specify the configurations as arguments to \texttt{Trainer.extend()}. For a callable not inheriting this class, the default configurations of this class are used unless the user explicitly specifies them in \texttt{Trainer.extend()} method.

**Variables**

- \texttt{trigger} – Default value of trigger for this extension. It is set to \texttt{(1, 'iteration')} by default.
- \texttt{priority} – Default priority of the extension. It is set to \texttt{PRIORITY_READER} by default.

**Methods**

\texttt{__call__} (\texttt{trainer})

Invokes the extension.

Implementations should override this operator. This method is called at iterations which the corresponding trigger accepts.

**Parameters**

- \texttt{trainer} (\texttt{Trainer}) – Trainer object that calls this operator.

\texttt{finalize} ()

Finalizes the extension.
This method is called at the end of the training loop.

**initialize** *(trainer)*

Initialize up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of `Trainer` can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, `ExponentialShift` extension changes the optimizer’s hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The `ExponentialShift` extension recovers it in its `initialize` method if it has been loaded from a snapshot, or just setting the initial value otherwise.

**Parameters**

- **trainer** *(Trainer)* – Trainer object that runs the training loop.

**serialize** *(serializer)*

Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

**Attributes**

- **default_name**
  Default name of the extension.
  
  It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

- **priority** *(int)* = 100

- **trigger** *(1, ‘iteration’)*

**chainer.training.make_extension**

`chainer.training.make_extension(trigger=None, default_name=None, priority=None, finalizer=None, initializer=None, **kwargs)`

Decorator to make given functions into trainer extensions.

This decorator just adds some attributes to a given function. The value of the attributes are given by the arguments of this decorator.

See [*Extension*](#) for details of trainer extensions. Most of the default values of arguments also follow those for this class.

**Parameters**

- **trigger** – Default trigger of the extension.
  
- **default_name** – Default name of the extension. The name of a given function is used by default.
  
- **priority** *(int)* – Default priority of the extension.
  
- **finalizer** – Finalizer function of this extension. It is called at the end of the training loop.
  
- **initializer** – Initializer function of this extension. It is called at the beginning of the training loop.
Trigger

Trigger is a callable object to decide when to process some specific event within the training loop. It takes a Trainer object as the argument, and returns True if some event should be fired.

It is mainly used to determine when to call an extension. It is also used to determine when to quit the training loop.

chainer.training.get_trigger

chainer.training.get_trigger(trigger)

Gets a trigger object.

Trigger object is a callable that accepts a Trainer object as an argument and returns a boolean value. When it returns True, various kinds of events can occur depending on the context in which the trigger is used. For example, if the trigger is passed to the Trainer as the stop trigger, the training loop breaks when the trigger returns True. If the trigger is passed to the extend() method of a trainer, then the registered extension is invoked only when the trigger returns True.

This function returns a trigger object based on the argument. If trigger is already a callable, it just returns the trigger. If trigger is None, it returns a trigger that never fires. Otherwise, it passes the value to IntervalTrigger.

Parameters trigger – Trigger object. It can be either an already built trigger object (i.e., a callable object that accepts a trainer object and returns a bool value), or a tuple. In latter case, the tuple is passed to IntervalTrigger.

Returns trigger if it is a callable, otherwise a IntervalTrigger object made from trigger.

Debug mode

In debug mode, Chainer checks values of variables on runtime and shows more detailed error messages. It helps you to debug your programs. Instead it requires additional overhead time.

In debug mode, Chainer checks all results of forward and backward computation, and if it founds a NaN value, it raises RuntimeError. Some functions and links also check validity of input values.

As of v2.0.0, it is recommended to turn on the debug mode using chainer.config.debug. See Configuring Chainer for the way to use the config object. We leave the reference of the conventional ways (which have been available since Chainer v1) as follows.

chainer.is_debug

chainer.is_debug()

Get the debug mode.

Returns Return True if Chainer is in debug mode.

Return type bool
**chainer.set_debug**

chainer.set_debug(debug)
Set the debug mode.

**Note:** This method changes global state. When you use this method on multi-threading environment, it may affects other threads.

**Parameters**

- **debug** *(bool)* – New debug mode.

**chainer.DebugMode**

class chainer.DebugMode(debug)
Debug mode context.

This class provides a context manager for debug mode. When entering the context, it sets the debug mode to the value of `debug` parameter with memorizing its original value. When exiting the context, it sets the debug mode back to the original value.

Deprecated since version v2.0.0: DebugMode is deprecated. Use `using_config('debug', debug)` instead.

**Parameters**

- **debug** *(bool)* – Debug mode used in the context.

**Methods**

- `__enter__()`
- `__exit__(*args)`

**Configuring Chainer**

Chainer provides some global settings that affect the behavior of some functionalities. Such settings can be configured using the unified configuration system. The system provides a transparent way to manage the configuration for each process and for each thread.

The configuration is managed by two global objects: `chainer.global_config` and `chainer.config`.

- The `global_config` object maintains the configuration shared in the Python process. This is an instance of the GlobalConfig class. It can be used just as a plain object, and users can freely set any attributes on it.

- The `config` object, on the other hand, maintains the configuration for the current thread. This is an instance of the LocalConfig class. It behaves like a thread-local object, and any attribute modifications are only visible to the current thread.

If no value is set to `config` for a given key, `global_config` is transparently referred. Thanks to this transparent lookup, users can always use `config` to read any configuration so that the thread-local configuration is used if available and otherwise the default global setting is used.

The following entries of the configuration are currently provided by Chainer. Some entries support environment variables to set the default values. Note that the default values are set in the global config.
**chainer.config.cudnn_deterministic** Flag to configure deterministic computations in cuDNN APIs. If it is True, convolution functions that use cuDNN use the deterministic mode (i.e., the computation is reproducible). Otherwise, the results of convolution functions using cuDNN may be non-deterministic in exchange for the performance. The default value is False.

**chainer.config.debug** Debug mode flag. If it is True, Chainer runs in the debug mode. See Debug mode for more information of the debug mode. The default value is given by CHAINER_DEBUG environment variable (set to 0 or 1) if available, otherwise uses False.

**chainer.config.enable_backprop** Flag to enable backpropagation support. If it is True, Function makes a computational graph of Variable for back-propagation. Otherwise, it does not make a computational graph. So a user cannot call backward() method to results of the function. The default value is True.

**chainer.config.keep_graph_on_report** Flag to configure whether or not to let report() keep the computational graph. If it is False, report() does not keep the computational graph when a Variable object is reported. It means that report() stores a copy of the Variable object which is purged from the computational graph. If it is True, report() just stores the Variable object as is with the computational graph left attached. The default value is False.

**chainer.config.train** Training mode flag. If it is True, Chainer runs in the training mode. Otherwise, it runs in the testing (evaluation) mode. The default value is True.

**chainer.config.type_check** Type checking mode flag. If it is True, Chainer checks the types (data types and shapes) of inputs on Function applications. Otherwise, it skips type checking. The default value is given by CHAINER_TYPE_CHECK environment variable (set to 0 or 1) if available, otherwise uses True.

**chainer.config.use_cudnn** Flag to configure whether or not to use cuDNN. This is a tertiary flag with 'always', 'auto', and 'never' as its allowed values. The meaning of each flag is as follows.

- If it is 'always', Chainer will try to use cuDNN everywhere if possible.
- If it is 'auto', Chainer will use cuDNN only if it is known that the usage does not degrade the performance.
- If it is 'never', Chainer will never use cuDNN anywhere.

The default value is 'auto'.

Users can also define their own configurations. There are two ways:

1. Use Chainer’s configuration objects. In this case, it is strongly recommended to prefix the name by “user_” to avoid name conflicts with configurations introduced to Chainer in the future.

2. Use your own configuration objects. Users can define their own configuration objects using chainer.configuration.GlobalConfig and chainer.configuration.LocalConfig. In this case, there is no need to take care of the name conflicts.

### Example

If you want to share a setting within the process, set an attribute to the global configuration.

```python
>>> chainer.global_config.user_my_setting = 123
```

This value is automatically extracted by referring to the local config.

```python
>>> chainer.config.user_my_setting
123
```

If you set an attribute to the local configuration, the value is only visible to the current thread.

---

3.1. Core functionalities
We often want to temporarily modify the configuration for the current thread. It can be done by using `using_config()`. For example, if you only want to enable debug mode in a fragment of code, write as follows.

```python
>>> with chainer.using_config('debug', True):
...     pass  # code running in the debug mode
```

We often want to switch to the test mode for an evaluation. This is also done in the same way.

```python
>>> with chainer.using_config('train', False):
...     pass  # code running in the test mode
```

Note that `Evaluator` automatically switches to the test mode, and thus you do not need to manually switch in the loss function for the evaluation.

You can also make your own code behave differently in training and test modes as follows.

```python
if chainer.config.train:
    pass  # code only running in the training mode
else:
    pass  # code only running in the test mode
```

---

<table>
<thead>
<tr>
<th><code>chainer.global_config</code></th>
<th>The plain object that represents the global configuration of Chainer.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.config</code></td>
<td>Thread-local configuration of Chainer.</td>
</tr>
<tr>
<td><code>chainer.using_config</code></td>
<td>Context manager to temporarily change the thread-local configuration.</td>
</tr>
<tr>
<td><code>chainer.configuration.GlobalConfig</code></td>
<td>The plain object that represents the global configuration of Chainer.</td>
</tr>
<tr>
<td><code>chainer.configuration.LocalConfig</code></td>
<td>Thread-local configuration of Chainer.</td>
</tr>
</tbody>
</table>

---

**chainer.global_config**

`chainer.global_config = <chainer.configuration.GlobalConfig object>`

The plain object that represents the global configuration of Chainer.

**chainer.config**

`chainer.config = <chainer.configuration.LocalConfig object>`

Thread-local configuration of Chainer.

This class implements the local configuration. When a value is set to this object, the configuration is only updated in the current thread. When a user tries to access an attribute and there is no local value, it automatically retrieves a value from the global configuration.

**chainer.using_config**

`chainer.using_config(name, value, config=chainer.config)`

Context manager to temporarily change the thread-local configuration.

**Parameters**
• **name** (**str**) – Name of the configuration to change.
• **value** – Temporary value of the configuration entry.
• **config** (**LocalConfig**) – Configuration object. Chainer’s thread-local configuration is used by default.

**chainer.configuration.GlobalConfig**

```python
class chainer.configuration.GlobalConfig
    The plain object that represents the global configuration of Chainer.
```

**Methods**

```python
show (file=sys.stdout)
    Prints the global config entries.

    The entries are sorted in the lexicographical order of the entry name.

    **Parameters** **file** – Output file-like object.
```

**chainer.configuration.LocalConfig**

```python
class chainer.configuration.LocalConfig (global_config)
    Thread-local configuration of Chainer.

    This class implements the local configuration. When a value is set to this object, the configuration is only updated in the current thread. When a user tries to access an attribute and there is no local value, it automatically retrieves a value from the global configuration.
```

**Methods**

```python
show (file=sys.stdout)
    Prints the config entries.

    The entries are sorted in the lexicographical order of the entry names.

    **Parameters** **file** – Output file-like object.
```

**Example**

You can easily print the list of configurations used in the current thread.

```bash
>>> chainer.config.show()
depth: False
depth: True
depth: False
```

---

3.1. Core functionalities
Utilities

CUDA utilities

Device, context and memory management on CuPy.

Chainer uses CuPy (with very thin wrapper) to exploit the speed of GPU computation. Following modules and classes defined in CuPy are imported to chainer.cuda module for convenience (refer to this table when reading chainer’s source codes).

<table>
<thead>
<tr>
<th>imported name</th>
<th>original name</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.cuda.cupy</td>
<td>cupy</td>
</tr>
<tr>
<td>chainer.cuda.ndarray</td>
<td>cupy.ndarray</td>
</tr>
<tr>
<td>chainer.cuda.cupy.cuda</td>
<td>cupy.cuda</td>
</tr>
<tr>
<td>chainer.cuda.Device</td>
<td>cupy.cuda.Device</td>
</tr>
<tr>
<td>chainer.cuda.Event</td>
<td>cupy.cuda.Event</td>
</tr>
<tr>
<td>chainer.cuda.Stream</td>
<td>cupy.cuda.Stream</td>
</tr>
</tbody>
</table>

Chainer replaces the default allocator of CuPy by its memory pool implementation. It enables us to reuse the device memory over multiple forward/backward computations, and temporary arrays for consecutive elementwise operations.

Devices

<table>
<thead>
<tr>
<th>chainer.cuda.get_device</th>
<th>Gets the device from a device object, an ID integer or an array object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.cuda.get_device_from_id</td>
<td>Gets the device from an ID integer.</td>
</tr>
<tr>
<td>chainer.cuda.get_device_from_array</td>
<td>Gets the device from a list of CuPy array or a single CuPy array.</td>
</tr>
</tbody>
</table>

chainer.cuda.get_device

chainer.cuda.get_device (*args)

Gets the device from a device object, an ID integer or an array object.

**Note:** This API is deprecated. Please use get_device_from_id() or get_device_from_array() instead.

This is a convenient utility to select a correct device if the type of arg is unknown (i.e., one can use this function on arrays that may be on CPU or GPU). The returned device object supports the context management protocol of Python for the with statement.

**Parameters** args – Values to specify a GPU device. The first device object, integer or cupy.ndarray object is used to select a device. If it is a device object, it is returned. If it is an integer, the corresponding device is returned. If it is a CuPy array, the device on which this array reside is returned. If any arguments are neither integers nor CuPy arrays, a dummy device object representing CPU is returned.

**Returns** Device object specified by given args.

**See also:**
See cupy.cuda.Device for the device selection not by arrays.
chainer.cuda.get_device_from_id

chainer.cuda.get_device_from_id(device_id)
Gets the device from an ID integer.

Parameters device_id (int or None) – The ID of the device which this function returns.

chainer.cuda.get_device_from_array

chainer.cuda.get_device_from_array(*arrays)
Gets the device from a list of CuPy array or a single CuPy array.

The device on which the given CuPy array reside is returned.

Parameters array (cupy.ndarray or list of cupy.ndarray) – A CuPy array which this function returns the device corresponding to. If a list of cupy.ndarray s are given, it returns the first device object of an array in the list.

CuPy array allocation and copy

<table>
<thead>
<tr>
<th>chainer.cuda.copy</th>
<th>Copies a cupy.ndarray object using the default stream.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.cuda.to_cpu</td>
<td>Copies the given GPU array to host CPU.</td>
</tr>
<tr>
<td>chainer.cuda.to_gpu</td>
<td>Copies the given CPU array to specified device.</td>
</tr>
</tbody>
</table>

chainer.cuda.copy

chainer.cuda.copy(array, out=None, out_device=None, stream=None)
Copies a cupy.ndarray object using the default stream.

This function can copy the device array to the destination array on another device.

Parameters

- array (cupy.ndarray) – Array to be copied.
- out (cupy.ndarray) – Destination array. If it is not None, then out_device argument is ignored.
- out_device – Destination device specifier. Actual device object is obtained by passing this value to get_device().
- stream (cupy.cuda.Stream) – CUDA stream.

Returns

Copied array.

If out is not specified, then the array is allocated on the device specified by out_device argument.

Return type cupy.ndarray

chainer.cuda.to_cpu

chainer.cuda.to_cpu(array, stream=None)
Copies the given GPU array to host CPU.
Parameters

- **array** – Array to be sent to CPU.
- **stream** (cupy.cuda.Stream) – CUDA stream.

Returns

Array on CPU.

If given `array` is already on CPU, then this function just returns `array` without performing any copy.

Return type `numpy.ndarray`

chainer.cuda.to_gpu

**chainer.cuda.to_gpu** *(array, device=None, stream=None)*

Copies the given CPU array to specified device.

Parameters

- **array** – Array to be sent to GPU.
- **device** – Device specifier.
- **stream** (cupy.cuda.Stream) – CUDA stream. If not None, the copy runs asynchronously.

Returns

Array on GPU.

If `array` is already on GPU, then this function just returns `array` without performing any copy. Note that this function does not copy `cupy.ndarray` into specified device.

Return type `cupy.ndarray`

Kernel definition utilities

<table>
<thead>
<tr>
<th>chainer.cuda.memoize</th>
<th>Makes a function memoizing the result for each argument and device.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.cuda.clear_memo</td>
<td>Clears the memoized results for all functions decorated by memoize.</td>
</tr>
<tr>
<td>chainer.cuda.elementwise</td>
<td>Creates an elementwise kernel function.</td>
</tr>
<tr>
<td>chainer.cuda.reduce</td>
<td>Creates a global reduction kernel function.</td>
</tr>
</tbody>
</table>

chainer.cuda.memoize

**chainer.cuda.memoize** *(for_each_device=False)*

Makes a function memoizing the result for each argument and device.

This is a similar version of `cupy.memoize()`. The difference is that this function can be used in the global scope even if CUDA is not available. In such case, this function does nothing.

**Note:** This decorator acts as a dummy if CUDA is not available. It cannot be used for general purpose memoization even if `for_each_device` is set to False.
chainer.cuda.clear_memo

chainer.cuda.clear_memo()
Clears the memoized results for all functions decorated by memoize.
This function works like cupy.clear_memo() as a counterpart for chainer.cuda.memoize(). It can be used even if CUDA is not available. In such a case, this function does nothing.

chainer.cuda.elementwise

chainer.cuda.elementwise(*args, **kwargs)
Creates an elementwise kernel function.

This function uses memoize() to cache the kernel object, i.e. the resulting kernel object is cached for each argument combination and CUDA device.
The arguments are the same as those for cupy.ElementwiseKernel, except that the name argument is mandatory.

chainer.cuda.reduce

chainer.cuda.reduce(*args, **kwargs)
Creates a global reduction kernel function.

This function uses memoize() to cache the resulting kernel object, i.e. the resulting kernel object is cached for each argument combination and CUDA device.
The arguments are the same as those for cupy.ReductionKernel, except that the name argument is mandatory.

CPU/GPU generic code support

chainer.cuda.get_array_module

chainer.cuda.get_array_module(*args)
Gets an appropriate one from numpy or cupy.

This is almost equivalent to cupy.get_array_module(). The differences are that this function can be used even if CUDA is not available and that it will return their data arrays’ array module for Variable arguments.

Parameters
args – Values to determine whether NumPy or CuPy should be used.

Returns
cupy or numpy is returned based on the types of the arguments.

Return type
module

cuDNN support

chainer.cuda.set_max_workspace_size

chainer.cuda.set_max_workspace_size() Sets the workspace size for cuDNN.

chainer.cuda.get_max_workspace_size

chainer.cuda.get_max_workspace_size() Gets the workspace size for cuDNN.
**chainer.cuda.set_max_workspace_size**

**chainer.cuda.set_max_workspace_size(size)**

Sets the workspace size for cuDNN.

Check “cuDNN Library User Guide” for detail.

**Parameters**

- **size** – The workspace size for cuDNN.

**chainer.cuda.get_max_workspace_size**

**chainer.cuda.get_max_workspace_size()**

Gets the workspace size for cuDNN.

Check “cuDNN Library User Guide” for detail.

**Returns**

The workspace size for cuDNN.

**Return type**

int

---

**Common algorithms**

**chainer.utils.WalkerAlias**

Implementation of Walker’s alias method.

**chainer.utils.WalkerAlias**

**class** chainer.utils.WalkerAlias(probs)

Implementation of Walker’s alias method.

This method generates a random sample from given probabilities \( p_1, \ldots, p_n \) in \( O(1) \) time. It is more efficient than choice(). This class works on both CPU and GPU.

**Parameters**

- **probs** *(float list)* – Probabilities of entries. They are normalized with \( \text{sum}(probs) \).

**See:** Wikipedia article

**Methods**

**sample**(shape)

Generates a random sample based on given probabilities.

**Parameters**

- **shape** *(tuple of int)* – Shape of a return value.

**Returns**

Returns a generated array with the given shape. If a sampler is in CPU mode the return value is a numpy.ndarray object, and if it is in GPU mode the return value is a cupy.ndarray object.

**sample_cpu**(shape)

**sample_gpu**(shape)

**to_cpu()**

Make a sampler CPU mode.
to_gpu()

Make a sampler GPU mode.

**Reporter**

**Reporter**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.Reporter</code></td>
<td>Object to which observed values are reported.</td>
</tr>
<tr>
<td><code>chainer.get_current_reporter</code></td>
<td>Returns the current reporter object.</td>
</tr>
<tr>
<td><code>chainer.report</code></td>
<td>Reports observed values with the current reporter object.</td>
</tr>
<tr>
<td><code>chainer.report_scope</code></td>
<td>Returns a report scope with the current reporter.</td>
</tr>
</tbody>
</table>

**chainer.Reporter**

```python
from chainer import Reporter, report, report_scope

reporter = Reporter()
observer = object()  # it can be an arbitrary (reference) object
reporter.add_observer('my_observer:', observer)
observation = {}
with reporter.scope(observation):
    reporter.report({'x': 1}, observer)

observation
{'my_observer:x': 1}
```

There are also a global API to add values:

```python
observation = {}
with report_scope(observation):
    report({'x': 1}, observer)

observation
{'my_observer:x': 1}
```

The most important application of Reporter is to report observed values from each link or chain in the training and validation procedures. *Trainer* and some extensions prepare their own Reporter object with the hierarchy of the target link registered as observers. We can use `report()` function inside any links and chains to report the observed values (e.g., training loss, accuracy, activation statistics, etc.).

**Variables**

- **observation** – Dictionary of observed values.
Methods

__enter__()
Makes this reporter object current.

__exit__(exc_type, exc_value, traceback)
Recovers the previous reporter object to the current.

add_observer(name, observer)
Registers an observer of values.
Observer defines a scope of names for observed values. Values observed with the observer are registered with names prefixed by the observer name.

Parameters

• name (str) – Name of the observer.
• observer – The observer object. Note that the reporter distinguishes the observers by their object ids (i.e., id(owner)), rather than the object equality.

add_observers(prefix, observers)
Registers multiple observers at once.
This is a convenient method to register multiple objects at once.

Parameters

• prefix (str) – Prefix of each name of observers.
• observers – Iterator of name and observer pairs.

report(values, observer=None)
Reports observed values.
The values are written with the key, prefixed by the name of the observer object if given.

Note: As of v2.0.0, if a value is of type Variable, the variable is copied without preserving the computational graph and the new variable purged from the graph is stored to the observer. This behavior can be changed by setting chainer.config.keep_graph_on_report to True.

Parameters

• values (dict) – Dictionary of observed values.
• observer – Observer object. Its object ID is used to retrieve the observer name, which is used as the prefix of the registration name of the observed value.

scope(*args, **kwds)
Creates a scope to report observed values to observation.
This is a context manager to be passed to with statements. In this scope, the observation dictionary is changed to the given one.
It also makes this reporter object current.

Parameters observation (dict) – Observation dictionary. All observations reported inside of the with statement are written to this dictionary.
**`chainer.get_current_reporter`**

```
chainer.get_current_reporter()
```

Returns the current reporter object.

**`chainer.report`**

```
chainer.report(values, observer=None)
```

Reports observed values with the current reporter object.

Any reporter object can be set current by the `with` statement. This function calls the `Report.report()` method of the current reporter. If no reporter object is current, this function does nothing.

**Example**

The most typical example is a use within links and chains. Suppose that a link is registered to the current reporter as an observer (for example, the target link of the optimizer is automatically registered to the reporter of the `Trainer`). We can report some values from the link as follows:

```python
class MyRegressor(chainer.Chain):
    def __init__(self, predictor):
        super(MyRegressor, self).__init__(predictor=predictor)
    def __call__(self, x, y):
        # This chain just computes the mean absolute and squared
        # errors between the prediction and y.
        pred = self.predictor(x)
        abs_error = F.sum(F.abs(pred - y)) / len(x.data)
        loss = F.mean_squared_error(pred, y)

        # Report the mean absolute and squared errors.
        report({'abs_error': abs_error, 'squared_error': loss}, self)
        return loss
```

If the link is named 'main' in the hierarchy (which is the default name of the target link in the `StandardUpdater`), these reported values are named 'main/abs_error' and 'main/squared_error'. If these values are reported inside the `Evaluator` extension, 'validation/' is added at the head of the link name, thus the item names are changed to 'validation/main/abs_error' and 'validation/main/squared_error' ('validation' is the default name of the `Evaluator` extension).

**Parameters**

- `values (dict)` – Dictionary of observed values.
- `observer` – Observer object. Its object ID is used to retrieve the observer name, which is used as the prefix of the registration name of the observed value.

**`chainer.report_scope`**

```
chainer.report_scope(*args, **kwds)
```

Returns a report scope with the current reporter.
This is equivalent to `get_current_reporter().scope(observation)`, except that it does not make the reporter current redundantly.

**Summary and DictSummary**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.Summary</code></td>
<td>Online summarization of a sequence of scalars.</td>
</tr>
<tr>
<td><code>chainer.DictSummary</code></td>
<td>Online summarization of a sequence of dictionaries.</td>
</tr>
</tbody>
</table>

**chainer.Summary**

class `chainer.Summary`

Online summarization of a sequence of scalars.

Summary computes the statistics of given scalars online.

**Methods**

- `add(value)`
  Adds a scalar value.
  
  **Parameters**
  
  value (scalar value to accumulate). It is either a NumPy scalar or a zero-dimensional array (on CPU or GPU).

- `compute_mean()`
  Computes the mean.

- `make_statistics()`
  Computes and returns the mean and standard deviation values.

  **Returns**
  Mean and standard deviation values.

  **Return type**
  tuple

**chainer.DictSummary**

class `chainer.DictSummary`

Online summarization of a sequence of dictionaries.

DictSummary computes the statistics of a given set of scalars online. It only computes the statistics for scalar values and variables of scalar values in the dictionaries.

**Methods**

- `add(d)`
  Adds a dictionary of scalars.

  **Parameters**
  
  d (dictionary of scalars to accumulate). Only elements of scalars, zero-dimensional arrays, and variables of zero-dimensional arrays are accumulated.

- `compute_mean()`
  Creates a dictionary of mean values.

  It returns a single dictionary that holds a mean value for each entry added to the summary.
Returns Dictionary of mean values.

Return type dict

make_statistics() Creates a dictionary of statistics.

It returns a single dictionary that holds mean and standard deviation values for every entry added to the summary. For an entry of name 'key', these values are added to the dictionary by names 'key' and 'key.std', respectively.

Returns Dictionary of statistics of all entries.

Return type dict

Experimental feature annotation

chainer.utils.experimental Declares that user is using an experimental feature.

chainer.utils.experimental (api_name) Declares that user is using an experimental feature.

The developer of an API can mark it as experimental by calling this function. When users call experimental APIs, FutureWarning is issued. The presentation of FutureWarning is disabled by setting chainer.disable_experimental_warning to True, which is False by default.

The basic usage is to call it in the function or method we want to mark as experimental along with the API name.

```
from chainer import utils
def f(x):
    utils.experimental('chainer.foo.bar.f')
    # concrete implementation of f follows
f(1)
```

... FutureWarning: chainer.foo.bar.f is experimental. The interface can change in the future. ...

We can also make a whole class experimental. In that case, we should call this function in its __init__ method.

```
class C():
    def __init__(self):
        utils.experimental('chainer.foo.C')
C()
```

... FutureWarning: chainer.foo.C is experimental. The interface can change in the future. ...

If we want to mark __init__ method only, rather than class itself, it is recommended that we explicitly feed its API name.
class D():
    def __init__(self):
        utils.experimental('D.__init__')

D()

... FutureWarning: D.__init__ is experimental. The interface can change in the
  future. ...

Currently, we do not have any sophisticated way to mark some usage of non-experimental function as experimental. But we can support such usage by explicitly branching it.

def g(x, experimental_arg=None):
    if experimental_arg is not None:
        utils.experimental('experimental_arg of chainer.foo.g')

Parameters api_name (str) – The name of an API marked as experimental.

## Assertion and Testing

Chainer provides some facilities to make debugging easy.

### Type checking utilities

Function uses a systematic type checking of the chainer.utils.type_check module. It enables users to easily find bugs of forward and backward implementations. You can find examples of type checking in some function implementations.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.utils.type_check.Expr</td>
<td>Abstract syntax tree of an expression.</td>
</tr>
<tr>
<td>chainer.utils.type_check.expect</td>
<td>Evaluates and tests all given expressions.</td>
</tr>
<tr>
<td>chainer.utils.type_check.TypeInfo</td>
<td>Type information of an input/gradient array.</td>
</tr>
<tr>
<td>chainer.utils.type_check.TypeInfoTuple</td>
<td>Type information of input/gradient tuples.</td>
</tr>
</tbody>
</table>

chainer.utils.type_check.Expr

class chainer.utils.type_check.Expr (priority)

Abstract syntax tree of an expression.

It represents an abstract syntax tree, and isn’t a value. You can get its actual value with eval() function, and get syntax representation with the __str__() method. Each comparison operator (e.g. ==) generates a new Expr object which represents the result of comparison between two expressions.

Example

Let x and y be instances of Expr, then

```python
>>> x = Variable(1, 'x')
>>> y = Variable(1, 'y')
>>> c = (x == y)
```

is also an instance of Expr. To evaluate and get its value, call eval() method:
Chainer Documentation, Release 2.0.2

>>> c.eval()
True

Call str function to get a representation of the original equation:
>>> str(c)
'x == y'

You can actually compare an expression with a value:
>>> (x == 1).eval()
True

Note that you can’t use boolean operators such as and, as they try to cast expressions to boolean values:
>>> z = Variable(1, 'z')
>>> x == y and y == z # raises an error
Traceback (most recent call last):
RuntimeError: Don't convert Expr to bool. Please call Expr.eval method to
˓→evaluate expression.

Methods
__call__(*args)
__getitem__(key)
eval()
Evaluates the tree to get actual value.
Behavior of this function depends on an implementation class. For example, a binary operator + calls the
__add__ function with the two results of eval() function.
__eq__(x, y)
__ne__(x, y)
__lt__(x, y)
__le__(x, y)
__gt__(x, y)
__ge__(x, y)
__nonzero__()
__bool__()
chainer.utils.type_check.expect
chainer.utils.type_check.expect(*bool_exprs)
Evaluates and tests all given expressions.
This function evaluates given boolean expressions in order. When at least one expression is evaluated as False,
that means the given condition is not satisfied. You can check conditions with this function.

3.3. Assertion and Testing

115


Parameters `bool_exprs` *(tuple of Bool expressions)* – Bool expressions you want to evaluate.

**chainer.utils.type_check.TypeInfo**

```python
class chainer.utils.type_check.TypeInfo(shape, dtype)
```

Type information of an input/gradient array. It contains type information of an array, such as the shape of array and the number of dimensions. This information is independent of CPU or GPU array.

**Methods**

**chainer.utils.type_check.TypeInfoTuple**

```python
class chainer.utils.type_check.TypeInfoTuple
```

Type information of input/gradient tuples. It is a sub-class of tuple containing `TypeInfo`. The `i`-th element of this object contains type information of the `i`-th input/gradient data. As each element is `Expr`, you can easily check its validity.

**Methods**

- `count(value)` → integer – return number of occurrences of value
- `index(value[, start[, stop]])` → integer – return first index of value. Raises ValueError if the value is not present.
- `size()`
  Returns an expression representing its length.

  Returns An expression object representing length of the tuple.

  Return type `Expr`

**Gradient checking utilities**

Most function implementations are numerically tested by gradient checking. This method computes numerical gradients of forward routines and compares their results with the corresponding backward routines. It enables us to make the source of issues clear when we hit an error of gradient computations. The `chainer.gradient_check` module makes it easy to implement the gradient checking.

| `chainer.gradient_check.check_backward` | Test backward procedure of a given function. |
| `chainer.gradient_check.numerical_grad` | Computes numerical gradient by finite differences. |

**chainer.gradient_check.check_backward**

```python
counterpart `chainer.gradient_check.check_backward`
```

`chainer.gradient_check.check_backward(func, x, data, y, grad, params=(), eps=0.001, atol=1e-05, rtol=0.0001, no_grads=None, dtype=None)`

Test backward procedure of a given function.

This function automatically check backward-process of given function. For example, when you have a `Function` class `MyFunc`, that gets two arguments and returns one value, you can make its test like this:
```python
def test_my_func(self):
    func = MyFunc()
    x1_data = xp.array(...)  # replacements needed
    x2_data = xp.array(...)  # replacements needed
    gy_data = xp.array(...)  # replacements needed
    check_backward(func, (x1_data, x2_data), gy_data)
```

This method creates `Variable` objects with `x_data` and calls `func` with the `Variable` s to get its result as `Variable`. Then, it sets `y_grad` array to `grad` attribute of the result and calls `backward` method to get gradients of the inputs. To check correctness of the gradients, the function calls `numerical_grad()` to calculate numerically the gradients and compares the types of gradients with `chainer.testing.assert_allclose()`. If input objects (`x1_data` or/and `x2_data` in this example) represent integer variables, their gradients are ignored.

You can simplify a test when `MyFunc` gets only one argument:

```python
    check_backward(func, x1_data, gy_data)
```

If `MyFunc` is a loss function which returns a zero-dimensional array, pass `None` to `gy_data`. In this case, it sets 1 to `grad` attribute of the result:

```python
    check_backward(my_loss_func, (x1_data, x2_data), None)
```

If `MyFunc` returns multiple outputs, pass all gradients for outputs as a tuple:

```python
    gy1_data = xp.array(...)  # replacements needed
    gy2_data = xp.array(...)  # replacements needed
    check_backward(func, x1_data, (gy1_data, gy2_data))
```

You can also test a `Link`. To check gradients of parameters of the link, set a tuple of the parameters to `params` arguments:

```python
    check_backward(my_link, (x1_data, x2_data), gy_data,
                   (my_link.W, my_link.b))
```

Note that `params` are not `ndarray` s, but `Variables`.

Function objects are acceptable as `func` argument:

```python
    check_backward(lambda x1, x2: f(x1, x2),
                   (x1_data, x2_data), gy_data)
```

**Note:** `func` is called many times to get numerical gradients for all inputs. This function doesn’t work correctly when `func` behaves randomly as it gets different gradients.

**Parameters**

- `func` (*callable*) – A function which gets `Variable` s and returns `Variable` s. `func` must returns a tuple of `Variable` s or one `Variable`. You can use `Function` object, `Link` object or a function satisfying the condition.

- `x_data` (*ndarray or tuple of ndarrays*) – A set of `ndarray` s to be passed to `func`. If `x_data` is one `ndarray` object, it is treated as `(x_data,)`.

---

3.3. Assertion and Testing 117
• **y_grad** *(ndarray or tuple of ndarrays or None)* – A set of ndarray representing gradients of return-values of `func`. If `y_grad` is one ndarray object, it is treated as `(y_grad,)`. If `func` is a loss-function, `y_grad` should be set to `None`.

• **params** *(Variable or tuple of ~chainer.Variable)* – A set of `Variable` s whose gradients are checked. When `func` is a Link object, set its parameters as `params`. If `params` is one `Variable` object, it is treated as `(params,)`.

• **eps** *(float)* – Epsilon value to be passed to `numerical_grad()`.

• **atol** *(float)* – Absolute tolerance to be passed to `chainer.testing.assert_allclose()`.

• **rtol** *(float)* – Relative tolerance to be passed to `chainer.testing.assert_allclose()`.

• **no_grads** *(list of bool)* – Flag to skip variable for gradient assertion. It should be same length as `x_data`.

• **dtype** *(dtype)* – `x_data` and `y_grad` are casted to this dtype when calculating numerical gradients. Only float types and `None` are allowed.

See: `numerical_grad()`

**chainer.gradient_check.numerical_grad**

`chainer.gradient_check.numerical_grad(f, inputs, grad_outputs, eps=0.001)`

Computes numerical gradient by finite differences.

This function is used to implement gradient check. For usage example, see unit tests of `chainer.functions`.

**Parameters**

• **f** *(function)* – Python function with no arguments that runs forward computation and returns the result.

• **inputs** *(tuple of arrays)* – Tuple of arrays that should be treated as inputs. Each element of them is slightly modified to realize numerical gradient by finite differences.

• **grad_outputs** *(tuple of arrays)* – Tuple of arrays that are treated as output gradients.

• **eps** *(float)* – Epsilon value of finite differences.

**Returns** Numerical gradient arrays corresponding to `inputs`.

**Return type** tuple

**Standard Assertions**

The assertions have same names as NumPy’s ones. The difference from NumPy is that they can accept both `numpy.ndarray` and `cupy.ndarray`.

`chainer.testing.assert_allclose` Asserts if some corresponding element of `x` and `y` differs too much.
chainer.testing.assert_allclose

chainer.testing.assert_allclose(x, y, atol=1e-05, rtol=0.0001, verbose=True)

Asserts if some corresponding element of x and y differs too much.

This function can handle both CPU and GPU arrays simultaneously.

Parameters

- `x` – Left-hand-side array.
- `y` – Right-hand-side array.
- `atol` (`float`) – Absolute tolerance.
- `rtol` (`float`) – Relative tolerance.
- `verbose` (`bool`) – If True, it outputs verbose messages on error.

Function testing utilities

Chainer provides some utilities for testing its functions.

chainer.testing.unary_math_function_unittest

chainer.testing.unary_math_function_unittest(func, func_expected=None, label_expected=None, make_data=None)

Decorator for testing unary mathematical Chainer functions.

This decorator makes test classes test unary mathematical Chainer functions. Tested are forward and backward computations on CPU and GPU across parameterized shape and dtype.

Parameters

- `func` (`Function`) – Chainer function to be tested by the decorated test class.
- `func_expected` – Function used to provide expected values for testing forward computation. If not given, a corresponding numpy function for func is implicitly picked up by its class name.
- `label_expected` (`string`) – String used to test labels of Chainer functions. If not given, the class name of func lowered is implicitly used.
- `make_data` – Function to customize input and gradient data used in the tests. It takes shape and dtype as its arguments, and returns a tuple of input and gradient data. By default, uniform distribution ranged \([-1, 1]\) is used for both.

The decorated test class tests forward and backward computations on CPU and GPU across the following parameterized ed parameters:

- `shape`: rank of zero, and rank of more than zero
- `dtype`: `numpy.float16`, `numpy.float32` and `numpy.float64`

Additionally, it tests the label of the Chainer function.

Chainer functions tested by the test class decorated with the decorator should have the following properties:

- Unary, taking one parameter and returning one value
• dtype of input and output are the same
• Elementwise operation for the supplied ndarray

Example

The following code defines a test class that tests sin() Chainer function, which takes a parameter with dtype of float and returns a value with the same dtype.

```python
>>> import unittest
>>> from chainer import testing
>>> from chainer import functions as F
>>> @testing.unary_math_function_unittest(F.Sin())
... class TestSin(unittest.TestCase):
...    pass

Because the test methods are implicitly injected to TestSin class by the decorator, it is enough to place pass in the class definition.

Now the test is run with nose module.

```python
>>> import nose
>>> nose.run(
...    defaultTest=__name__,
...    argv=['', '-a', '!gpu'],
...    exit=False)
True
```

To customize test data, make_data optional parameter can be used. The following is an example of testing sqrt Chainer function, which is tested in positive value domain here instead of the default input.

```python
>>> import numpy
>>> def make_data(shape, dtype):
...    x = numpy.random.uniform(0.1, 1, shape).astype(dtype)
...    gy = numpy.random.uniform(-1, 1, shape).astype(dtype)
...    return x, gy
>>> @testing.unary_math_function_unittest(F.Sqrt(),
...    make_data=make_data)
... class TestSqrt(unittest.TestCase):
...    pass

>>>
```

make_data function which returns input and gradient data generated in proper value domains with given shape and dtype parameters is defined, then passed to the decorator's make_data parameter.

### Standard Function implementations

Chainer provides basic Function implementations in the chainer.functions package. Most of them are wrapped by plain Python functions, which users should use.
**Note:** As of v1.5, the concept of parameterized functions are gone, and they are replaced by corresponding `Link` implementations. They are still put in the `functions` namespace for backward compatibility, though it is strongly recommended to use them via the `chainer.links` package.

## Activation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.functions.clipped_relu</code></td>
<td>Clipped Rectifier Unit function.</td>
</tr>
<tr>
<td><code>chainer.functions.crelu</code></td>
<td>Concatenated Rectified Linear Unit function.</td>
</tr>
<tr>
<td><code>chainer.functions.elu</code></td>
<td>Exponential Linear Unit function.</td>
</tr>
<tr>
<td><code>chainer.functions.leaky_relu</code></td>
<td>Leaky Rectified Linear Unit function.</td>
</tr>
<tr>
<td><code>chainer.functions.log_softmax</code></td>
<td>Channel-wise log-softmax function.</td>
</tr>
<tr>
<td><code>chainer.functions.lstm</code></td>
<td>Long Short-Term Memory units as an activation function.</td>
</tr>
<tr>
<td><code>chainer.functions.maxout</code></td>
<td>Maxout activation function.</td>
</tr>
<tr>
<td><code>chainer.functions.prelu</code></td>
<td>Parametric ReLU function.</td>
</tr>
<tr>
<td><code>chainer.functions.relu</code></td>
<td>Rectified Linear Unit function.</td>
</tr>
<tr>
<td><code>chainer.functions.sigmoid</code></td>
<td>Element-wise sigmoid logistic function.</td>
</tr>
<tr>
<td><code>chainer.functions.slstm</code></td>
<td>S-LSTM units as an activation function.</td>
</tr>
<tr>
<td><code>chainer.functions.softmax</code></td>
<td>Softmax function.</td>
</tr>
<tr>
<td><code>chainer.functions.softmax</code></td>
<td>Element-wise softplus function.</td>
</tr>
<tr>
<td><code>chainer.functions.tanh</code></td>
<td>Elementwise hyperbolic tangent function.</td>
</tr>
</tbody>
</table>

### chainer.functions.clipped_relu

**chainer.functions.clipped_relu**`(x, z=20.0)`  
Clipped Rectifier Unit function.

For a clipping value $z(>0)$, it computes

#### Parameters

- **x** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Input variable. A $(s_1, s_2, ..., s_n)$-shaped float array.
- **z** (*float*) – Clipping value. (default = 20.0)

#### Returns

Output variable. A $(s_1, s_2, ..., s_n)$-shaped float array.

#### Return type

*Variable*

### Example

```python
>>> x = np.random.uniform(-100, 100, (10, 20)).astype('f')
>>> z = 10.0
>>> np.any(x < 0)
True
>>> np.any(x > z)
True
>>> y = F.clipped_relu(x, z=z)
>>> np.any(y.data < 0)
```

---

3.4. **Standard Function implementations** 121
>>> np.any(y.data > z)
False

chainer.functions.crelu

chainer.functions.crelu(x, axis=1)
Concatenated Rectified Linear Unit function.
This function is expressed as follows

\[ f(x) = (\max(0, x), \max(0, -x)) \]

Here, two output values are concatenated along an axis.

See: https://arxiv.org/abs/1603.05201

Parameters

- x (Variable or numpy.ndarray or cupy.ndarray) – Input variable. A \((s_1, s_2, ..., s_N)\)-shaped float array.
- axis (int) – Axis that the output values are concatenated along. Default is 1.

Returns Output variable of concatenated array. If the axis is 1, a \((s_1, s_2 \times 2, ..., s_N)\)-shaped float array.

Return type Variable

Example

```python
>>> x = np.array([[-1, 0], [2, -3]], 'f')
>>> x
array([[-1., 0.],
       [ 2., -3.]], dtype=float32)
>>> y = F.crelu(x, axis=1)
>>> y.data
array([[ 0., 0., 1., 0.],
       [ 2., 0., 0., 3.]], dtype=float32)
```

chainer.functions.elu

chainer.functions.elu(x, alpha=1.0)
Exponential Linear Unit function.
For a parameter \(\alpha\), it is expressed as

\[ f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\exp(x) - 1) & \text{if } x < 0, \end{cases} \]

See: https://arxiv.org/abs/1511.07289

Parameters
• **x** *(Variable or numpy.ndarray or cupy.ndarray)* – Input variable. A \((s_1, s_2, ..., s_N)\)-shaped float array.

• **alpha** *(float)* – Parameter \(\alpha\). Default is 1.0.

Returns Output variable. A \((s_1, s_2, ..., s_N)\)-shaped float array.

Return type **Variable**

Example

```python
>>> x = np.array([[-1, 0], [2, -3]], 'f')
>>> x
array([[-1., 0.],
       [ 2., -3.]], dtype=float32)
>>> y = F.elu(x, alpha=1.)
>>> y.data
array([[-0.63212055, 0. ],
       [ 2. , -0.95021296]], dtype=float32)
```

chainer.functions.hard_sigmoid

chainer.functions.hard_sigmoid(x)

Element-wise hard-sigmoid function.

This function is defined as

\[
f(x) = \begin{cases} 
0 & \text{if } x < -2.5 \\
0.2x + 0.5 & \text{if } -2.5 < x < 2.5 \\
1 & \text{if } 2.5 < x.
\end{cases}
\]

Parameters **x** *(Variable or numpy.ndarray or cupy.ndarray)* – Input variable. A \((s_1, s_2, ..., s_N)\)-shaped float array.

Returns Output variable. A \((s_1, s_2, ..., s_N)\)-shaped float array.

Return type **Variable**

Example

It maps the input values into the range of [0, 1].

```python
>>> x = np.array([-2.6, -1, 0, 1, 2.6])
>>> x
array([-2.6, -1. , 0. , 1. , 2.6])
>>> F.hard_sigmoid(x).data
array([ 0. , 0.3, 0.5, 0.7, 1. ])```

chainer.functions.leaky_relu

chainer.functions.leaky_relu(x, slope=0.2)

Leaky Rectified Linear Unit function.

This function is expressed as

\[
f(x) = \max(x, ax),
\]
where $a$ is a configurable slope value.

**Parameters**

- $x$ (Variable or numpy.ndarray or cupy.ndarray) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.
- slope (float) – Slope value $a$.

**Returns** Output variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

**Return type** Variable

Example

```python
>>> x = np.array([[[-1, 0], [2, -3], [-2, 1]], 'f'])
>>> x
array([[-1., 0.],
       [ 2., -3.],
       [-2., 1.]], dtype=float32)
```

```python
def leaky_relu(x, slope=0.2).data
array([[0.2, 0.],
       [2., -0.60000002],
       [-0.40000001, 1.]], dtype=float32)
```

---

**chainer.functions.log_softmax**

chainer.functions.log_softmax($x$)

Channel-wise log-softmax function.

This function computes its logarithm of softmax along the second axis. Let $c = (c_1, c_2, ..., c_D)$ be the slice of $x$ along with the second axis. For each slice $c$, it computes the logarithm of the function $f(c)$ defined as

$$f(c) = \frac{\exp(c)}{\sum_d \exp(c_d)}.$$  

This method is theoretically equivalent to $\log(\text{softmax}(x))$ but is more stable.

**Note:** $\log(\text{softmax}(x))$ may cause underflow when $x$ is too small, because $\text{softmax}(x)$ may returns 0. $\log_{\text{softmax}}$ method is more stable.

**Parameters** $x$ (Variable or numpy.ndarray or cupy.ndarray) – Input variable. A $n$-dimensional $(n \geq 2)$ float array.

**Returns** Output variable. A $n$-dimensional $(n \geq 2)$ float array, which is the same shape with $x$.

**Return type** Variable

See also:

- $\text{softmax}()$

Example

---
```python
>>> x = np.array([[0, 1, 2], [0, 2, 4]], 'f')
>>> x
array([[ 0., 1., 2.],
       [ 0., 2., 4.]], dtype=float32)
>>> F.log_softmax(x).data
array([[-2.40760589, -1.40760589, -0.40760589],
       [-4.14293146, -2.14293146, -0.14293146]], dtype=float32)
>>> np.allclose(F.log_softmax(x).data, F.log(F.softmax(x)).data)
True
```

**chainer.functions.lstm**

The `chainer.functions.lstm(c_prev, x)` function implements LSTM units with forget gates. Let the previous cell state `c_prev` and the input array `x`.

First, the input array `x` is split into four arrays `a, i, f, o` of the same shapes along the second axis. It means that `x`'s second axis must have 4 times the `c_prev`'s second axis.

The split input arrays are corresponding to:

- `a`: sources of cell input
- `i`: sources of input gate
- `f`: sources of forget gate
- `o`: sources of output gate

Second, it computes the updated cell state `c` and the outgoing signal `h` as:

\[
c = \tanh(a)\sigma(i) + c_{\text{prev}}\sigma(f), \\
h = \tanh(c)\sigma(o),
\]

where `\sigma` is the elementwise sigmoid function. These are returned as a tuple of two variables.

This function supports variable length inputs. The mini-batch size of the current input must be equal to or smaller than that of the previous one. When mini-batch size of `x` is smaller than that of `c`, this function only updates `c[0:len(x)]` and doesn’t change the rest of `c, c[len(x):]`. So, please sort input sequences in descending order of lengths before applying the function.

**Parameters**

- `c_prev (Variable or numpy.ndarray or cupy.ndarray)` – Variable that holds the previous cell state. The cell state should be a zero array or the output of the previous call of LSTM.
- `x (Variable or numpy.ndarray or cupy.ndarray)` – Variable that holds the sources of cell input, input gate, forget gate and output gate. It must have the second dimension whose size is four times of that of the cell state.

**Returns**

Two `Variable` objects `c` and `h`. `c` is the updated cell state. `h` indicates the outgoing signal.

**Return type**

tuple
See the original paper proposing LSTM with forget gates: Long Short-Term Memory in Recurrent Neural Networks.

See also:

LSTM

Example

Assuming \( y \) is the current incoming signal, \( c \) is the previous cell state, and \( h \) is the previous outgoing signal from an \texttt{lstm} function. Each of \( y, c \) and \( h \) has \texttt{n\_units} channels. Most typical preparation of \( x \) is:

```python
>>> n_units = 100
>>> y = chainer.Variable(np.zeros((1, n_units), 'f'))
>>> h = chainer.Variable(np.zeros((1, n_units), 'f'))
>>> c = chainer.Variable(np.zeros((1, n_units), 'f'))
>>> model = chainer.Chain(w=L.Linear(n_units, 4 * n_units),
                       v=L.Linear(n_units, 4 * n_units),)
>>> x = model.w(y) + model.v(h)
>>> c, h = F.lstm(c, x)
```

It corresponds to calculate the input array \( x \), or the input sources \( a, i, f, o \), from the current incoming signal \( y \) and the previous outgoing signal \( h \). Different parameters are used for different kind of input sources.

Note: We use the naming rule below.

- **incoming signal** The formal input of the formulation of LSTM (e.g. in NLP, word vector or output of lower RNN layer). The input of \texttt{chainer.links.LSTM} is the incoming signal.
- **input array** The array which is linear transformed from incoming signal and the previous outgoing signal. The input array contains four sources, the sources of cell input, input gate, forget gate and output gate. The input of \texttt{chainer.functions.LSTM} is the input array.

### chainer.functions.maxout

\texttt{chainer.functions.maxout}(\( x, \text{pool\_size}, \text{axis}=1 \))

Maxout activation function.

It accepts an input tensor \( x \), reshapes the \texttt{axis} dimension (say the size being \( M \times \text{pool\_size} \)) into two dimensions (\( M, \text{pool\_size} \)), and takes maximum along the \texttt{axis} dimension.

**Parameters**

- \( x \) (\texttt{Variable} or \texttt{numpy.ndarray} or \texttt{cupy.ndarray}) – Input variable. A \textprimed{n}-dimensional \((n \geq \text{axis})\) float array. In general, its first dimension is assumed to be the \textit{minibatch dimension}. The other dimensions are treated as one concatenated dimension.

- \texttt{pool\_size} (int) – The size used for downsampling of pooling layer.

- \texttt{axis} (int) – The \texttt{axis} dimension to be reshaped. The size of \texttt{axis} dimension should be \( M \times \text{pool\_size} \).

**Returns** Output variable. The shape of the output is same as \( x \) except that \texttt{axis} dimension is transformed from \( M \times \text{pool\_size} \) to \( M \).

**Return type** \texttt{Variable}
See also:

Maxout

Example

Typically, \( x \) is the output of a linear layer or a convolution layer. The following is the example where we use \( \text{maxout}() \) in combination with a Linear link.

```python
>>> in_size, out_size, pool_size = 10, 10, 10
>>> bias = np.arange(out_size * pool_size).astype('f')
>>> l = L.Linear(in_size, out_size * pool_size, initial_bias=bias)
>>> x = np.zeros((1, in_size), 'f')  # prepare data
>>> x = l(x)
>>> y = F.maxout(x, pool_size)
>>> x.shape
(1, 100)
>>> y.shape
(1, 10)
>>> x.reshape((out_size, pool_size)).data
array([[ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.],
       [40., 41., 42., 43., 44., 45., 46., 47., 48., 49.],
       [50., 51., 52., 53., 54., 55., 56., 57., 58., 59.],
       [60., 61., 62., 63., 64., 65., 66., 67., 68., 69.],
       [70., 71., 72., 73., 74., 75., 76., 77., 78., 79.],
       [80., 81., 82., 83., 84., 85., 86., 87., 88., 89.],
       [90., 91., 92., 93., 94., 95., 96., 97., 98., 99.]],
 dtype=float32)
>>> y.data
 dtype=float32)
```

\[ \text{chainer.functions.prelu} \]

\texttt{chainer.functions.prelu}(x, W)

Parametric ReLU function.

It accepts two arguments: an input \( x \) and a weight array \( W \) and computes the output as \( PReLU(x) = \max(x, W \ast x) \), where \( \ast \) is an elementwise multiplication for each sample in the batch.

When the PReLU function is combined with two-dimensional convolution, the elements of parameter \( a \) are typically shared across the same filter of different pixels. In order to support such usage, this function supports the shape of parameter array that indicates leading dimensions of input arrays except the batch dimension.

For example \( W \) has the shape of \( (2,3,4) \), \( x \) must have the shape of \( (B,2,3,4,S1,...,SN) \) where \( B \) is batch size and the number of trailing \( S \)'s is arbitrary non-negative integer.

**Parameters**

- **x** (Variable) – Input variable. Its first argument is assumed to be the minibatch dimension.

- **W** (Variable) – Weight variable.

**Returns** Output variable
chainer.functions.relu

chainer.functions.relu(x)

Rectified Linear Unit function.

\[ f(x) = \max(0, x). \]

Parameters x (Variable or numpy.ndarray or cupy.ndarray) – Input variable. A \((s_1, s_2, ..., s_N)\)-shaped float array.

Returns Output variable. A \((s_1, s_2, ..., s_N)\)-shaped float array.

Return type Variable

Example

```python
>>> x = np.array([[[-1, 0], [2, -3]], [-2, 1]], 'f')
>>> np.any(x < 0)
True
>>> y = F.relu(x)
>>> np.any(y.data < 0)
False
>>> y.shape
(3, 2)
```

chainer.functions.sigmoid

chainer.functions.sigmoid(x)

Element-wise sigmoid logistic function.

\[ f(x) = \left(1 + \exp(-x)\right)^{-1}. \]

Parameters x (Variable or numpy.ndarray or cupy.ndarray) – Input variable. A \((s_1, s_2, ..., s_N)\)-shaped float array.

Returns Output variable. A \((s_1, s_2, ..., s_N)\)-shaped float array.

Return type Variable

Example

It maps the input values into the range of \([0, 1]\).

```python
>>> x = np.arange(-2, 3, 2).astype('f')
>>> x
array([-2.,  0.,  2.], dtype=float32)
>>> F.sigmoid(x).data
array([ 0.11920291, 0.5 , 0.88079709], dtype=float32)
```
chainer.functions.s lstm

chainer.functions.s lstm(c_prev1, c_prev2, x1, x2)

S-LSTM units as an activation function.

This function implements S-LSTM unit. It is an extension of LSTM unit applied to tree structures. The function is applied to binary trees. Each node has two child nodes. It gets four arguments, previous cell states \(c_{\text{prev}1}\) and \(c_{\text{prev}2}\), and input arrays \(x1\) and \(x2\).

First both input arrays \(x1\) and \(x2\) are split into eight arrays \(a_1, i_1, o_1, f_1, a_2, i_2, o_2, f_2\). They have the same shape along the second axis. It means that \(x1\) and \(x2\) 's second axis must have 4 times the length of \(c_{\text{prev}1}\) and \(c_{\text{prev}2}\).

The split input arrays are corresponding to:

- \(a_i\): sources of cell input
- \(i_i\): sources of input gate
- \(f_i\): sources of forget gate
- \(o_i\): sources of output gate

It computes the updated cell state \(c\) and the outgoing signal \(h\) as:

\[
    c = \tanh(a_1 + a_2)\sigma(i_1 + i_2) + c_{\text{prev}1}\sigma(f_1) + c_{\text{prev}2}\sigma(f_2),
    h = \tanh(c)\sigma(o_1 + o_2),
\]

where \(\sigma\) is the elementwise sigmoid function. The function returns \(c\) and \(h\) as a tuple.

**Parameters**

- **c_prev1** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Variable that holds the previous cell state of the first child node. The cell state should be a zero array or the output of the previous call of LSTM.

- **c_prev2** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Variable that holds the previous cell state of the second child node.

- **x1** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Variable that holds the sources of cell input, input gate, forget gate and output gate from the first child node. It must have the second dimension whose size is four times of that of the cell state.

- **x2** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Variable that holds the input sources from the second child node.

**Returns** Two *Variable* objects \(c\) and \(h\). \(c\) is the cell state. \(h\) indicates the outgoing signal.

**Return type** tuple

See detail in paper: Long Short-Term Memory Over Tree Structures.

**Example**

Assuming \(c1, c2\) is the previous cell state of children, and \(h1, h2\) is the previous outgoing signal from children. Each of \(c1, c2, h1\) and \(h2\) has \(n\_units\) channels. Most typical preparation of \(x1, x2\) is:
It corresponds to calculate the input array $x_1$, or the input sources $a_1, i_1, f_1, o_1$ from the previous cell state of first child node $c_1$ and the previous outgoing signal from first child node $h_1$. Different parameters are used for different kind of input sources.

**chainer.functions.softmax**

```
chainer.functions.softmax(x, axis=1)
```

Softmax function.

This function computes its softmax along an axis. Let $x = (x_1, x_2, ..., x_D)^T$ be the D dimensional index array and $f(x)$ be the D dimensional input array. For each index $x$ of the input array $f(x)$, it computes the probability $p(x)$ defined as $p(x) = \frac{\exp(f(x))}{\sum_d \exp(f(x_d))}$.

**Parameters**

- $x$ (Variable) – Input variable.

**Returns**

Output variable.

**Return type**

Variable

**chainer.functions.softplus**

```
chainer.functions.softplus(x, beta=1.0)
```

Element-wise softplus function.

The softplus function is the smooth approximation of ReLU.

$$f(x) = \frac{1}{\beta} \log(1 + \exp(\beta x)),$$

where $\beta$ is a parameter. The function becomes curved and akin to ReLU as the $\beta$ is increasing.

**Parameters**

- $x$ (Variable or numpy.ndarray or cupy.ndarray) – Input variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

- $\text{beta}$ (float) – Parameter $\beta$.

**Returns**

Output variable. A $(s_1, s_2, ..., s_N)$-shaped float array.

**Return type**

Variable

**Example**
```python
>>> x = np.arange(-2, 3, 2).astype('f')
>>> x
array([-2., 0., 2.], dtype=float32)
>>> F.softplus(x, beta=1.0).data
array([0.12692813, 0.69314718, 2.12692809], dtype=float32)
```

**chainer.functions.tanh**

chainer.functions.tanh(x)

Elementwise hyperbolic tangent function.

\[ f(x) = \tanh(x). \]

**Parameters**

- **x** ([Variable or numpy.ndarray or cupy.ndarray]) – Input variable. A \((s_1, s_2, ..., s_N)\)-shaped float array.

**Returns**

Output variable. A \((s_1, s_2, ..., s_N)\)-shaped float array.

**Return type** [Variable]

**Example**

```python
>>> x = np.arange(-1, 4, 2).astype('f')
>>> x
array([-1., 1., 3.], dtype=float32)
>>> F.tanh(x).data
array([-0.76159418, 0.76159418, 0.99505478], dtype=float32)
```

### Array manipulations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>chainer.functions.broadcast</strong></td>
<td>Broadcast given variables.</td>
</tr>
<tr>
<td><strong>chainer.functions.broadcast_to</strong></td>
<td>Broadcast a given variable to a given shape.</td>
</tr>
<tr>
<td><strong>chainer.functions.cast</strong></td>
<td>Cast an input variable to a given type.</td>
</tr>
<tr>
<td><strong>chainer.functions.concat</strong></td>
<td>Concatenates given variables along an axis.</td>
</tr>
<tr>
<td><strong>chainer.functions.copy</strong></td>
<td>Copies the input variable onto the specified device.</td>
</tr>
<tr>
<td><strong>chainer.functions.depth2space</strong></td>
<td>Computes the depth2space transformation for subpixel calculations.</td>
</tr>
<tr>
<td><strong>chainer.functions.dstack</strong></td>
<td>Concatenate variables along third axis (depth wise).</td>
</tr>
<tr>
<td><strong>chainer.functions.expand_dims</strong></td>
<td>Expands dimensions of an input variable without copy.</td>
</tr>
<tr>
<td><strong>chainer.functions.flatten</strong></td>
<td>Flatten a given array into one dimension.</td>
</tr>
<tr>
<td><strong>chainer.functions.fliplr</strong></td>
<td>Flip array in the left/right direction.</td>
</tr>
<tr>
<td><strong>chainer.functions.flipud</strong></td>
<td>Flip array in the up/down direction.</td>
</tr>
<tr>
<td><strong>chainer.functions.get_item</strong></td>
<td>Extract elements from array with specified shape, axes and offsets.</td>
</tr>
<tr>
<td><strong>chainer.functions.hstack</strong></td>
<td>Concatenate variables horizontally (column wise).</td>
</tr>
<tr>
<td><strong>chainer.functions.im2col</strong></td>
<td>Extract patches from an image based on the filter.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 3.31 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.functions.pad</code></td>
<td>Pad an input variable.</td>
</tr>
<tr>
<td><code>chainer.functions.pad_sequence</code></td>
<td>Pad given arrays to make a matrix.</td>
</tr>
<tr>
<td><code>chainer.functions.permute</code></td>
<td>Permutates a given variable along an axis.</td>
</tr>
<tr>
<td><code>chainer.functions.reshape</code></td>
<td>Reshapes an input variable without copy.</td>
</tr>
<tr>
<td><code>chainer.functions.resize_images</code></td>
<td>Resize images to the given shape.</td>
</tr>
<tr>
<td><code>chainer.functions.rollaxis</code></td>
<td>Roll the axis backwards to the given position.</td>
</tr>
<tr>
<td><code>chainer.functions.select_item</code></td>
<td>Select elements stored in given indices.</td>
</tr>
<tr>
<td><code>chainer.functions.separate</code></td>
<td>Separates an array along a given axis.</td>
</tr>
<tr>
<td><code>chainer.functions.space2depth</code></td>
<td>Computes the space2depth transformation for subpixel calculations.</td>
</tr>
<tr>
<td><code>chainer.functions.spatial_transformer_grid</code></td>
<td>2D Spatial Transformer grid.</td>
</tr>
<tr>
<td><code>chainer.functions.spatial_transformer_sampler</code></td>
<td>2D Spatial Transformer sampler.</td>
</tr>
<tr>
<td><code>chainer.functions.split_axis</code></td>
<td>Splits given variables along an axis.</td>
</tr>
<tr>
<td><code>chainer.functions.squeeze</code></td>
<td>Remove dimensions of size one from the shape of a ndarray.</td>
</tr>
<tr>
<td><code>chainer.functions.stack</code></td>
<td>Concatenate variables along a new axis.</td>
</tr>
<tr>
<td><code>chainer.functions.swapaxes</code></td>
<td>Swap two axes of a variable.</td>
</tr>
<tr>
<td><code>chainer.functions.tile</code></td>
<td>Construct an array by tiling a given array.</td>
</tr>
<tr>
<td><code>chainer.functions.transpose</code></td>
<td>Permute the dimensions of an input variable without copy.</td>
</tr>
<tr>
<td><code>chainer.functions.transpose_sequence</code></td>
<td>Transpose a list of Variables.</td>
</tr>
<tr>
<td><code>chainer.functions.vstack</code></td>
<td>Concatenate variables vertically (row wise).</td>
</tr>
<tr>
<td><code>chainer.functions.where</code></td>
<td>Choose elements depending on condition.</td>
</tr>
</tbody>
</table>

**chainer.functions.broadcast**

`chainer.functions.broadcast(*args)`

Broadcast given variables.

**Parameters**

`args` *(Variable or numpy.ndarray or cupy.ndarray)* – Input variables to be broadcasted. Each dimension of the shapes of the input variables must have the same size.

**Returns**

Variable or tuple of Variable objects which are broadcasted from given arguments.

**Return type** Variable

**Example**

```python
>>> x = np.random.uniform(0, 1, (3, 2)).astype('f')
>>> y = F.broadcast(x)
>>> np.all(x == y.data)
True
>>> z = np.random.uniform(0, 1, (3, 2)).astype('f')
>>> y, w = F.broadcast(x, z)
>>> np.all(x == y.data) & np.all(z == w.data)
True
```

**chainer.functions.broadcast_to**

`chainer.functions.broadcast_to(x, shape)`

Broadcast a given variable to a given shape.
Parameters

- **x** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Input variable be broadcasted. A \((s_1, s_2, ..., s_N)\)-shaped float array.

- **shape** (*tuple*) – Tuple of *int* of the shape of the output variable.

Returns Output variable broadcasted to the given shape.

Return type *Variable*

Example

```python
>>> x = np.arange(0, 3)
>>> y = F.broadcast_to(x, (3, 3))
>>> y.data
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

---

**chainer.functions.cast**

`chainer.functions.cast(x, typ)`

Cast an input variable to a given type.

Parameters

- **x** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Input variable to be casted. A \((s_1, s_2, ..., s_N)\)-shaped float array.

- **typ** (*str* of *dtype* or *numpy.dtype*) – Typecode or data type to cast.

Returns Variable holding a casted array.

Return type *Variable*

Example

```python
>>> x = np.arange(0, 3, dtype=np.float64)
>>> y = F.cast(x, np.float32)
>>> y.dtype
dtype('float32')
```

---

**chainer.functions.concat**

`chainer.functions.concat(xs, axis=1)`

Concatenates given variables along an axis.

3.4. Standard Function implementations
Parameters

- **xs** (tuple of :class:`Variable` or :class:`numpy.ndarray` or :class:`cupy.ndarray`) – Input variables to be concatenated. The variables must have the same shape, except in the dimension corresponding to axis.

- **axis** (int) – The axis along which the arrays will be joined. Default is 1.

Returns

The concatenated variable.

Return type

:class:`Variable`

Example

```python
>>> x = np.arange(0, 12).reshape(3, 4)
>>> x
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> y = np.arange(0, 3).reshape(3, 1)
>>> y
array([[0],
       [1],
       [2]])
>>> z = F.concat((x, y), axis=1)
>>> z.data
array([[ 0,  1,  2,  3,  0],
       [ 4,  5,  6,  7,  1],
       [ 8,  9, 10, 11,  2]])
```

.. currentsection:: chainer.functions.copy

:class:`chainer.functions.copy`

**chainer.functions.copy** (:obj:`x`, :obj:`dst`)

Copies the input variable onto the specified device.

This function copies the array of input variable onto the device specified by :obj:`dst`. When :obj:`dst == -1`, it copies the array onto the host memory. This function supports copies from host to host, from host to device, from device to device and from device to host.

Parameters

- **x** (:class:`Variable` or :class:`numpy.ndarray` or :class:`cupy.ndarray`) – Variable to be copied.

- **dst** (int) – Target device specifier.

Returns

Output variable.

Return type

:class:`Variable`

Example

```python
>>> import chainer.cuda as cuda

>>> x = np.random.uniform(-1, 1, (5, 10))
>>> cuda.get_device_from_array(x).id
-1
>>> y = F.copy(x, 0)  # from host to device0
>>> cuda.get_device_from_array(y.data).id
0
```
>>> z = F.copy(y, -1) # from device0 to host
>>> cuda.get_device_from_array(z.data).id
-1

chainer.functions.depth2space

chainer.functions.depth2space($X, r$)
Computes the depth2space transformation for subpixel calculations.

**Parameters**

- $X$ (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Variable holding a 4d array of shape (batch, channel * $r * r$, dim1, dim2).
- $r$ (*int*) – the upscaling factor.

**Returns** A variable holding the upscaled array from interspersed depth layers. The shape is (batch, channel, dim1 * $r$, dim2 * $r$).

**Return type** *Variable*

**Note:** This can be used to compute super-resolution transformations. See [https://arxiv.org/abs/1609.05158](https://arxiv.org/abs/1609.05158) for details.

**See also:**

`space2depth()`

**Example**

```python
>>> X = np.arange(24).reshape(1, 4, 2, 3).astype('f')
>>> X.shape
(1, 4, 2, 3)
>>> X
array([[[[ 0., 1., 2.],
         [ 3., 4., 5.]],
         [[ 6., 7., 8.],
         [ 9., 10., 11.]],
         [[12., 13., 14.],
         [15., 16., 17.]],
         [[18., 19., 20.],
         [21., 22., 23.]]], dtype=float32)
>>> y = F.depth2space(X, 2)
>>> y.shape
(1, 1, 4, 6)
>>> y.data
array([[[[ 0.,  6.,  1.,  7.,  2.,  8.],
         [12., 18., 13., 19., 14., 20.],
         [ 3.,  9.,  4., 10.,  5., 11.],
         [15., 21., 16., 22., 17., 23.]]], dtype=float32)
```
chainer.functions.dstack

chainer.functions.dstack(xs)
Concatenate variables along third axis (depth wise).

**Parameters**
xs (list of Variable or numpy.ndarray or cupy.ndarray) – Input variables to be concatenated. The variables must have the same ndim. When the variables have the third axis (i.e. \( ndim \geq 3 \)), the variables must have the same shape along all but the third axis. When the variables do not have the third axis (i.e. \( ndim < 3 \)), the variables must have the same shape.

**Returns**
Output variable. When the input variables have the third axis (i.e. \( ndim \geq 3 \)), the shapes of inputs and output are the same along all but the third axis. The length of third axis is the sum of the lengths of inputs’ third axis. When the shape of variables are \((N_1, N_2)\) (i.e. \( ndim = 2 \)), the shape of output is \((N_1, N_2, 2)\). When the shape of variables are \((N_1,)\) (i.e. \( ndim = 1 \)), the shape of output is \((1, N_1, 2)\). When the shape of variables are \(()\) (i.e. \( ndim = 0 \)), the shape of output is \((1, 1, 2)\).

**Return type** Variable

**Example**

```python
>>> x1 = np.array((1, 2, 3))
>>> x1.shape
(3,)
>>> x2 = np.array((2, 3, 4))
>>> x2.shape
(3,)
>>> y = F.dstack((x1, x2))
>>> y.shape
(1, 3, 2)
>>> y.data
array([[ 1,  2],
        [ 2,  3],
        [ 3,  4]])
```

```python
>>> x1 = np.arange(0, 6).reshape(3, 2)
>>> x1.shape
(3, 2)
>>> x1
array([[0, 1],
        [2, 3],
        [4, 5]])
>>> x2 = np.arange(6, 12).reshape(3, 2)
>>> x2.shape
(3, 2)
>>> x2
array([[ 6,  7],
        [ 8,  9],
        [10, 11]])
>>> y = F.dstack([x1, x2])
>>> y.shape
(3, 2, 2)
>>> y.data
array([[ [ 0,  6],
            [ 1,  7]],
        [[ 2,  8],
            [ 3,  9]]])
```
x1 = np.arange(0, 12).reshape(3, 2, 2)
x2 = np.arange(12, 18).reshape(3, 2, 1)
y = F.dstack([x1, x2])
y.shape
(3, 2, 3)
y.data
array([[ 0,  1, 12],
       [ 2,  3, 13],
       [ 4,  5, 14],
       [ 6,  7, 15],
       [ 8,  9, 16],
       [10, 11, 17]])

chainer.functions.expand_dims

chainer.functions.expand_dims(x, axis)
Expands dimensions of an input variable without copy.

Parameters

- **x** (`Variable` or `numpy.ndarray` or `cupy.ndarray`) – Input variable.
- **axis** (`int`) – Position where new axis is to be inserted. The `axis` parameter is acceptable when $-\text{ndim} - 1 \leq \text{axis} \leq \text{ndim}$. (`ndim` is the dimension of input variables). When `axis < 0`, the result is the same with `ndim + 1 - |axis|`.

Returns

Variable that holds a expanded input. The `ndim` of output is one greater than that of `x`.

Return type
`Variable`

Example

```python
>>> x = np.array([1, 2, 3])
>>> x.shape
(3,)
>>> y = F.expand_dims(x, axis=0)
>>> y.shape
(1, 3)
>>> y.data
array([[ 1,  2,  3]])
>>> y = F.expand_dims(x, axis=1)
>>> y.shape
(3, 1)
>>> y.data
array([[1],
        [2],
        [3]])
>>> y = F.expand_dims(x, axis=-2)
>>> y.shape
(1, 3)
```
chainer.functions.flatten

**chainer.functions.flatten(x)**
Flatten a given array into one dimension.

**Parameters**
x (Variable or numpy.ndarray or cupy.ndarray) – Input variable.

**Returns**
Output variable flattened to one dimension.

**Return type**
Variable

**Note:** When you input a scalar array (i.e. the shape is ()), you can also get the one dimension array whose shape is (1,).

**Example**

```python
>>> x = np.array([[1, 2], [3, 4]])
>>> x.shape
(2, 2)
>>> y = F.flatten(x)
>>> y.shape
(4,)
>>> y.data
array([1, 2, 3, 4])
```

```python
>>> x = np.arange(8).reshape(2, 2, 2)
>>> x.shape
(2, 2, 2)
>>> y = F.flatten(x)
>>> y.shape
(8,)
>>> y.data
array([0, 1, 2, 3, 4, 5, 6, 7])
```

chainer.functions.flipr

**chainer.functions.flipr(a)**
Flip array in the left/right direction.

**Parameters**
x (Variable) – Input variable.

**Returns**
Output variable.

**Return type**
Variable
chainer.functions.flipud

chainer.functions.flipud(a)
Flip array in the up/down direction.

Parameters xs (Variable) – Input variable.

Returns Output variable.

Return type Variable

chainer.functions.get_item

chainer.functions.get_item(x, slices)
Extract elements from array with specified shape, axes and offsets.

Parameters

• x (Variable) – A variable to be sliced.

• slices (int, slice, Ellipsis, None, integer array-like, boolean array-like or tuple of them) – It is an integer, a slice, an ellipsis, a numpy.newaxis, an integer array-like, a boolean array-like or tuple of them.

Returns

Variable object which contains sliced array of x.

Return type Variable

Note: It only supports types that are supported by CUDA’s atomicAdd when an integer array is included in slices. The supported types are numpy.float32, numpy.int32, numpy.uint32, numpy.uint64 and numpy.ulonglong.

Note: It does not support slices that contains multiple boolean arrays.

Note: See NumPy document for details of indexing.

chainer.functions.hstack

chainer.functions.hstack(xs)
Concatenate variables horizontally (column wise).

Parameters xs (list of Variable or numpy.ndarray or cupy.ndarray) – Input variables to be concatenated. The variables must have the same ndim. When the variables have the second axis (i.e. ndim ≥ 2), the variables must have the same shape along all but the second axis. When the variables do not have the second axis (i.e. ndim < 2), the variables need not to have the same shape.

Returns Output variable. When the input variables have the second axis (i.e. ndim ≥ 2), the shapes of inputs and output are the same along all but the second axis. The length of second axis is the sum of the lengths of inputs’ second axis. When the variables do not have the second axis (i.e. ndim < 2), the shape of output is (N, ) (N is the sum of the input variables’ size).
Return type **Variable**

Example

```python
>>> x1 = np.array((1, 2, 3))
>>> x1.shape
(3,)
>>> x2 = np.array((2, 3, 4))
>>> x2.shape
(3,)
>>> y = F.hstack((x1, x2))
>>> y.shape
(6,)
>>> y.data
array([1, 2, 3, 2, 3, 4])

>>> x1 = np.arange(0, 12).reshape(3, 4)
>>> x1.shape
(3, 4)
>>> x1
array([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])

>>> x2 = np.arange(12, 18).reshape(3, 2)
>>> x2.shape
(3, 2)
>>> x2
array([[12, 13],
        [14, 15],
        [16, 17]])
>>> y = F.hstack([x1, x2])
>>> y.shape
(3, 6)
>>> y.data
array([[ 0,  1,  2,  3, 12, 13],
        [ 4,  5,  6,  7, 14, 15],
        [ 8,  9, 10, 11, 16, 17]])
```

chainer.functions.im2col

**chainer.functions.im2col** *(x, ksize, stride=1, pad=0, cover_all=False, dilate=1)*

Extract patches from an image based on the filter.

This function rearranges patches of an image and put them in the channel dimension of the output.

Patches are extracted at positions shifted by multiples of `stride` from the first position `–pad` for each spatial axis. The right-most (or bottom-most) patches do not run over the padded spatial size.

Notation: here is a notation.

• **n** is the batch size.

• **c** is the number of the input channels.

• **h** and **w** are the height and width of the input image, respectively.

• **k_H** and **k_W** are the height and width of the filters, respectively.

• **s_Y** and **s_X** are the strides of the filter.
• $p_H$ and $p_W$ are the spatial padding sizes.
• $d_Y$ and $d_X$ are the dilation factors of filter application.

The output size $(h_O, w_O)$ is determined by the following equations when `cover_all = False`:

$$
\begin{align*}
    h_O &= (h + 2p_H - k_H - (k_H - 1) \cdot (d_Y - 1))/s_Y + 1, \\
    w_O &= (w + 2p_W - k_W - (k_W - 1) \cdot (d_X - 1))/s_X + 1.
\end{align*}
$$

When `cover_all = True`, the output size is determined by the following equations:

$$
\begin{align*}
    h_O &= (h + 2p_H - k_H - (k_H - 1) + s_Y - 1)/s_Y + 1, \\
    w_O &= (w + 2p_W - k_W - (k_W - 1) + s_X - 1)/s_X + 1.
\end{align*}
$$

Parameters

- `x` ('Variable') – Input variable of shape $(n, c, h, w)$.
- `ksize` ('int or pair of ints') – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k)` are equivalent.
- `stride` ('int or pair of ints') – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- `pad` ('int or pair of ints') – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.
- `cover_all` ('bool') – If True, all spatial locations are rearranged into some output pixels. It may make the output size larger.
- `dilate` ('int or pair of ints') – Dilation factor of filter applications. `dilate=d` and `dilate=(d, d)` are equivalent.

Returns: Output variable whose shape is $(n, c \cdot k_H \cdot k_W, h_O, w_O)$

Return type: 'Variable'

chainer.functions.pad

chainer.functions.pad(x, pad_width, mode, **keywords)

Pad an input variable.

Parameters

- `x` (chainer.Variable or :class: numpy.ndarray or cupy.ndarray) – Input data.
- `pad_width` ('int or array-like') – Number of values padded to the edges of each axis.
- `mode` ('str') – Specifies how the function fills the periphery of the array. The mode is passed to numpy.pad() or cupy.pad(). If it is 'constant', the input is padded by a constant value specified by constant_values.
- `constant_values` ('int or array-like') – Constant values to fill the periphery in the 'constant' mode.

Returns: Output variable.

Return type: 'Variable'
chainer.functions.pad_sequence

chainer.functions.pad_sequence(xs, length=None, padding=0)

Pad given arrays to make a matrix.

Parameters

- **xs** (list of ~chainer.Variable) – Variables you want to concatenate.
- **length** (None or int) – Size of the first dimension of a padded array. If it is None, the longest size of the first dimension of xs is used.
- **padding** (int or float) – Value to fill.

Returns

It returns a padded matrix. Its shape is \((n, \text{length}, \ldots)\), where \(n == \text{len}(xs)\).

Return type Variable

chainer.functions.permute

chainer.functions.permute(x, indices, axis=0, inv=False)

Permutates a given variable along an axis.

This function permutate \(x\) with given \(indices\). That means \(y[i] = x[indices[i]]\) for all \(i\). Note that this result is same as \(y = x\).take(indices). indices must be a permutation of \([0, 1, \ldots, \text{len}(x) - 1]\).

When \(inv\) is True, \(indices\) is treated as its inverse. That means \(y[indices[i]] = x[i]\).

Parameters

- **x** (Variable) – Variable to permutate.
- **indices** (Variable) – Indices to extract from the variable.
- **axis** (int) – Axis that the input array is permutate along.
- **inv** (bool) – If True, \(indices\) is treated as its inverse.

Returns Output variable.

Return type Variable

chainer.functions.reshape

chainer.functions.reshape(x, shape)

Reshapes an input variable without copy.

Parameters

- **x** (Variable or numpy.ndarray or cupy.ndarray) – Input variable.
- **shape** (tuple of int) – Expected shape of the output array. The number of elements which the array of shape contains must be equal to that of input array. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

Returns Variable that holds a reshaped version of the input variable.

Return type Variable
See also:

numpy.reshape(), cupy.reshape()

Example

```python
>>> x = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
>>> y = F.reshape(x, (8,))
>>> y.shape
(8,)
>>> y.data
array([1, 2, 3, 4, 5, 6, 7, 8])
>>> y = F.reshape(x, (4, -1)) # the shape of output is inferred
>>> y.shape
(4, 2)
>>> y.data
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> y = F.reshape(x, (4, 3)) # the shape of input and output are not consistent
Traceback (most recent call last):
  ...
chainer.utils.type_check.InvalidType:
  Invalid operation is performed in: Reshape (Forward)
  Expect: prod(in_types[0].shape) == prod((4, 3))
  Actual: 8 != 12
```

chainer.functions.resize_images

chainer.functions.resize_images(x, output_shape)

Resize images to the given shape.

This function resizes 2D data to output_shape. Currently, only bilinear interpolation is supported as the sampling method.

Notation: here is a notation for dimensionalities.

- \( n \) is the batch size.
- \( c_I \) is the number of the input channels.
- \( h \) and \( w \) are the height and width of the input image, respectively.
- \( h_O \) and \( w_O \) are the height and width of the output image.

Parameters

- **x** (*Variable*) – Input variable of shape \((n, c_I, h, w)\).
- **output_shape** (*tuple*) – This is a tuple of length 2 whose values are \((h_O, w_O)\).
  
  Note that the order of height and width is opposite of the one in OpenCV.

Returns Resized image whose shape is \((n, c_I, h_O, w_O)\).

Return type **Variable**
chainer.functions.rollaxis

chainer.functions.rollaxis(x, axis, start=0)
Roll the axis backwards to the given position.

Parameters

• x (Variable) – Input variable.
• axis (int) – The axis to roll backwards.
• start (int) – The place to which the axis is moved.

Returns Variable whose axis is rolled.

Return type Variable

chainer.functions.select_item

chainer.functions.select_item(x, t)
Select elements stored in given indices.
This function returns t.choose(x.T), that means y[i] == x[i, t[i]] for all i.

Parameters

• x (Variable) – Variable storing arrays.
• t (Variable) – Variable storing index numbers.

Returns Variable that holds t-th element of x.

Return type Variable

chainer.functions.separate

chainer.functions.separate(x, axis=0)
Separates an array along a given axis.
This function separates an array along a given axis. For example, shape of an array is (2, 3, 4). When it separates the array with axis=1, it returns three (2, 4) arrays.
This function is an inverse of chainer.functions.stack().

Parameters

• x (Variable) – Variable to be separated.
• axis (int) – Axis along which variables are separated.

Returns Output variables.

Return type tuple of chainer.Variable

See also:

chainer.functions.stack()
chainer.functions.space2depth

chainer.functions.space2depth(X, r)
Computes the space2depth transformation for subpixel calculations.

Parameters
- **X** *(Variable or numpy.ndarray or cupy.ndarray)* – Variable holding a 4d array of shape (batch, channel, dim1 * r, dim2 * r).
- **r** *(int)* – the downscaling factor.

Returns A variable holding the downscaled layer array from subpixel array sampling. The shape is (batch, channel * r * r, dim1, dim2).

Return type **Variable**

**Note:** This can be used to compute inverse super-resolution transformations. See https://arxiv.org/abs/1609.05158 for details.

**See also:**
depth2space()

**Example**

```python
>>> X = np.arange(24).reshape(1, 1, 4, 6).astype('f')
>>> X.shape
(1, 1, 4, 6)
>>> X
array([[[0., 1., 2., 3., 4., 5.],
        [6., 7., 8., 9., 10., 11.],
        [12., 13., 14., 15., 16., 17.],
        [18., 19., 20., 21., 22., 23.]]], dtype=float32)
>>> y = F.space2depth(X, 2)
>>> y.shape
(1, 4, 2, 3)
>>> y.data
array([[[0., 2., 4.],
        [12., 14., 16.]],
        [[1., 3., 5.],
        [13., 15., 17.]],
        [[6., 8., 10.],
        [18., 20., 22.]],
        [[7., 9., 11.],
        [19., 21., 23.]]], dtype=float32)
```

chainer.functions.spatial_transformer_grid

chainer.functions.spatial_transformer_grid(theta, output_shape, use_cudnn=True)
2D Spatial Transformer grid.
This function generates coordinates of the points sampled from an image to perform warping described in Spatial Transformer Networks.

Given a coordinate in the warped image \((x^t_i, y^t_i)\), the point sampled from the source image \((x^s_i, y^s_i)\) are calculated by the following equation.

\[
\begin{pmatrix}
  x^s_i \\
  y^s_i \\
\end{pmatrix} = \begin{pmatrix}
  \theta_{11} & \theta_{12} & \theta_{13} \\
  \theta_{21} & \theta_{22} & \theta_{23} \\
\end{pmatrix} \begin{pmatrix}
  x^t_i \\
  y^t_i \\
  1 \\
\end{pmatrix}
\]

Notation: here is a notation for dimensionalities.
- \(n\) is the batch size.
- \(h_O\) and \(w_O\) are the height and the width of the output image.

**Parameters**
- \(\text{theta} (\text{Variable})\) – An array of shape \((n, 2, 3)\). This is a batch of \(2 \times 3\) matrix used for the warping described above.
- \(\text{output\_shape (tuple)}\) – A tuple of 2 elements: \(h_O, w_O\).
- \(\text{use\_cudnn (bool)}\) – If True, then this function uses cuDNN if available. Note that, cuDNN supports SpatialTransformerGrid from version 5.0.0.

**Returns** A variable of shape \((n, 2, h_O, w_O)\). In the 2nd dimension, the first element is the coordinate along the x axis, and the second element is the coordinate along the y axis. All the coordinates in the image are scaled to fit range \([-1, 1]\]. This means that the coordinate \((-1, -1)\) corresponds to the upper-left corner of the input image.

**Return type** \(\text{Variable}\)

```
chainer.functions.spatial_transformer_sampler
```

\[ \text{chainer.functions.spatial\_transformer\_sampler} (x, \text{grid}, \text{use\_cudnn=}\text{True}) \]

2D Spatial Transformer sampler.

This is a differentiable image sampler. With a set of sampling points \(\text{grid}\) and an input feature map \(x\), this produces a sampled output feature map.

This function currently only supports bilinear interpolation as a sampling kernel.

When coordinates in \(\text{grid}\) is outside range \([-1, 1]\], values are sampled from a zero padded input image.

Notation: here is a notation for dimensionalities.
- \(n\) is the batch size.
- \(c_I\) is the number of the input channels.
- \(h\) and \(w\) are the height and width of the input image, respectively.
- \(h_O\) and \(w_O\) are the height and width of the output image.

See detail in the following paper: Spatial Transformer Networks.

**Parameters**
- \(x (\text{Variable})\) – Input variable of shape \((n, c_I, h, w)\).
• **grid** (*Variable*) – Coordinate variable of shape \((n, 2, h_O, w_O)\). Each coordinate defines the spatial location in the input where a sampling kernel is applied to get the value at a particular pixel in the output. \(grid[idx, :, i, j]\) corresponds to the coordinate that is used to sample the values for an output pixel at location \((i, j)\).

In the second dimension, the first coordinate corresponds to the location along the horizontal axis, and the second coordinate corresponds to the location along the vertical axis.

The coordinate \((-1, -1)\) corresponds to the upper-left corner of the input image.

• **use_cudnn** (*bool*) – If True, then this function uses cuDNN if available. Note that, cuDNN supports SpatialTransformerSampler from version 5.0.0.

**Returns**  Output feature map of shape \((n, c_I, h_O, w_O)\).

**Return type**  *Variable*

---

**chainer.functions.split_axis**

*chainer.functions.split_axis*(\(x, indices_or_sections, axis, force_tuple=True\))

Splits given variables along an axis.

**Parameters**

- **\(x\)** (*tuple of Variables*) – Variables to be split.
- **indices_or_sections** (*int or 1-D array*) – If this argument is an integer, N, the array will be divided into N equal arrays along axis. If it is a 1-D array of sorted integers, it indicates the positions where the array is split.
- **axis** (*int*) – Axis that the input array is split along.
- **force_tuple** (*bool*) – If True (the default) this method returns a tuple even when the number of outputs is one. Otherwise, if False a Variable will be returned when the number of outputs is one.

**Returns**  
Tuple of *Variable* objects if the number of outputs is more than 1 or *Variable* otherwise. When force_tuple is True, returned value is always a tuple regardless of the number of outputs.

**Return type**  *tuple or Variable*

---

**Note:** This function raises *ValueError* if at least one of the outputs is split to zero-size (i.e. axis-th value of its shape is zero).

---

**chainer.functions.squeeze**

*chainer.functions.squeeze*(\(x, axis=None\))

Remove dimensions of size one from the shape of a ndarray.

**Parameters**

- **axis** (*None or int or tuple of ints*) – A subset of the single-dimensional entries in the shape to remove. If None is supplied, all of them are removed. The dimension index starts at zero. If an axis with dimension greater than one is selected, an error is raised.

---

3.4. Standard Function implementations
Returns  Variable whose dimensions of size 1 are removed.
Return type  Variable

chainer.functions.stack

chainer.functions.stack(xs, axis=0)
Concatenate variables along a new axis.

Parameters

• xs (list of Variable or numpy.ndarray or cupy.ndarray) – Input variables to be concatenated. The variables must have the same shape.

• axis (int) – The axis along which the arrays will be stacked. The axis parameter is acceptable when \(-\text{ndim} \leq \text{axis} \leq \text{ndim}\). (ndim is the dimension of input variables). When \(\text{axis} < 0\), the result is the same with \(\text{ndim} + 1 - |\text{axis}|\).

Returns  Output variable. Let \(x_1, x_2, \ldots, x_n\) and \(y\) be the input variables and the output variable, \(y[:, \ldots, 0, \ldots, :]\) is \(x_1\), \(y[:, \ldots, 1, \ldots, :]\) is \(x_2\) and \(y[:, \ldots, n-1, \ldots, :]\) is \(x_n\) (The indexed axis indicates the axis).
Return type  Variable

Example

```python
>>> x1 = np.arange(0, 12).reshape(3, 4)
>>> x1.shape
(3, 4)
>>> x1
array([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x2 = np.arange(12, 24).reshape(3, 4)
>>> x2.shape
(3, 4)
>>> x2
array([[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])
>>> y = F.stack([x1, x2], axis=0)
>>> y.shape
(2, 3, 4)
>>> y.data
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
        [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])
>>> y = F.stack([x1, x2], axis=1)
>>> y.shape
(3, 2, 4)
>>> y.data
array([[[ 0,  1,  2,  3],
        [12, 13, 14, 15]],
        [[ 4,  5,  6,  7],
        [16, 17, 18, 19]],
        [[ 8,  9, 10, 11],
        [20, 21, 22, 23]])
```


```python
>> y = F.stack([x1, x2], axis=2)
>> y.shape
(3, 4, 2)
>> y.data
array([[ 0, 12],
    [ 1, 13],
    [ 2, 14],
    [ 3, 15]],
   [[ 4, 16],
    [ 5, 17],
    [ 6, 18],
    [ 7, 19]],
   [[ 8, 20],
    [ 9, 21],
    [10, 22],
    [11, 23]])
>> y = F.stack([x1, x2], axis=-1)
>> y.shape
(3, 4, 2)
```

**chainer.functions.swapaxes**

**chainer.functions.swapaxes** *(x, axis1, axis2)*

Swap two axes of a variable.

**Parameters**

- `x` *(Variable)* – Input variable.
- `axis1` *(int)* – The first axis to swap.
- `axis2` *(int)* – The second axis to swap.

**Returns** Variable whose axes are swapped.

**Return type** *Variable*

**chainer.functions.tile**

**chainer.functions.tile** *(x, reps)*

Construct an array by tiling a given array.

**Parameters**

- `x` *(Variable or numpy.ndarray or cupy.ndarray)* – Input variable. Let the length of `reps` be `d`. If `x.ndim < d`, `x` is treated as `d`-dimensional array by prepending new axes. For example, when the shape of `x` is `(2,)` and tiled with 2-dim repetitions, `x` is treated as the shape `(1, 2)`. If `x.ndim > d`, `reps` is treated as `x.ndim`-dimensional by pre-pending 1’s. For example, when the shape of `x` is `(2, 3, 2, 3)`, the 2-dim `reps` of `(2, 2)` is treated as `(1, 1, 2, 2)`.
• **reps** (int or tuple of int s) – The number of times which x is replicated along each axis.

**Returns** The tiled output Variable. Let the length of reps be d, the output has the dimension of max(d, x.ndim).

**Return type** Variable

**Example**

```python
>>> x = np.array([0, 1, 2])
>>> x.shape
(3,)
>>> y = np.tile(x, 2)
>>> y.shape
(6,)
>>> y
array([0, 1, 2, 0, 1, 2])
>>> y = np.tile(x, (2, 2))
>>> y.shape
(2, 6)
>>> y
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
>>> y = np.tile(x, (2, 1, 2))
>>> y.shape
(2, 1, 6)
>>> y
array([[[0, 1, 2, 0, 1, 2],
        [0, 1, 2, 0, 1, 2]],
       [[0, 1, 2, 0, 1, 2],
        [0, 1, 2, 0, 1, 2]]])

>>> x = np.array([[1, 2], [3, 4]])
>>> x.shape
(2, 2)
>>> y = np.tile(x, 2)
>>> y.shape
(2, 4)
>>> y
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
>>> y = np.tile(x, (2, 2))
>>> y.shape
(4, 4)
>>> y
array([[1, 2, 1, 2],
       [3, 4, 3, 4],
       [1, 2, 1, 2],
       [3, 4, 3, 4]])
>>> y = np.tile(x, (2, 1, 2))
>>> y.shape
(2, 2, 4)
>>> y
array([[[1, 2, 1, 2],
        [3, 4, 3, 4]],
       [[[1, 2, 1, 2],
         [3, 4, 3, 4]]])
```
chainer.functions.transpose

chainer.functions.transpose(x, axes=None)
Permute the dimensions of an input variable without copy.

Parameters

• x (Variable) – Input variable.
• axes (tuple of ints) – By default, reverse the dimensions, otherwise permute the axes according to the values given.

Returns Variable whose axes are permuted.
Return type Variable

chainer.functions.transpose_sequence

chainer.functions.transpose_sequence(xs)
Transpose a list of Variables.

This function transposes a list of Variables and returns a list of Variables. For example a user gives [(0, 1, 2, 3), (4, 5), (6)], the function returns [(0, 4, 6), (1, 5), (2), (3)]. Note that a given list needs to be sorted by each length of Variable.

Parameters xs (list of ~chainer.Variable) – Variables to transpose.

Returns Transposed list.
Return type tuple or Variable

chainer.functions.vstack

chainer.functions.vstack(xs)
Concatenate variables vertically (row wise).

Parameters xs (list of Variable or numpy.ndarray or cupy.ndarray) – Input variables to be concatenated. The variables must have the same ndim. When the variables have the second axis (i.e. ndim ≥ 2), the variables must have the same shape along all but the first axis. When the variables do not have the second axis (i.e. ndim < 2), the variables must have the same shape.

Returns Output variable. When the input variables have the second axis (i.e. ndim ≥ 2), the shapes of inputs and output are the same along all but the first axis. The length of first axis is the sum of the lengths of inputs’ first axis. When the variables do not have the second axis (i.e. ndim < 2), the shape of output is (2, N) (N is the size of the input variable).

Return type Variable

Example

```python
>>> x1 = np.array([1, 2, 3])
>>> x1.shape
(3,)
>>> x2 = np.array([2, 3, 4])
>>> x2.shape
```
```python
>>> y = F.vstack((x1, x2))
>>> y.shape
(2, 3)
>>> y.data
array([[1, 2, 3],
       [2, 3, 4]])
>>> x1 = np.arange(0, 12).reshape(3, 4)
>>> x1.shape
(3, 4)
>>> x1
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> x2 = np.arange(12, 20).reshape(2, 4)
>>> x2.shape
(2, 4)
>>> x2
array([[12, 13, 14, 15],
       [16, 17, 18, 19]])
>>> y = F.vstack([x1, x2])
>>> y.shape
(5, 4)
>>> y.data
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

### chainer.functions.where

This function choose values depending on a given `condition`. All `condition`, `x`, and `y` must have the same shape.

**Parameters**

- `condition` (*Variable*) – Variable containing the condition. Only boolean array is permitted.
- `x` (*Variable*) – Variable chosen when `condition` is True.
- `y` (*Variable*) – Variable chosen when `condition` is False.

**Returns** Variable containing chosen values.

**Return type** *Variable*

### Neural network connections

**chainer.functions.bilinear** Applies a bilinear function based on given parameters.
Table 3.32 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.functions.convolution_2d</code></td>
<td>Two-dimensional convolution function.</td>
</tr>
<tr>
<td><code>chainer.functions.convolution_nd</code></td>
<td>N-dimensional convolution function.</td>
</tr>
<tr>
<td><code>chainer.functions.deconvolution_2d</code></td>
<td>Two dimensional deconvolution function.</td>
</tr>
<tr>
<td><code>chainer.functions.deconvolution_nd</code></td>
<td>N-dimensional deconvolution function.</td>
</tr>
<tr>
<td><code>chainer.functions.depthwise_convolution_2d</code></td>
<td>Two-dimensional depthwise convolution function.</td>
</tr>
<tr>
<td><code>chainer.functions.dilated_convolution_2d</code></td>
<td>Two-dimensional dilated convolution function.</td>
</tr>
<tr>
<td><code>chainer.functions.embed_id</code></td>
<td>Efficient linear function for one-hot input.</td>
</tr>
<tr>
<td><code>chainer.functions.linear</code></td>
<td>Linear function, or affine transformation.</td>
</tr>
<tr>
<td><code>chainer.functions.n_step_bigru</code></td>
<td>Stacked Bi-directional Gated Recurrent Unit function.</td>
</tr>
<tr>
<td><code>chainer.functions.n_step_bilstm</code></td>
<td>Stacked Bi-directional Long Short-Term Memory function.</td>
</tr>
<tr>
<td><code>chainer.functions.n_step_birnn</code></td>
<td>Stacked Bi-directional RNN function for sequence inputs.</td>
</tr>
<tr>
<td><code>chainer.functions.n_step_gru</code></td>
<td>Stacked Uni-directional Gated Recurrent Unit function.</td>
</tr>
<tr>
<td><code>chainer.functions.n_step_lstm</code></td>
<td>Stacked Uni-directional Long Short-Term Memory function.</td>
</tr>
<tr>
<td><code>chainer.functions.n_step_rnn</code></td>
<td>Stacked Uni-directional RNN function for sequence inputs.</td>
</tr>
</tbody>
</table>

**chainer.functions.bilinear**

chainer.functions.bilinear(e1, e2, W1=None, V2=None, b=None)

Applies a bilinear function based on given parameters.

This is a building block of Neural Tensor Network (see the reference paper below). It takes two input variables and one or four parameters, and outputs one variable.

To be precise, denote six input arrays mathematically by $e_1 \in \mathbb{R}^{I \times J}$, $e_2 \in \mathbb{R}^{I \times K}$, $W \in \mathbb{R}^{J \times K \times L}$, $V^1 \in \mathbb{R}^{J \times L}$, $V^2 \in \mathbb{R}^{K \times L}$, and $b \in \mathbb{R}^L$, where $I$ is mini-batch size. In this document, we call $V^1$, $V^2$, and $b$ linear parameters.

The output of forward propagation is calculated as

$$y_{il} = \sum_{jk} e_{ij}^1 e_{ik}^2 W_{jkl} + \sum_j e_{ij}^1 V_{jl}^1 + \sum_k e_{ik}^2 V_{kl}^2 + b_l.$$  

Note that $V^1$, $V^2$, $b$ are optional. If these are not given, then this function omits the last three terms in the above equation.

**Note:** This function accepts an input variable $e_1$ or $e_2$ of a non-matrix array. In this case, the leading dimension is treated as the batch dimension, and the other dimensions are reduced to one dimension.

**Note:** In the original paper, $J$ and $K$ must be equal and the author denotes $[V^1 V^2]$ (concatenation of matrices) by $V$.

**Parameters**

- **e1 (Variable)** – Left input variable.
- **e2 (Variable)** – Right input variable.
- **W (Variable)** – Quadratic weight variable.
- **V1 (Variable)** – Left coefficient variable.
• **v2** *(Variable)* – Right coefficient variable.

• **b** *(Variable)* – Bias variable.

Returns  
Output variable.

Return type  
Variable

See:  
Reasoning With Neural Tensor Networks for Knowledge Base Completion [Socher+, NIPS2013].

**chainer.functions.convolution_2d**

chainer.functions.convolution_2d(x, W=None, b=None, stride=1, pad=0, cover_all=False)

Two-dimensional convolution function.

This is an implementation of two-dimensional convolution in ConvNets. It takes three variables: the input image \(x\), the filter weight \(W\), and the bias vector \(b\).

Notation: here is a notation for dimensionalities.

- \(n\) is the batch size.
- \(c_I\) and \(c_O\) are the number of the input and output channels, respectively.
- \(h_I\) and \(w_I\) are the height and width of the input image, respectively.
- \(h_K\) and \(w_K\) are the height and width of the filters, respectively.
- \(h_P\) and \(w_P\) are the height and width of the spatial padding size, respectively.

Then the Convolution2D function computes correlations between filters and patches of size \((h_K, w_K)\) in \(x\). Note that correlation here is equivalent to the inner product between expanded vectors. Patches are extracted at positions shifted by multiples of stride from the first position \((-h_P, -w_P)\) for each spatial axis. The right-most (or bottom-most) patches do not run over the padded spatial size.

Let \((s_Y, s_X)\) be the stride of filter application. Then, the output size \((h_O, w_O)\) is determined by the following equations:

\[
\begin{align*}
h_O &= (h_I + 2h_P - h_K)/s_Y + 1, \\
w_O &= (w_I + 2w_P - w_K)/s_X + 1.
\end{align*}
\]

If cover_all option is True, the filter will cover the all spatial locations. So, if the last stride of filter does not cover the end of spatial locations, an additional stride will be applied to the end part of spatial locations. In this case, the output size \((h_O, w_O)\) is determined by the following equations:

\[
\begin{align*}
h_O &= (h_I + 2h_P - h_K + s_Y - 1)/s_Y + 1, \\
w_O &= (w_I + 2w_P - w_K + s_X - 1)/s_X + 1.
\end{align*}
\]

If the bias vector is given, then it is added to all spatial locations of the output of convolution.

The output of this function can be non-deterministic when it uses cuDNN. If chainer.configuration.config.cudnn_deterministic is True and cuDNN version is >= v3, it forces cuDNN to use a deterministic algorithm.

**Warning:** deterministic argument is not supported anymore since v2. Instead, use chainer.using_config('cudnn_deterministic', value) (value is either True or False). See chainer.using_config().

Parameters
Chainer Documentation, Release 2.0.2

- **x** (`Variable` or `numpy.ndarray` or `cupy.ndarray`) – Input variable of shape 
  \((n, c_i, h_i, w_i)\).

- **W** (`Variable` or `numpy.ndarray` or `cupy.ndarray`) – Weight variable of shape 
  \((c_O, c_i, h_K, w_K)\).

- **b** (`Variable` or `numpy.ndarray` or `cupy.ndarray`) – Bias variable of length 
  \(c_O\) (optional).

- **stride** (int or pair of int s) – Stride of filter applications. 
  \(s\) and \((s, s)\) are equivalent.

- **pad** (int or pair of int s) – Spatial padding width for input arrays. 
  \(p\) and \((p, p)\) are equivalent.

- **cover_all** (bool) – If True, all spatial locations are convoluted into some output pixels.

**Returns** Output variable of shape \((n, c_O, h_O, w_O)\).

**Return type** `Variable`

**See also:** 
`Convolution2D`

**Example**

```python
>>> n = 10
>>> c_i, c_o = 3, 1
>>> h_i, w_i = 30, 40
>>> h_k, w_k = 10, 10
>>> h_p, w_p = 5, 5
>>> x = np.random.uniform(0, 1, (n, c_i, h_i, w_i)).astype('f')
>>> x.shape
(10, 3, 30, 40)
>>> W = np.random.uniform(0, 1, (c_o, c_i, h_k, w_k)).astype('f')
>>> W.shape
(1, 3, 10, 10)
>>> b = np.random.uniform(0, 1, (c_o,)).astype('f')
>>> b.shape
(1,)
>>> s_y, s_x = 5, 7
>>> y = F.convolution_2d(x, W, b, stride=(s_y, s_x), pad=(h_p, w_p))
>>> y.shape
(10, 1, 7, 6)
>>> h_o = int((h_i + 2 * h_p - h_k) / s_y + 1)
>>> w_o = int((w_i + 2 * w_p - w_k) / s_x + 1)
>>> y.shape == (n, c_o, h_o, w_o)
True
>>> y = F.convolution_2d(x, W, b, stride=(s_y, s_x), pad=(h_p, w_p), cover_all=True)
>>> y.shape == (n, c_o, h_o, w_o + 1)
True
```

`chainer.functions.convolution_nd`

`chainer.functions.convolution_nd(x, W, b=None, stride=1, pad=0, cover_all=False)`

N-dimensional convolution function.

3.4. Standard Function implementations 155
This is an implementation of N-dimensional convolution which is generalized two-dimensional convolution in ConvNets. It takes three variables: the input \( x \), the filter weight \( W \) and the bias vector \( b \).

Notation: here is a notation for dimensionalities.

- \( N \) is the number of spatial dimensions.
- \( n \) is the batch size.
- \( c_I \) and \( c_O \) are the number of the input and output channels, respectively.
- \( d_1, d_2, \ldots, d_N \) are the size of each axis of the input’s spatial dimensions, respectively.
- \( k_1, k_2, \ldots, k_N \) are the size of each axis of the filters, respectively.
- \( l_1, l_2, \ldots, l_N \) are the size of each axis of the output’s spatial dimensions, respectively.
- \( p_1, p_2, \ldots, p_N \) are the size of each axis of the spatial padding size, respectively.

Then the `convolution_nd` function computes correlations between filters and patches of size \((k_1, k_2, \ldots, k_N)\) in \( x \). Note that correlation here is equivalent to the inner product between expanded tensors. Patches are extracted at positions shifted by multiples of \( \text{stride} \) from the first position \((-p_1, -p_2, \ldots, -p_N)\) for each spatial axis.

Let \((s_1, s_2, \ldots, s_N)\) be the stride of filter application. Then, the output size \((l_1, l_2, \ldots, l_N)\) is determined by the following equations:

\[
l_n = \frac{d_n + 2p_n - k_n}{s_n} + 1 \quad (n = 1, \ldots, N)
\]

If \( \text{cover\_all} \) option is \( \text{True} \), the filter will cover the all spatial locations. So, if the last stride of filter does not cover the end of spatial locations, an additional stride will be applied to the end part of spatial locations. In this case, the output size is determined by the following equations:

\[
l_n = \frac{d_n + 2p_n - k_n + s_n - 1}{s_n} + 1 \quad (n = 1, \ldots, N)
\]

The N-dimensional convolution function is defined as follows.

**Parameters**

- \( \text{x} \) (**Variable** or `numpy.ndarray` or `cupy.ndarray`) – Input variable of shape \((n, c_I, d_1, d_2, \ldots, d_N)\).
- \( \text{W} \) (**Variable** or `numpy.ndarray` or `cupy.ndarray`) – Weight variable of shape \((c_O, c_I, k_1, k_2, \ldots, k_N)\).
- \( \text{b} \) (**Variable** or `numpy.ndarray` or `cupy.ndarray`) – One-dimensional bias variable with length \( c_O \) (optional).
- \( \text{stride} \) (**int** or **tuple** of **int** s) – Stride of filter applications \((s_1, s_2, \ldots, s_N)\). \( \text{stride}=s \) is equivalent to \((s, s, \ldots, s)\).
- \( \text{pad} \) (**int** or **tuple** of **int** s) – Spatial padding width for input arrays \((p_1, p_2, \ldots, p_N)\). \( \text{pad}=p \) is equivalent to \((p, p, \ldots, p)\).
- \( \text{cover\_all} \) (**bool**) – If \( \text{True} \), all spatial locations are convoluted into some output pixels. It may make the output size larger. \( \text{cover\_all} \) needs to be \( \text{False} \) if you want to use cuDNN.

**Returns** Output variable of shape \((n, c_O, l_1, l_2, \ldots, l_N)\).

**Return type** **Variable**

**Note:** This function uses cuDNN implementation for its forward and backward computation if ALL of the following conditions are satisfied:
cuda.cudnn_enabled is True
chainer.config.use_cudnn is 'always' or 'auto'
The number of spatial dimensions is more than one.
cover_all is False
The input’s dtype is equal to the filter weight’s.
The dtype is FP16, FP32 or FP64. (FP16 is only available when cuDNN version $\geq$ v3.)

See also:
ConvolutionND, convolution_2d()

Example

```python
>>> n = 10
>>> c_i, c_o = 3, 1
>>> d1, d2, d3 = 30, 40, 50
>>> k1, k2, k3 = 10, 10, 10
>>> p1, p2, p3 = 5, 5, 5
>>> x = np.random.uniform(0, 1, (n, c_i, d1, d2, d3)).astype('f')
>>> x.shape
(10, 3, 30, 40, 50)
>>> W = np.random.uniform(0, 1, (c_o, c_i, k1, k2, k3)).astype('f')
>>> W.shape
(1, 3, 10, 10, 10)
>>> b = np.random.uniform(0, 1, (c_o)).astype('f')
>>> b.shape
(1,)
>>> s1, s2, s3 = 2, 4, 6
>>> y = F.convolution_nd(x, W, b, stride=(s1, s2, s3), pad=(p1, p2, p3))
>>> y.shape
(10, 1, 16, 11, 9)
>>> l1 = int((d1 + 2 * p1 - k1) / s1 + 1)
>>> l2 = int((d2 + 2 * p2 - k2) / s2 + 1)
>>> l3 = int((d3 + 2 * p3 - k3) / s3 + 1)
>>> y.shape == (n, c_o, l1, l2, l3)
True
>>> y = F.convolution_nd(x, W, b, stride=(s1, s2, s3), pad=(p1, p2, p3), cover_all=True)
>>> y.shape == (n, c_o, l1, l2, l3 + 1)
True
```

chainer.functions.deconvolution_2d

chainer.functions.deconvolution_2d(x, W, b=None, stride=1, pad=0, outsize=None)

Two dimensional deconvolution function.

This is an implementation of two-dimensional deconvolution. In most of deep learning frameworks and papers, this function is called transposed convolution. But because of historical reasons (e.g. paper by Ziller Deconvolutional Networks) and backward compatibility, this function is called deconvolution in Chainer.

It takes three variables: input image x, the filter weight W, and the bias vector b.

Notation: here is a notation for dimensionalities.
• $n$ is the batch size.
• $c_I$ and $c_O$ are the number of the input and output channels, respectively.
• $h_I$ and $w_I$ are the height and width of the input image, respectively.
• $h_K$ and $w_K$ are the height and width of the filters, respectively.
• $h_P$ and $w_P$ are the height and width of the spatial padding size, respectively.

Let $(s_Y, s_X)$ be the stride of filter application. Then, the output size $(h_O, w_O)$ is estimated by the following equations:

$$h_O = s_Y(h_I - 1) + h_K - 2h_P,$$
$$w_O = s_X(w_I - 1) + w_K - 2w_P.$$

The output of this function can be non-deterministic when it uses cuDNN. If `chainer.configuration.config.deterministic` is `True` and cuDNN version is $\geq$ v3, it forces cuDNN to use a deterministic algorithm.

**Warning:** deterministic argument is not supported anymore since v2. Instead, use `chainer.using_config('cudnn_deterministic', value)` (value is either `True` or `False`). See `chainer.using_config()`.

**Parameters**

- `x` (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Input variable of shape $(n, c_I, h_I, w_I)$.
- `W` (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Weight variable of shape $(c_I, c_O, h_K, w_K)$.
- `b` (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Bias variable of length $c_O$ (optional).
- `stride` (*int* or pair of *int*s) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- `pad` (*int* or pair of *int*s) – Spatial padding width for input arrays. `pad=p` and `pad=(p, p)` are equivalent.
- `outsize` (*tuple* of *int*s) – Expected output size of deconvolutional operation. It should be pair of height and width $(h_O, w_O)$. Default value is `None` and the outsize is estimated by input size, stride and pad.

**Returns** Output variable of shape $(n, c_O, h_O, w_O)$.

**Return type** *Variable*

**Example**

```python
>>> n = 10
>>> c_i, c_o = 1, 3
>>> h_i, w_i = 5, 10
>>> h_k, w_k = 10, 10
>>> h_p, w_p = 5, 5
>>> x = np.random.uniform(0, 1, (n, c_i, h_i, w_i)).astype('f')
>>> x.shape
(10, 1, 5, 10)
```
```python
>>> W = np.random.uniform(0, 1, (c_i, c_o, h_k, w_k)).astype('f')
>>> W.shape
(1, 3, 10, 10)
>>> b = np.random.uniform(0, 1, c_o).astype('f')
>>> b.shape
(3,)
>>> s_y, s_x = 5, 5
>>> y = F.deconvolution_2d(x, W, b, stride=(s_y, s_x), pad=(h_p, w_p))
>>> y.shape
(10, 3, 20, 45)
>>> h_o = s_y * (h_i - 1) + h_k - 2 * h_p
>>> w_o = s_x * (w_i - 1) + w_k - 2 * w_p
>>> y.shape == (n, c_o, h_o, w_o)
True
```

**chainer.functions.deconvolution_nd**

chainer.functions.deconvolution_nd(x, W, b=None, stride=1, pad=0, outsize=None)

N-dimensional deconvolution function.

This is an implementation of N-dimensional deconvolution which generalizes two-dimensional one. In most of deep learning frameworks and papers, this function is called transposed convolution. But because of historical reasons (e.g. paper by Ziller Deconvolutional Networks) and backward compatibility, this function is called deconvolution in Chainer.

It takes three variables: the input \( x \), the filter weight \( W \), and the bias vector \( b \).

Notation: here is a notation for dimensionalities.

- \( N \) is the number of spatial dimensions.
- \( n \) is the batch size.
- \( c_I \) and \( c_O \) are the number of the input and output channels, respectively.
- \( d_1, d_2, ..., d_N \) are the size of each axis of the input's spatial dimensions, respectively.
- \( k_1, k_2, ..., k_N \) are the size of each axis of the filters, respectively.
- \( p_1, p_2, ..., p_N \) are the size of each axis of the spatial padding size, respectively.
- \( s_1, s_2, ..., s_N \) are the stride of each axis of filter application, respectively.

If `outsize` option is `None`, the output size \((l_1, l_2, ..., l_N)\) is determined by the following equations with the items in the above list:

\[
l_n = s_n(d_n - 1) + k_n - 2p_n \quad (n = 1, ..., N)
\]

If `outsize` option is given, the output size is determined by `outsize`. In this case, the `outsize` \( (l_1, l_2, ..., l_N) \) must satisfy the following equations:

\[
d_n = [(l_n + 2p_n - k_n)/s_n] + 1 \quad (n = 1, ..., N)
\]

**Parameters**

- **x** *(Variable or numpy.ndarray or cupy.ndarray)* — Input variable of shape \((n, c_I, d_1, d_2, ..., d_N)\).
- **W** *(Variable or numpy.ndarray or cupy.ndarray)* — Weight variable of shape \((c_I, c_O, k_1, k_2, ..., k_N)\).*
• **b** (*Variable* or `numpy.ndarray` or `cupy.ndarray`) – One-dimensional bias variable with length \(c_O\) (optional).

• **stride** (*int* or *tuple* of *int* \(s\)) – Stride of filter applications \((s_1, s_2, \ldots, s_N)\).
  \(\text{stride}=s\) is equivalent to \((s, s, \ldots, s)\).

• **pad** (*int* or *tuple* of *int* \(p\)) – Spatial padding width for input arrays \((p_1, p_2, \ldots, p_N)\).
  \(\text{pad}=p\) is equivalent to \((p, p, \ldots, p)\).

• **outsize** (*tuple* of *int* \(l\)) – Expected output size of deconvolutional operation. It should be a tuple of ints \((l_1, l_2, \ldots, l_N)\). Default value is `None` and the outsize is estimated by input size, stride and pad.

**Returns** Output variable of shape \((n, c_O, l_1, l_2, \ldots, l_N)\).

**Return type** *Variable*

See also:

`links.DeconvolutionND`, `deconvolution_2d()`

---

**Example**

**Example1**: the case when `outsize` is not given.

```python
>>> n = 10
>>> c_i, c_o = 3, 1
>>> d1, d2, d3 = 5, 10, 15
>>> k1, k2, k3 = 10, 10, 10
>>> p1, p2, p3 = 5, 5, 5
>>> x = np.random.uniform(0, 1, (n, c_i, d1, d2, d3)).astype('f')
>>> x.shape
(10, 3, 5, 10, 15)
>>> W = np.random.uniform(0, 1, (c_i, c_o, k1, k2, k3)).astype('f')
>>> W.shape
(3, 1, 10, 10, 10)
>>> b = np.random.uniform(0, 1, (c_o)).astype('f')
>>> b.shape
(1,)
>>> s1, s2, s3 = 2, 4, 6
>>> y = F.deconvolution_nd(x, W, b, stride=(s1, s2, s3), pad=(p1, p2, p3))
>>> y.shape
(10, 1, 8, 36, 84)
>>> l1 = s1 * (d1 - 1) + k1 - 2 * p1
>>> l2 = s2 * (d2 - 1) + k2 - 2 * p2
>>> l3 = s3 * (d3 - 1) + k3 - 2 * p3
>>> y.shape == (n, c_o, l1, l2, l3)
True
```

**Example2**: the case when `outsize` is given.

```python
>>> n = 10
>>> c_i, c_o = 3, 1
>>> d1, d2, d3 = 5, 10, 15
>>> k1, k2, k3 = 10, 10, 10
>>> p1, p2, p3 = 5, 5, 5
>>> x = np.random.uniform(0, 1, (n, c_i, d1, d2, d3)).astype('f')
>>> x.shape
(10, 3, 5, 10, 15)
>>> W = np.random.uniform(0, 1, (c_i, c_o, k1, k2, k3)).astype('f')
>>> W.shape
(3, 1, 10, 10, 10)
>>> y = F.deconvolution_nd(x, W, b, outsize=(l1, l2, l3), pad=(p1, p2, p3))
>>> y.shape
(10, 1, 8, 36, 84)
>>> l1 = s1 * (d1 - 1) + k1 - 2 * p1
>>> l2 = s2 * (d2 - 1) + k2 - 2 * p2
>>> l3 = s3 * (d3 - 1) + k3 - 2 * p3
>>> y.shape == (n, c_o, l1, l2, l3)
True
```
```
>>> W.shape
(3, 1, 10, 10, 10)
>>> b = np.random.uniform(0, 1, (c_o)).astype('f')
>>> b.shape
(1,)
>>> s1, s2, s3 = 2, 4, 6
>>> l1, l2, l3 = 9, 38, 87
>>> d1 == int((l1 + 2 * p1 - k1) / s1) + 1
True
>>> d2 == int((l2 + 2 * p2 - k2) / s2) + 1
True
>>> d3 == int((l3 + 2 * p3 - k3) / s3) + 1
True
>>> y = F.deconvolution_nd(x, W, b, stride=(s1, s2, s3), pad=(p1, p2, p3),
→outsize=(l1, l2, l3))
>>> y.shape
(10, 1, 9, 38, 87)
>>> y.shape == (n, c_o, l1, l2, l3)
True
```

**chainer.functions.depthwise_convolution_2d**

**chainer.functions. depthwise_convolution_2d** $(x, W, b=None, stride=1, pad=0)$

Two-dimensional depthwise convolution function.

This is an implementation of two-dimensional depthwise convolution. It takes two or three variables: the input image $x$, the filter weight $W$, and optionally, the bias vector $b$.

Notation: here is a notation for dimensionalities.

- $n$ is the batch size.
- $c_I$ is the number of the input.
- $c_M$ is the channel multiplier.
- $h$ and $w$ are the height and width of the input image, respectively.
- $h_O$ and $w_O$ are the height and width of the output image, respectively.
- $k_H$ and $k_W$ are the height and width of the filters, respectively.

**Parameters**

- **x** *(chainer.Variable or numpy.ndarray or cupy.ndarray)* – Input variable of shape $(n, c_I, h, w)$.
- **W** *(Variable)* – Weight variable of shape $(c_M, c_I, k_H, k_W)$.
- **b** *(Variable)* – Bias variable of length $c_M * c_I$ (optional).
- **stride** *(int or pair of ints)* – Stride of filter applications. $stride=s$ and $stride=(s, s)$ are equivalent.
- **pad** *(int or pair of ints)* – Spatial padding width for input arrays. $pad=p$ and $pad=(p, p)$ are equivalent.

**Returns**

Output variable. Its shape is $(n, c_I * c_M, h_O, w_O)$.

**Return type** Variable
Like Convolution2D, DepthwiseConvolution2D function computes correlations between filters and patches of size \((k_H, k_W)\) in \(x\). But unlike Convolution2D, DepthwiseConvolution2D does not add up input channels of filters but concatenates them. For that reason, the shape of outputs of depthwise convolution are \((n, c_I \times c_M, h_O, w_O)\), \(c_M\) is called channel_multiplier.

\((h_O, w_O)\) is determined by the equivalent equation of Convolution2D.

If the bias vector is given, then it is added to all spatial locations of the output of convolution.

See: L. Sifre. Rigid-motion scattering for image classification

See also:

DepthwiseConvolution2D

Example

```python
>>> x = np.random.uniform(0, 1, (2, 3, 4, 7))
>>> W = np.random.uniform(0, 1, (2, 3, 3, 3))
>>> b = np.random.uniform(0, 1, (6,))
>>> y = F.depthwise_convolution_2d(x, W, b)
>>> y.shape
(2, 6, 2, 5)
```

chainer.functions.dilated_convolution_2d

chainer.functions.dilated_convolution_2d(x, W, b=None, stride=1, pad=0, dilate=1, cover_all=False)

Two-dimensional dilated convolution function.

This is an implementation of two-dimensional dilated convolution in ConvNets. It takes three variables: the input image \(x\), the filter weight \(W\), and the bias vector \(b\).

Notation: here is a notation for dimensionalities.

- \(n\) is the batch size.
- \(c_I\) and \(c_O\) are the number of the input and output, respectively.
- \(h\) and \(w\) are the height and width of the input image, respectively.
- \(k_H\) and \(k_W\) are the height and width of the filters, respectively.

Parameters

- \(x\) (Variable) – Input variable of shape \((n, c_I, h, w)\).
- \(W\) (Variable) – Weight variable of shape \((c_O, c_I, k_H, k_W)\).
- \(b\) (Variable) – Bias variable of length \(c_O\) (optional).
- \(stride\) (int or pair of ints) – Stride of filter applications. \(stride=s\) and \(stride=(s, s)\) are equivalent.
- \(pad\) (int or pair of ints) – Spatial padding width for input arrays. \(pad=p\) and \(pad=(p, p)\) are equivalent.
- \(dilate\) (int or pair of ints) – Dilation factor of filter applications. \(dilate=d\) and \(dilate=(d, d)\) are equivalent.
• **cover_all (bool)** – If True, all spatial locations are convoluted into some output pixels. It may make the output size larger.

**Returns** Output variable.

**Return type** Variable

The two-dimensional dilated convolution function is defined as follows. Then the `DilatedConvolution2D` function computes correlations between filters and patches of size \((k_H, k_W)\) in \(x\). Patches here are extracted at intervals of the dilation factor. Note that correlation here is equivalent to the inner product between expanded vectors. Patches are extracted at intervals of the dilation factor and at positions shifted by multiples of `stride` from the first position `-pad` for each spatial axis. The right-most (or bottom-most) patches do not run over the padded spatial size.

Let \((s_Y, s_X)\) be the stride of filter application, \((p_H, p_W)\) the spatial padding size, and \((d_Y, d_X)\) the dilation factor of filter application. Then, the output size \((h_O, w_O)\) is determined by the following equations:

\[
\begin{align*}
h_O &= (h + 2p_H - k_H - (k_H - 1) * (d_Y - 1))/s_Y + 1, \\
w_O &= (w + 2p_W - k_W - (k_W - 1) * (d_X - 1))/s_X + 1.
\end{align*}
\]

If the bias vector is given, then it is added to all spatial locations of the output of convolution.

**See also:**

`DilatedConvolution2D`

---

**chainer.functions.embed_id**

`chainer.functions.embed_id (x, W, ignore_label=None)`

Efficient linear function for one-hot input.

This function implements so called *word embedding*. It takes two arguments: a set of IDs (words) \(x\) in \(B\) dimensional integer vector, and a set of all ID (word) embeddings \(W\) in \(V \times d\) float32 matrix. It outputs \(B \times d\) matrix whose \(i\)-th column is the \(x[i]\)-th column of \(W\).

This function is only differentiable on the input \(W\).

**Parameters**

• **(Variable)** – Batch vectors of IDs.

• **W (Variable)** – Representation of each ID (a.k.a. word embeddings).

• **ignore_label (int or None)** – If ignore_label is an int value, \(i\)-th column of return value is filled with 0.

**Returns** Output variable.

**Return type** Variable

**See also:**

`EmbedID`

---

**chainer.functions.linear**

`chainer.functions.linear (x, W, b=\text{None})`

Linear function, or affine transformation.

It accepts two or three arguments: an input minibatch \(x\), a weight matrix \(W\), and optionally a bias vector \(b\). It computes .. math:: \(Y = xW^\text{top} + b\).
Parameters

- **x** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Input variable, which is a \((s_B, s_1, s_2, \ldots, s_n)\)-shaped float array. Its first dimension \((s_B)\) is assumed to be the mini-batch dimension. The other dimensions are treated as concatenated one dimension whose size must be \((s_1 \times \cdots \times s_n = N)\).

- **W** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Weight variable of shape \((M, N)\), where \((N = s_1 \times \cdots \times s_n)\).

- **b** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Bias variable (optional) of shape \((M,)\).

Returns

Output variable. A float array with shape of \((s_B, M)\).

Return type

*Variable*

See also:

*Linear*

Example

```python
>>> x = np.random.uniform(0, 1, (3, 4)).astype('f')
>>> W = np.random.uniform(0, 1, (5, 4)).astype('f')
>>> b = np.random.uniform(0, 1, (5,)).astype('f')
>>> y = F.linear(x, W, b)
>>> y.shape
(3, 5)
```

**chainer.functions.n_step_bigru**

This function calculates stacked Bi-directional GRU with sequences. This function gets an initial hidden state \(h_0\), an input sequence \(x\), weight matrices \(W\), and bias vectors \(b\). This function calculates hidden states \(h_t\) for each time \(t\) from input \(x_t\).

\[
egin{align*}
    r_t^f &= \sigma(W_t^f x_t + W_t^f h_{t-1} + b_t^f + b_0^f) \\
    z_t^f &= \sigma(W_t^f x_t + W_t^f h_{t-1} + b_t^f + b_0^f) \\
    h_t' &= \tanh(W_t^f x_t + b_t^f + r_t^f \cdot (W_t^f h_{t-1} + b_t^f)) \\
    h_t &= (1 - z_t^f) \cdot h_t' + z_t^f \cdot h_{t-1} \\
    r_t^b &= \sigma(W_t^b x_t + W_t^b h_{t-1} + b_t^b + b_0^b) \\
    z_t^b &= \sigma(W_t^b x_t + W_t^b h_{t-1} + b_t^b + b_0^b) \\
    h_t' &= \tanh(W_t^b x_t + b_t^b + r_t^b \cdot (W_t^b h_{t-1} + b_t^b)) \\
    h_t &= (1 - z_t^b) \cdot h_t' + z_t^b \cdot h_{t-1} \\
    h_t &= [h_t^f; h_t^b]
\end{align*}
\]

where \(W^f\) is weight matrices for forward-GRU, \(W^b\) is weight matrices for backward-GRU.

As the function accepts a sequence, it calculates \(h_t\) for all \(t\) with one call. Six weight matrices and six bias vectors are required for each layers. So, when \(S\) layers exists, you need to prepare \(6S\) weight matrices and \(6S\) bias vectors.
If the number of layers \(n\_layers\) is greater than 1, input of \(k\)-th layer is hidden state \(h_t\) of \(k-1\)-th layer. Note that all input variables except first layer may have different shape from the first layer.

**Parameters**

- **\(n\_layers\)** *(int)* – Number of layers.
- **\(dropout\_ratio\)** *(float)* – Dropout ratio.
- **\(hx\)** *(chainer.Variable)* – Variable holding stacked hidden states. Its shape is \((S, B, N)\) where \(S\) is number of layers and is equal to \(n\_layers\), \(B\) is mini-batch size, and \(N\) is dimension of hidden units.
- **\(ws\)** *(list of list of chainer.Variable)* – Weight matrices. \(ws[i]\) represents weights for \(i\)-th layer. Each \(ws[i]\) is a list containing six matrices. \(ws[i][j]\) is corresponding with \(W_j\) in the equation. Only \(ws[0][j]\) where \(0 <= j < 3\) is \((I, N)\) shape as they are multiplied with input variables. All other matrices has \((N, N)\) shape.
- **\(bs\)** *(list of list of chainer.Variable)* – Bias vectors. \(bs[i]\) represents biases for \(i\)-th layer. Each \(bs[i]\) is a list containing six vectors. \(bs[i][j]\) is corresponding with \(b_j\) in the equation. Shape of each matrix is \((N, )\) where \(N\) is dimension of hidden units.
- **\(xs\)** *(list of chainer.Variable)* – A list of Variable holding input values. Each element \(xs[t]\) holds input value for time \(t\). Its shape is \((B_t, I)\), where \(B_t\) is mini-batch size for time \(t\), and \(I\) is size of input units. Note that this functions supports variable length sequences. When sequences has different lengths, sort sequences in descending order by length, and transpose the sorted sequence. \(\text{transpose_sequence()}\) transpose a list of Variable holding sequence. So \(xs\) needs to satisfy \(xs[t].shape[0] >= xs[t + 1].shape[0]\).
- **\(use\_bi\_direction\)** *(bool)* – If True, this function uses Bi-direction GRU.

**Returns**

This functions returns a tuple containing three elements, \(hy\) and \(ys\).

- **\(hy\)** is an updated hidden states whose shape is same as \(hx\).
- **\(ys\)** is a list of Variable. Each element \(ys[t]\) holds hidden states of the last layer corresponding to an input \(xs[t]\). Its shape is \((B_t, N)\) where \(B_t\) is mini-batch size for time \(t\), and \(N\) is size of hidden units. Note that \(B_t\) is the same value as \(xs[t]\).

**Return type** *tuple*

\[
\text{chainer.functions.n\_step\_bilstm}
\]

\[
\text{chainer.functions.n\_step\_bilstm}(n\_layers, dropout\_ratio, hx, cx, ws, bs, xs)
\]

Stacked Bi-directional Long Short-Term Memory function.

This function calculates stacked Bi-directional LSTM with sequences. This function gets an initial hidden state \(h_0\), an initial cell state \(c_0\), an input sequence \(x\), weight matrices \(W\), and bias vectors \(b\). This function calculates...
hidden states $h_t$ and $c_t$ for each time $t$ from input $x_t$.

\[ i_t^f = \sigma(W^f_0 x_t + W^f_1 h_{t-1} + b^f_0 + b^f_1), \]
\[ f_t^f = \sigma(W^f_1 x_t + W^f_2 h_{t-1} + b^f_1 + b^f_2), \]
\[ o_t^f = \sigma(W^f_2 x_t + W^f_3 h_{t-1} + b^f_2 + b^f_3), \]
\[ a_t^f = \tanh(W^f_3 x_t + W^f_4 h_{t-1} + b^f_3 + b^f_4), \]
\[ c_t^f = f_t^f \cdot c_{t-1}^f + i_t^f \cdot o_t^f, \]
\[ h_t^f = o_t^f \cdot \tanh(c_t^f), \]

\[ i_t^b = \sigma(W^b_0 x_t + W^b_4 h_{t-1} + b^b_0 + b^b_4), \]
\[ f_t^b = \sigma(W^b_1 x_t + W^b_5 h_{t-1} + b^b_1 + b^b_5), \]
\[ o_t^b = \sigma(W^b_2 x_t + W^b_6 h_{t-1} + b^b_2 + b^b_6), \]
\[ a_t^b = \tanh(W^b_3 x_t + W^b_7 h_{t-1} + b^b_3 + b^b_7), \]
\[ c_t^b = f_t^b \cdot c_{t-1}^b + i_t^b \cdot a_t^b, \]
\[ h_t^b = o_t^b \cdot \tanh(c_t^b), \]
\[ h_t = [h_t^f; h_t^b] \]

where $W^f$ is weight matrices for forward-LSTM, $W^b$ is weight matrices for backward-LSTM.

As the function accepts a sequence, it calculates $h_t$ for all $t$ with one call. Eight weight matrices and eight bias vectors are required for each layers. So, when $S$ layers exists, you need to prepare $8S$ weight matrices and $8S$ bias vectors.

If the number of layers $n\_layers$ is greater than 1, input of $k$-th layer is hidden state $h_{t}$ of $k-1$-th layer. Note that all input variables except first layer may have different shape from the first layer.

**Warning:** train and use_cudnn arguments are not supported anymore since v2. Instead, use chainer.using_config('train', train) and chainer.using_config('use_cudnn', use_cudnn) respectively. See chainer.using_config().

**Parameters**

- **n\_layers** *(int)* – Number of layers.
• dropout_ratio (float) – Dropout ratio.

• hx (chainer.Variable) – Variable holding stacked hidden states. Its shape is \((S, B, N)\) where \(S\) is number of layers and is equal to \(n\_layers\), \(B\) is mini-batch size, and \(N\) is dimension of hidden units.

• cx (chainer.Variable) – Variable holding stacked cell states. It has the same shape as \(hx\).

• ws (list of list of chainer.Variable) – Weight matrices. \(ws[i]\) represents weights for \(i\)-th layer. Each \(ws[i]\) is a list containing eight matrices. \(ws[i][j]\) is corresponding with \(W_j\) in the equation. Only \(ws[0][j]\) where \(0 \leq j < 4\) is \((I, N)\) shape as they are multiplied with input variables. All other matrices has \((N, N)\) shape.

• bs (list of list of chainer.Variable) – Bias vectors. \(bs[i]\) represents biases for \(i\)-th layer. Each \(bs[i]\) is a list containing eight vectors. \(bs[i][j]\) is corresponding with \(b_j\) in the equation. Shape of each matrix is \((N,)\) where \(N\) is dimension of hidden units.

• xs (list of chainer.Variable) – A list of Variable holding input values. Each element \(xs[t]\) holds input value for time \(t\). Its shape is \((B_\_t, I)\), where \(B_\_t\) is mini-batch size for time \(t\), and \(I\) is size of input units. Note that this functions supports variable length sequences. When sequences has different lengths, sort sequences in descending order by length, and transpose the sorted sequence. transpose_sequence() transpose a list of Variable() holding sequence. So \(xs\) needs to satisfy \(xs[t].shape[0] >= xs[t+1].shape[0]\).

**Returns**

This functions returns a tuple containing three elements, \(hy, cy\) and \(ys\).

- \(hy\) is an updated hidden states whose shape is same as \(hx\).
- \(cy\) is an updated cell states whose shape is same as \(cx\).
- \(ys\) is a list of Variable. Each element \(ys[t]\) holds hidden states of the last layer corresponding to an input \(xs[t]\). Its shape is \((B_\_t, N)\) where \(B_\_t\) is mini-batch size for time \(t\), and \(N\) is size of hidden units. Note that \(B_\_t\) is the same value as \(xs[t]\).

**Return type** tuple

`chainer.functions.n_step_birnn`

`chainer.functions.n_step_birnn(n_layers, dropout_ratio, hx, ws, bs, xs, activation='tanh')`

Stacked Bi-directional RNN function for sequence inputs.

This function calculates stacked Bi-directional RNN with sequences. This function gets an initial hidden state \(h_0\), an initial cell state \(c_0\), an input sequence \(x\), weight matrices \(W\), and bias vectors \(b\). This function calculates hidden states \(h_t\) and \(c_t\) for each time \(t\) from input \(x_t\).

\[
\begin{align*}
h^f_t &= f(W^f_0 x_t + W^f_1 h_{t-1} + b^f_0 + b^f_1), \\
h^b_t &= f(W^b_0 x_t + W^b_1 h_{t-1} + b^b_0 + b^b_1), \\
h_t &= [h^f_t; h^b_t],
\end{align*}
\]

where \(f\) is an activation function.

3.4. Standard Function implementations
Weight matrices $W$ contains two matrices $W^f$ and $W^b$. $W^f$ is weight matrices for forward directional RNN. $W^b$ is weight matrices for backward directional RNN.

$W^f$ contains $W^f_0$ for an input sequence and $W^f_1$ for a hidden state. $W^b$ contains $W^b_0$ for an input sequence and $W^b_1$ for a hidden state.

Bias matrices $b$ contains two matrices $b^f$ and $b^b$. $b^f$ contains $b^f_0$ for an input sequence and $b^f_1$ for a hidden state. $b^b$ contains $b^b_0$ for an input sequence and $b^b_1$ for a hidden state.

As the function accepts a sequence, it calculates $h_t$ for all $t$ with one call. Two weight matrices and two bias vectors are required for each layer. So, when $S$ layers exist, you need to prepare $2S$ weight matrices and $2S$ bias vectors.

If the number of layers $n_{\text{layers}}$ is greater than 1, input of $k$-th layer is hidden state $h_t$ of $k-1$-th layer. Note that all input variables except first layer may have different shape from the first layer.

**Parameters**

- **n_layers** (*int*) – Number of layers.
- **dropout_ratio** (*float*) – Dropout ratio.
- **hx** (*chainer.Variable*) – Variable holding stacked hidden states. Its shape is $(S, B, N)$ where $S$ is number of layers and is equal to $n_{\text{layers}}, B$ is mini-batch size, and $N$ is dimension of hidden units.
- **ws** (*list of list of chainer.Variable*) – Weight matrices. $ws[i + di]$ represents weights for $i$-th layer. Note that $di = 0$ for forward-RNN and $di = 1$ for backward-RNN. Each $ws[i + di]$ is a list containing two matrices. $ws[i + di][j]$ is corresponding with $W^f_{i,j}$ if $di = 0$ and corresponding with $W^b_{i,j}$ if $di = 1$ in the equation. Only $ws[0][j]$ and $ws[1][j]$ where $0 \leq j < 1$ are $(I, N)$ shape as they are multiplied with input variables. All other matrices has $(N, N)$ shape.
- **bs** (*list of list of chainer.Variable*) – Bias vectors. $bs[i + di]$ represents biases for $i$-th layer. Note that $di = 0$ for forward-RNN and $di = 1$ for backward-RNN. Each $bs[i + di]$ is a list containing two vectors. $bs[i + di][j]$ is corresponding with $b^f_{i,j}$ if $di = 0$ and corresponding with $b^b_{i,j}$ if $di = 1$ in the equation. Shape of each matrix is $(N,)$ where $N$ is dimension of hidden units.
- **xs** (*list of chainer.Variable*) – A list of Variable holding input values. Each element $xs[t]$ holds input value for time $t$. Its shape is $(B_t, I)$, where $B_t$ is mini-batch size for time $t$, and $I$ is size of input units. Note that this functions supports variable length sequences. When sequences has different lengths, sort sequences in descending order by length, and transpose the sorted sequence. $\text{transpose_sequence()}$ transpose a list of Variable holding sequence. So $xs$ needs to satisfy $xs[t].shape[0] >= xs[t + 1].shape[0]$.
- **activation** (*str*) – Activation function name. Please select tanh or relu.

**Returns**

This functions returns a tuple containing three elements, $hy$ and $ys$.

- $hy$ is an updated hidden states whose shape is same as $hx$. 

---

**Warning:** $\text{train}$ and $\text{use_cudnn}$ arguments are not supported anymore since v2. Instead, use $\text{chainer.using_config('train', train)}$ and $\text{chainer.using_config('use_cudnn', use_cudnn)}$ respectively. See $\text{chainer.using_config()}$. 

---

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_layers</td>
<td>Number of layers.</td>
</tr>
<tr>
<td>dropout_ratio</td>
<td>Dropout ratio.</td>
</tr>
<tr>
<td>hx</td>
<td>Variable holding stacked hidden states.</td>
</tr>
<tr>
<td>ws</td>
<td>Weight matrices.</td>
</tr>
<tr>
<td>bs</td>
<td>Bias vectors.</td>
</tr>
<tr>
<td>xs</td>
<td>Input values.</td>
</tr>
<tr>
<td>activation</td>
<td>Activation function name.</td>
</tr>
</tbody>
</table>
• $y$s is a list of $\text{Variable}$. Each element $y_s[t]$ holds hidden states of the last layer corresponding to an input $x_s[t]$. Its shape is $(B_t, N)$ where $B_t$ is mini-batch size for time $t$, and $N$ is size of hidden units. Note that $B_t$ is the same value as $x_s[t]$.

Return type tuple

chainer.functions.n_step_gru

chainer.functions.n_step_gru($n_layers$, $dropout_ratio$, $hx$, $ws$, $bs$, $xs$)

Stacked Uni-directional Gated Recurrent Unit function.

This function calculates stacked Uni-directional GRU with sequences. This function gets an initial hidden state $h_0$, an input sequence $x$, weight matrices $W$, and bias vectors $b$. This function calculates hidden states $h_t$ for each time $t$ from input $x_t$.

\[
\begin{align*}
    r_t &= \sigma(W_0 x_t + W_3 h_{t-1} + b_0 + b_3) \\
    z_t &= \sigma(W_1 x_t + W_4 h_{t-1} + b_1 + b_4) \\
    h'_t &= \tanh(W_2 x_t + b_2 + r_t \cdot (W_5 h_{t-1} + b_5)) \\
    h_t &= (1 - z_t) \cdot h'_t + z_t \cdot h_{t-1}
\end{align*}
\]

As the function accepts a sequence, it calculates $h_t$ for all $t$ with one call. Six weight matrices and six bias vectors are required for each layers. So, when $S$ layers exists, you need to prepare $6S$ weight matrices and $6S$ bias vectors.

If the number of layers $n_layers$ is greater than 1, input of $k$-th layer is hidden state $h_t$ of $k-1$-th layer. Note that all input variables except first layer may have different shape from the first layer.

Warning: train and use_cudnn arguments are not supported anymore since v2. Instead, use chainer.using_config('train', train) and chainer.using_config('use_cudnn', use_cudnn) respectively. See chainer.using_config().

Parameters

• $n_layers$ (int) – Number of layers.
• $dropout_ratio$ (float) – Dropout ratio.
• $hx$ (chainer.Variable) – Variable holding stacked hidden states. Its shape is $(S, B, N)$ where $S$ is number of layers and is equal to $n_layers$, $B$ is mini-batch size, and $N$ is dimension of hidden units.
• $ws$ (list of list of chainer.Variable) – Weight matrices. $w_s[i]$ represents weights for i-th layer. Each $ws[i]$ is a list containing six matrices. $ws[i][j]$ is corresponding with $w_{-j}$ in the equation. Only $ws[0][j]$ where $0 <= j < 3$ is $(I, N)$ shape as they are multiplied with input variables. All other matrices has $(N, N)$ shape.
• $bs$ (list of list of chainer.Variable) – Bias vectors. $bs[i]$ represents biases for i-th layer. Each $bs[i]$ is a list containing six vectors. $bs[i][j]$ is corresponding with $b_{-j}$ in the equation. Shape of each matrix is $(N,)$ where $N$ is dimension of hidden units.
• $xs$ (list of chainer.Variable) – A list of $\text{Variable}$ holding input values. Each element $xs[t]$ holds input value for time $t$. Its shape is $(B_t, I)$, where $B_t$ is mini-batch size for time $t$, and $I$ is size of input units. Note that this functions supports variable length sequences. When sequences has different lengths, sort sequences in descending order by length, and transpose the sorted sequence. transpose_sequence() transpose a
list of Variable() holding sequence. So xs needs to satisfy xs[t].shape[0] >=
xs[t + 1].shape[0].

Returns

This functions returns a tuple conaining three elements, hy and ys.

- hy is an updated hidden states whose shape is same as hx.
- ys is a list of Variable. Each element ys[t] holds hidden states of the last layer
  corresponding to an input xs[t]. Its shape is (B_t, N) where B_t is mini-batch size
  for time t, and N is size of hidden units. Note that B_t is the same value as xs[t].

Return type  tuple

chainer.functions.n_step_lstm

chainer.functions.n_step_lstm(n_layers, dropout_ratio, hx, cx, ws, bs, xs)
Stacked Uni-directional Long Short-Term Memory function.

This function calculates stacked Uni-directional LSTM with sequences. This function gets an initial hidden state
ℎ₀, an initial cell state 𝑐₀, an input sequence 𝑥, weight matrices 𝑊, and bias vectors 𝑏. This function calculates
hidden states ℎₜ and 𝑐ₜ for each time 𝑡 from input 𝑥ₜ.

\[
\begin{align*}
i_t &= \sigma(W_0 x_t + W_4 h_{t-1} + b_0 + b_4) \\
f_t &= \sigma(W_1 x_t + W_5 h_{t-1} + b_1 + b_5) \\
o_t &= \sigma(W_2 x_t + W_6 h_{t-1} + b_2 + b_6) \\
a_t &= \tanh(W_3 x_t + W_7 h_{t-1} + b_3 + b_7) \\
c_t &= f_t \cdot c_{t-1} + i_t \cdot a_t \\
h_t &= o_t \cdot \tanh(c_t)
\end{align*}
\]

As the function accepts a sequence, it calculates ℎₜ for all 𝑡 with one call. Eight weight matrices and eight bias
vectors are required for each layers. So, when 𝑆 layers exists, you need to prepare 8𝑆 weight matrices and 8𝑆
bias vectors.

If the number of layers n_layers is greater than 1, input of 𝑘-th layer is hidden state ℎₜ of 𝑘-1-th layer.
Note that all input variables except first layer may have different shape from the first layer.

Warning: train and use_cudnn arguments are not supported anymore since v2. Instead, use chainer.using_config('train', train) and chainer.
using_config('use_cudnn', use_cudnn) respectively. See chainer.using_config().

Parameters

- n_layers (int) – Number of layers.
- dropout_ratio (float) – Dropout ratio.
- hx (chainer.Variable) – Variable holding stacked hidden states. Its shape is (𝑆, 𝐵, 𝑁) where 𝑆 is number of layers and is equal to n_layers, 𝐵 is mini-batch size, and 𝑁 is
dimention of hidden units.
- cx (chainer.Variable) – Variable holding stacked cell states. It has the same shape
as hx.
• **ws (list of list of chainer.Variable)** – Weight matrices. \(ws[i]\) represents weights for \(i\)-th layer. Each \(ws[i]\) is a list containing eight matrices. \(ws[i][j]\) is corresponding with \(W_j\) in the equation. Only \(ws[0][j]\) where 0 <= \(j\) < 4 is \((I, N)\) shape as they are multiplied with input variables. All other matrices has \((N, N)\) shape.

• **bs (list of list of chainer.Variable)** – Bias vectors. \(bs[i]\) represents biases for \(i\)-th layer. Each \(bs[i]\) is a list containing eight vectors. \(bs[i][j]\) is corresponding with \(b_j\) in the equation. Shape of each matrix is \((N,)\) where \(N\) is dimension of hidden units.

• **xs (list of chainer.Variable)** – A list of Variable holding input values. Each element \(xs[t]\) holds input value for time \(t\). Its shape is \((B_t, I)\), where \(B_t\) is mini-batch size for time \(t\) and \(I\) is size of input units. Note that this functions supports variable length sequences. When sequences has different lengths, sort sequences in descending order by length, and transpose the sorted sequence. `transpose_sequence()` transpose a list of Variable() holding sequence. So \(xs\) needs to satisfy \(xs[t].shape[0] >= xs[t + 1].shape[0]\).

**Returns**

This functions returns a tuple containing three elements, \(hy, cy\) and \(ys\).

- \(hy\) is an updated hidden states whose shape is same as \(hx\).
- \(cy\) is an updated cell states whose shape is same as \(cx\).
- \(ys\) is a list of Variable. Each element \(ys[t]\) holds hidden states of the last layer corresponding to an input \(xs[t]\). Its shape is \((B_t, N)\) where \(B_t\) is mini-batch size for time \(t\), and \(N\) is size of hidden units. Note that \(B_t\) is the same value as \(xs[t]\).

**Return type** tuple

See also:

`chainer.functions.lstm()`

`chainer.functions.n_step_rnn`

**chainer.functions.n_step_rnn** (\(n\)-layers, dropout_ratio, \(hx, ws, bs, xs, activation='tanh'\))

Stacked Uni-directional RNN function for sequence inputs.

This function calculates stacked Uni-directional RNN with sequences. This function gets an initial hidden state \(h_0\), an initial cell state \(c_0\), an input sequence \(x\), weight matrices \(W\), and bias vectors \(b\). This function calculates hidden states \(h_t\) and \(c_t\) for each time \(t\) from input \(x_t\).

\[
h_t = f(W_0 x_t + W_1 h_{t-1} + b_0 + b_1)
\]

where \(f\) is an activation function.

Weight matrices \(W\) contains two matrices \(W_0\) and \(W_1\). \(W_0\) is a parameter for an input sequence. \(W_1\) is a parameter for a hidden state. Bias matrices \(b\) contains two matrices \(b_0\) and \(b_1\). \(b_0\) is a parameter for an input sequence. \(b_1\) is a parameter for a hidden state.

As the function accepts a sequence, it calculates \(h_t\) for all \(t\) with one call. Two weight matrices and two bias vectors are required for each layer. So, when \(S\) layers exist, you need to prepare \(2S\) weight matrices and \(2S\) bias vectors.

If the number of layers \(n\)-layers is greater than 1, input of \(k\)-th layer is hidden state \(h_{-t}\) of \(k-1\)-th layer. Note that all input variables except first layer may have different shape from the first layer.
Warning: train and use_cudnn arguments are not supported anymore since v2. Instead, use chainer.using_config('train', train) and chainer.using_config('use_cudnn', use_cudnn) respectively. See chainer.using_config().

Parameters

- n_layers (int) – Number of layers.
- dropout_ratio (float) – Dropout ratio.
- hx (chainer.Variable) – Variable holding stacked hidden states. Its shape is \((S, B, N)\) where \(S\) is number of layers and is equal to n_layers, \(B\) is mini-batch size, and \(N\) is dimension of hidden units.
- ws (list of list of chainer.Variable) – Weight matrices. \(ws[i]\) represents weights for i-th layer. Each \(ws[i]\) is a list containing two matrices. \(ws[i][j]\) is corresponding with \(W_j\) in the equation. Only \(ws[0][j]\) where \(0 \leq j < 1\) is \((I, N)\) shape as they are multiplied with input variables. All other matrices has \((N, N)\) shape.
- bs (list of list of chainer.Variable) – Bias vectors. \(bs[i]\) represents biases for i-th layer. Each \(bs[i]\) is a list containing two vectors. \(bs[i][j]\) is corresponding with \(b_j\) in the equation. Shape of each matrix is \((N,)\) where \(N\) is dimension of hidden units.
- xs (list of chainer.Variable) – A list of Variable holding input values. Each element \(xs[t]\) holds input value for time \(t\). Its shape is \((B_t, I)\), where \(B_t\) is mini-batch size for time \(t\) and \(I\) is size of input units. Note that this functions supports variable length sequences. When sequences has different lengths, sort sequences in descending order by length, and transpose the sorted sequence. transpose_sequence() transpose a list of Variable holding sequence. So \(xs\) needs to satisfy \(xs[t].shape[0] \geq xs[t + 1].shape[0]\).
- activation (str) – Activation function name. Please select tanh or relu.

Returns

This functions returns a tuple containing three elements, \(hy\) and \(ys\).

- \(hy\) is an updated hidden states whose shape is same as \(hx\).
- \(ys\) is a list of Variable. Each element \(ys[t]\) holds hidden states of the last layer corresponding to an input \(xs[t]\). Its shape is \((B_t, N)\) where \(B_t\) is mini-batch size for time \(t\), and \(N\) is size of hidden units. Note that \(B_t\) is the same value as \(xs[t]\).

Return type tuple

Evaluation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.functions.accuracy</td>
<td>Computes multiclass classification accuracy of the minibatch.</td>
</tr>
<tr>
<td>chainer.functions.binary_accuracy</td>
<td>Computes binary classification accuracy of the minibatch.</td>
</tr>
<tr>
<td>chainer.functions.classification_summary</td>
<td>Calculates Precision, Recall, F beta Score, and support.</td>
</tr>
<tr>
<td>chainer.functions.r2_score</td>
<td>Computes R^2(coefficient of determination) regression score function.</td>
</tr>
</tbody>
</table>
chainer.functions.accuracy

chainer.functions.accuracy(y, t, ignore_label=None)
Computes multiclass classification accuracy of the minibatch.

Parameters

- **y** *(Variable or numpy.ndarray or cupy.ndarray)* – Array whose (i, j, k, ...)-th element indicates the score of the class j at the (i, k, ...)-th sample. The prediction label \( \hat{t} \) is calculated by the formula \( \hat{t}(i,k,...) = \arg\max_j y(i,j,k,...) \).
- **t** *(Variable or numpy.ndarray or cupy.ndarray of numpy.int32)* – Array of ground truth labels.
- **ignore_label** *(int or None)* – Skip calculating accuracy if the true label is ignore_label.

Returns A variable holding a scalar array of the accuracy.

Return type *Variable*

Note: This function is non-differentiable.

Example

We show the most common case, when y is the two dimensional array.

```python
>>> y = np.array([[0.1, 0.7, 0.2],
                    [8.0, 1.0, 2.0],
                    [-8.0, 1.0, 2.0],
                    [-8.0, -1.0, -2.0]],
               'i')
>>> t = np.array([1, 0, 2, 1], 'i')
>>> F.accuracy(y, t).data # 100% accuracy because all samples are correct
array(1.0)
>>> t = np.array([1, 0, 0, 0], 'i')
>>> F.accuracy(y, t).data # 50% accuracy because 1st and 2nd samples are correct.
array(0.5)
>>> F.accuracy(y, t, ignore_label=0).data # 100% accuracy because of ignoring the
array(1.0)
```

chainer.functions.binary_accuracy

chainer.functions.binary_accuracy(y, t)
Computes binary classification accuracy of the minibatch.

Parameters

- **y** *(Variable)* – Variable holding a matrix whose i-th element indicates the score of positive at the i-th example.
- **t** *(Variable)* – Variable holding an int32 vector of ground truth labels. If \( t[i] == -1 \), corresponding \( x[i] \) is ignored. Accuracy is zero if all ground truth labels are \(-1\).

Returns A variable holding a scalar array of the accuracy.

Return type *Variable*
chainer.functions.classification_summary

chainer.functions.classification_summary(y, t, label_num=None, beta=1.0, ignore_label=-1)

Calculates Precision, Recall, F beta Score, and support.
This function calculates the following quantities for each class.

• Precision: \( \frac{tp}{tp+fp} \)
• Recall: \( \frac{tp}{tp+tn} \)
• F beta Score: The weighted harmonic average of Precision and Recall.
• Support: The number of instances of each ground truth label.

Here, \( tp \), \( fp \), and \( tn \) stand for the number of true positives, false positives, and true negative, respectively.

label_num specifies the number of classes, that is, each value in \( t \) must be an integer in the range of \([0, label_num)\). If label_num is None, this function regards label_num as a maximum of in \( t \) plus one.

ignore_label determines which instances should be ignored. Specifically, instances with the given label are not taken into account for calculating the above quantities. By default, it is set to -1 so that all instances are taken into consideration, as labels are supposed to be non-negative integers. Setting ignore_label to a non-negative integer less than label_num is illegal and yields undefined behavior. In the current implementation, it arises RuntimeWarning and ignore_label-th entries in output arrays do not contain correct quantities.

Parameters

• \( y \) (Variable) – Variable holding a vector of scores.
• \( t \) (Variable) – Variable holding a vector of ground truth labels.
• \( label\_num \) (int) – The number of classes.
• \( beta \) (float) – The parameter which determines the weight of precision in the F-beta score.
• \( ignore\_label \) (int) – Instances with this label are ignored.

Returns 4-tuple of ~chainer.Variable of size \((label\_num,)\). Each element represents precision, recall, F beta score, and support of this minibatch.

chainer.functions.r2_score

chainer.functions.r2_score(pred, true, sample_weight=None, multioutput='uniform_average')
Computes R^2(coefficient of determination) regression score function.

Parameters

• \( pred \) (Variable) – Variable holding a vector, matrix or tensor of estimated target values.
• \( true \) (Variable) – Variable holding a vector, matrix or tensor of correct target values.
• \( sample\_weight \) – This argument is for compatibility with scikit-learn’s implementation of r2_score. Current implementation admits None only.
- **multioutput** *(string)* – ['uniform_average', 'raw_values']. If ‘uniform_average’, this function returns an average of $R^2$ score of multiple output. If ‘raw_average’, this function return a set of $R^2$ score of multiple output.

**Returns** A Variable holding a scalar array of the $R^2$ score if ‘multioutput’ is ‘uniform_average’ or a vector of $R^2$ scores if ‘multioutput’ is ‘raw_values’.

**Return type** Variable

**Note:** This function is non-differentiable.

### Loss functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>chainer.functions.bernoulli_nll</strong></td>
<td>Computes the negative log-likelihood of a Bernoulli distribution.</td>
</tr>
<tr>
<td><strong>chainer.functions.black_out</strong></td>
<td>BlackOut loss function.</td>
</tr>
<tr>
<td><strong>chainer.functions.connectionist_temporal_classification</strong></td>
<td>Connectionist Temporal Classification loss function.</td>
</tr>
<tr>
<td><strong>chainer.functions.contrastive</strong></td>
<td>Computes contrastive loss.</td>
</tr>
<tr>
<td><strong>chainer.functions.crf1d</strong></td>
<td>Calculates negative log-likelihood of linear-chain CRF.</td>
</tr>
<tr>
<td><strong>chainer.functions.argmax_crf1d</strong></td>
<td>Computes a state that maximizes a joint probability of the given CRF.</td>
</tr>
<tr>
<td><strong>chainer.functions.cross_covariance</strong></td>
<td>Computes the sum-squared cross-covariance penalty between $y$ and $z$.</td>
</tr>
<tr>
<td><strong>chainer.functions.gaussian_kl_divergence</strong></td>
<td>Computes the KL-divergence of Gaussian variables from the standard one.</td>
</tr>
<tr>
<td><strong>chainer.functions.gaussian_nll</strong></td>
<td>Computes the negative log-likelihood of a Gaussian distribution.</td>
</tr>
<tr>
<td><strong>chainer.functions.hinge</strong></td>
<td>Computes the hinge loss for a one-of-many classification task.</td>
</tr>
<tr>
<td><strong>chainer.functions.huber_loss</strong></td>
<td>Loss function which is less sensitive to outliers in data than MSE.</td>
</tr>
<tr>
<td><strong>chainer.functions.mean_absolute_error</strong></td>
<td>Mean absolute error function.</td>
</tr>
<tr>
<td><strong>chainer.functions.mean_squared_error</strong></td>
<td>Mean squared error function.</td>
</tr>
<tr>
<td><strong>chainer.functions.negative_sampling</strong></td>
<td>Negative sampling loss function.</td>
</tr>
<tr>
<td><strong>chainer.functions.triplet</strong></td>
<td>Computes triplet loss.</td>
</tr>
</tbody>
</table>

**chainer.functions.bernoulli_nll**

**chainer.functions.bernoulli_nll** *(x, y, reduce='sum')*

Computes the negative log-likelihood of a Bernoulli distribution.

This function calculates the negative log-likelihood of a Bernoulli distribution.

\[- \log B(x; p) = - \sum_i \{ x_i \log(p_i) + (1 - x_i) \log(1 - p_i) \},\]

where $p = \sigma(y)$, $\sigma(\cdot)$ is a sigmoid function, and $B(x; p)$ is a Bernoulli distribution.

---

3.4. Standard Function implementations 175
The output is a variable whose value depends on the value of the option `reduce`. If it is 'no', it holds the elementwise loss values. If it is 'sum', loss values are summed up.

Note: As this function uses a sigmoid function, you can pass a result of fully-connected layer (that means `Linear`) to this function directly.

**Parameters**
- \( x \) (**Variable** or `numpy.ndarray` or `cupy.ndarray`) – Input variable.
- \( y \) (**Variable** or `numpy.ndarray` or `cupy.ndarray`) – A variable representing the parameter of Bernoulli distribution.
- \( \text{reduce} \) (str) – Reduction option. Its value must be either 'sum' or 'no'. Otherwise, `ValueError` is raised.

**Returns** A variable representing the negative log-likelihood. If `reduce` is 'no', the output variable holds an array whose shape is same as one of (hence both of) input variables. If it is 'sum', the output variable holds a scalar value.

**Return type** **Variable**

**chainer.functions.black_out**

```python
chainer.functions.black_out(x, t, W, samples, reduce='mean')
```

BlackOut loss function.

BlackOut loss function is defined as

\[
-\log(p(t)) - \sum_{s \in S} \log(1 - p(s)),
\]

where \( t \) is the correct label, \( S \) is a set of negative examples and \( p(\cdot) \) is likelihood of a given label. And, \( p \) is defined as

\[
p(y) = \frac{\exp(W_y^{\top} x)}{\sum_{s \in \text{samples}} \exp(W_s^{\top} x)}.
\]

The output is a variable whose value depends on the value of the option `reduce`. If it is 'no', it holds the no loss values. If it is 'mean', this function takes a mean of loss values.

**Parameters**
- \( x \) (**Variable**) – Batch of input vectors. Its shape should be \((N, D)\).
- \( t \) (**Variable**) – Vector of ground truth labels. Its shape should be \((N,)\). Each elements \( v \) should satisfy \( 0 \geq v \geq V \) or \(-1\) where \( V \) is the number of label types.
- \( W \) (**Variable**) – Weight matrix. Its shape should be \((V, D)\)
- \( \text{samples} \) (**Variable**) – Negative samples. Its shape should be \((N, S)\) where \( S \) is the number of negative samples.
- \( \text{reduce} \) (str) – Reduction option. Its value must be either 'no' or 'mean'. Otherwise, `ValueError` is raised.

**Returns** A variable object holding loss value(s). If `reduce` is 'no', the output variable holds an array whose shape is \((N,)\). If it is 'mean', it holds a scalar.

**Return type** **Variable**
See: BlackOut: Speeding up Recurrent Neural Network Language Models With Very Large Vocabularies
See also: BlackOut.

\texttt{chainer.functions.connectionist\_temporal\_classification}

\begin{verbatim}
chainer.functions.connectionist_temporal_classification(x, t, blank_symbol,
    input_length=None, label_length=None, reduce='mean')
\end{verbatim}

Connectionist Temporal Classification loss function.

Connectionist Temporal Classification (CTC) [Graves2006] is a loss function of sequence labeling where the alignment between the inputs and target is unknown. See also [Graves2012]

The output is a variable whose value depends on the value of the option \texttt{reduce}. If it is 'no', it holds the samplewise loss values. If it is 'mean', it takes the mean of loss values.

Parameters

- \texttt{x (sequence of Variable)} – RNN output at each time. \texttt{x} must be a list of \texttt{Variable}s. Each element of \texttt{x}, \texttt{x[i]} is a \texttt{Variable} representing output of RNN at time \texttt{i}.
- \texttt{t (Variable)} – Expected label sequence.
- \texttt{blank\_symbol (int)} – Index of blank\_symbol. This value must be non-negative.
- \texttt{input\_length (Variable)} – Length of valid sequence for each of mini batch \texttt{x} (optional). If \texttt{input\_length} is skipped, it regards that all of \texttt{x} is valid input.
- \texttt{label\_length (Variable)} – Length of valid sequence for each of mini batch \texttt{t} (optional). If \texttt{label\_length} is skipped, it regards that all of \texttt{t} is valid input.
- \texttt{reduce (str)} – Reduction option. Its value must be either 'mean' or 'no'. Otherwise, \texttt{ValueError} is raised.

Returns A variable holding a scalar value of the CTC loss. If \texttt{reduce} is 'no', the output variable holds array whose shape is \((B,)\) where \(B\) is the number of samples. If it is 'mean', it holds a scalar.

Return type \texttt{Variable}

Note: You need to input \texttt{x} without applying to activation functions (e.g. softmax function), because this function applies softmax functions to \texttt{x} before calculating CTC loss to avoid numerical limitations. You also need to apply softmax function to forwarded values before you decode it.

Note: This function is differentiable only by \texttt{x}.

Note: This function supports (batch, sequence, 1-dimensional input)-data.
chainer.functions.contrastive

chainer.functions.contrastive(x0, x1, y, margin=1, reduce='mean')

Computes contrastive loss.

It takes a pair of samples and a label as inputs. The label is 1 when those samples are similar, or 0 when they are dissimilar.

Let $N$ and $K$ denote mini-batch size and the dimension of input variables, respectively. The shape of both input variables $x_0$ and $x_1$ should be $(N, K)$. The loss value of the $n$-th sample pair $L_n$ is

$$L_n = \frac{1}{2} \left( y_n d_n^2 + (1 - y_n) \max(margin - d_n, 0)^2 \right)$$

where $d_n = \|x_{0n} - x_{1n}\|_2$, $x_{0n}$ and $x_{1n}$ are $n$-th K-dimensional vectors of $x_0$ and $x_1$.

The output is a variable whose value depends on the value of the option reduce. If it is 'no', it holds the elementwise loss values. If it is 'mean', this function takes a mean of loss values.

Parameters

- x0 (Variable) – The first input variable. The shape should be $(N, K)$, where $N$ denotes the mini-batch size, and $K$ denotes the dimension of $x_0$.
- x1 (Variable) – The second input variable. The shape should be the same as $x_0$.
- y (Variable) – Labels. All values should be 0 or 1. The shape should be $(N,)$, where $N$ denotes the mini-batch size.
- margin (float) – A parameter for contrastive loss. It should be positive value.
- reduce (str) – Reduction option. Its value must be either 'mean' or 'no'. Otherwise, ValueError is raised.

Returns

A variable holding the loss value(s) calculated by the above equation. If reduce is 'no', the output variable holds array whose shape is same as one of (hence both of) input variables. If it is 'mean', the output variable holds a scalar value.

Return type Variable

Note: This cost can be used to train siamese networks. See Learning a Similarity Metric Discriminatively, with Application to Face Verification for details.

chainer.functions.crf1d

chainer.functions.crf1d(cost, xs, ys, reduce='mean')

Calculates negative log-likelihood of linear-chain CRF.

It takes a transition cost matrix, a sequence of costs, and a sequence of labels. Let $c_{st}$ be a transition cost from a label $s$ to a label $t$, $x_{it}$ be a cost of a label $t$ at position $i$, and $y_i$ be an expected label at position $i$. The negative log-likelihood of linear-chain CRF is defined as

$$L = - \left( \sum_{i=1}^{l} x_{iy_i} + \sum_{i=1}^{l-1} c_{y_i y_{i+1}} - \log(Z) \right),$$

where $l$ is the length of the input sequence and $Z$ is the normalizing constant called partition function.
Note: When you want to calculate the negative log-likelihood of sequences which have different lengths, sort the sequences in descending order of lengths and transpose the sequences. For example, you have three input sequences:

```python
>>> a1 = a2 = a3 = a4 = np.random.uniform(-1, 1, 3).astype('f')
>>> b1 = b2 = b3 = np.random.uniform(-1, 1, 3).astype('f')
>>> c1 = c2 = np.random.uniform(-1, 1, 3).astype('f')
>>> a = [a1, a2, a3, a4]
>>> b = [b1, b2, b3]
>>> c = [c1, c2]
```

where a1 and all other variables are arrays with \((K,)\) shape. Make a transpose of the sequences:

```python
>>> x1 = np.stack([a1, b1, c1])
>>> x2 = np.stack([a2, b2, c2])
>>> x3 = np.stack([a3, b3])
>>> x4 = np.stack([a4])
```

and make a list of the arrays:

```python
>>> xs = [x1, x2, x3, x4]
```

You need to make label sequences in the same fashion. And then, call the function:

```python
>>> cost = chainer.Variable(
...   np.random.uniform(-1, 1, (3, 3)).astype('f'))
>>> ys = [np.zeros(x.shape[0:1], dtype='i') for x in xs]
>>> loss = F.crf1d(cost, xs, ys)
```

It calculates mean of the negative log-likelihood of the three sequences.

The output is a variable whose value depends on the value of the option `reduce`. If it is 'no', it holds the elementwise loss values. If it is 'mean', it holds mean of the loss values.

Parameters

- `cost (Variable)` – A \(K \times K\) matrix which holds transition cost between two labels, where \(K\) is the number of labels.
- `xs (list of Variable)` – Input vector for each label. \(\text{len}(xs)\) denotes the length of the sequence, and each \(\text{Variable}\) holds a \(B \times K\) matrix, where \(B\) is mini-batch size, \(K\) is the number of labels. Note that \(B\)'s in all the variables are not necessary the same, i.e., it accepts the input sequences with different lengths.
- `ys (list of Variable)` – Expected output labels. It needs to have the same length as \(xs\). Each \(\text{Variable}\) holds a \(B\) integer vector. When \(x\) in \(xs\) has the different \(B\), corresponding \(y\) has the same \(B\). In other words, \(ys\) must satisfy \(ys[i].\text{shape} == xs[i].\text{shape}[0:1]\) for all \(i\).
- `reduce (str)` – Reduction option. Its value must be either 'mean' or 'no'. Otherwise, \(\text{ValueError}\) is raised.

Returns

- A variable holding the average negative log-likelihood of the input sequences.

Return type `Variable`

**chainer.functions.argmax_crf1d**

chainer.functions.argmax_crf1d(cost, xs)
Computes a state that maximizes a joint probability of the given CRF.

**Parameters**

- **cost** (Variable) – A $K \times K$ matrix which holds transition cost between two labels, where $K$ is the number of labels.
- **xs** (list of Variable) – Input vector for each label. len(xs) denotes the length of the sequence, and each Variable holds a $B \times K$ matrix, where $B$ is mini-batch size, $K$ is the number of labels. Note that $B$ s in all the variables are not necessary the same, i.e., it accepts the input sequences with different lengths.

**Returns**

A tuple of Variable object s and a list ps. The shape of s is $(B,)$, where $B$ is the mini-batch size. i-th element of s, s[i], represents log-likelihood of i-th data. ps is a list of numpy.ndarray or cupy.ndarray, and denotes the state that maximizes the point probability. len(ps) is equal to len(xs), and shape of each ps[i] is the mini-batch size of the corresponding xs[i]. That means, ps[i].shape == xs[i].shape[0:1].

**Return type** tuple

**chainer.functions.cross_covariance**

chainer.functions.cross_covariance(y, z, reduce='half_squared_sum')
Computes the sum-squared cross-covariance penalty between y and z.

The output is a variable whose value depends on the value of the option reduce. If it is 'no', it holds the covariant matrix that has as many rows (resp. columns) as the dimension of y (resp.z). If it is 'half_squared_sum', it holds the half of the Frobenius norm (i.e. L2 norm of a matrix flattened to a vector) of the covarianct matrix.

**Parameters**

- **y** (Variable) – Variable holding a matrix where the first dimension corresponds to the batches.
- **z** (Variable) – Variable holding a matrix where the first dimension corresponds to the batches.
- **reduce** (str) – Reduction option. Its value must be either 'half_squared_sum' or 'no'. Otherwise, ValueError is raised.

**Returns** A variable holding the cross covariance loss. If reduce is 'no', the output variable holds 2-dimensional array matrix of shape $(M, N)$ where $M$ (resp. $N$) is the number of columns of y (resp. z). If it is 'half_squared_sum', the output variable holds a scalar value.

**Return type** Variable
chainer.functions.gaussian_kl_divergence

chainer.functions.gaussian_kl_divergence(mean, ln_var, reduce='sum')

Computes the KL-divergence of Gaussian variables from the standard one.

Given two variable mean representing \(\mu\) and ln_var representing \(\log(\sigma^2)\), this function calculates the KL-divergence in elementwise manner between the given multi-dimensional Gaussian \(N(\mu, S)\) and the standard Gaussian \(N(0, I)\)

\[
D_{\text{KL}}(N(\mu, S)\|N(0, I)),
\]

where \(S\) is a diagonal matrix such that \(S_{ii} = \sigma^2_i\) and \(I\) is an identity matrix.

The output is a variable whose value depends on the value of the option reduce. If it is 'no', it holds the elementwise loss values. If it is 'sum', loss values are summed up.

**Parameters**

- mean (Variable or numpy.ndarray or cupy.ndarray) – A variable representing mean of given gaussian distribution, \(\mu\).
- ln_var (Variable or numpy.ndarray or cupy.ndarray) – A variable representing logarithm of variance of given gaussian distribution, \(\log(\sigma^2)\).
- reduce (str) – Reduction option. Its value must be either 'sum' or 'no'. Otherwise, ValueError is raised.

**Returns**

A variable representing KL-divergence between given gaussian distribution and the standard gaussian. If reduce is 'no', the output variable holds array whose shape is same as one of (hence both of) input variables. If it is 'sum', the output variable holds a scalar value.

**Return type** Variable

chainer.functions.gaussian_nll

chainer.functions.gaussian_nll(x, mean, ln_var, reduce='sum')

Computes the negative log-likelihood of a Gaussian distribution.

Given two variable mean representing \(\mu\) and ln_var representing \(\log(\sigma^2)\), this function computes in elementwise manner the negative log-likelihood of \(x\) on a Gaussian distribution \(N(\mu, S)\),

\[
-\log N(x; \mu, \sigma^2) = \log \left( \sqrt{(2\pi)^D |S|} \right) + \frac{1}{2} (x - \mu)^\top S^{-1} (x - \mu),
\]

where \(D\) is a dimension of \(x\) and \(S\) is a diagonal matrix where \(S_{ii} = \sigma^2_i\).

The output is a variable whose value depends on the value of the option reduce. If it is 'no', it holds the elementwise loss values. If it is 'sum', loss values are summed up.

**Parameters**

- x (Variable or numpy.ndarray or cupy.ndarray) – Input variable.
- mean (Variable or numpy.ndarray or cupy.ndarray) – A variable representing mean of a Gaussian distribution, \(\mu\).
• **ln_var** *(Variable or numpy.ndarray or cupy.ndarray)* – A variable representing logarithm of variance of a Gaussian distribution, \( \log(\sigma^2) \).

• **reduce** *(str)* – Reduction option. Its value must be either 'sum' or 'no'. Otherwise, ValueError is raised.

**Returns** A variable representing the negative log-likelihood. If reduce is 'no', the output variable holds array whose shape is same as one of (hence both of) input variables. If it is 'sum', the output variable holds a scalar value.

**Return type** *Variable*

### chainer.functions.hinge

```python
chainer.functions.hinge(x, t, norm='L1', reduce='mean')
```

Computes the hinge loss for a one-of-many classification task.

\[
L = \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} [\max(0, 1 - \delta\{t_n = k\}x_{nk})]^p
\]

where \( N \) denotes the batch size and \( K \) is the number of classes of interest,

\[
\delta\{\text{condition}\} = \begin{cases} 
1 & \text{if condition is true} \\
-1 & \text{otherwise},
\end{cases}
\]

and

\[
p = \begin{cases} 
1 & \text{if norm = L1} \\
2 & \text{if norm = L2}.
\end{cases}
\]

The output is a variable whose value depends on the value of the option reduce. If it is 'no', it holds the elementwise loss values. If it is 'mean', it takes the mean of loss values.

**Parameters**

• **x** *(Variable)* – Input variable. The shape of \( x \) should be \((N, K)\).

• **t** *(Variable)* – The \( N \)-dimensional label vector with values \( t_n \in \{0, 1, 2, \ldots, K - 1\} \). The shape of \( t \) should be \((N,)\).

• **norm** *(string)* – Specifies norm type. Either 'L1' or 'L2' is acceptable.

• **reduce** *(str)* – Reduction option. Its value must be either 'mean' or 'no'. Otherwise, ValueError is raised.

**Returns** A variable object holding a scalar array of the hinge loss \( L \). If reduce is 'no', the output variable holds array whose shape is same as one of (hence both of) input variables. If it is 'mean', the output variable holds a scalar value.

**Return type** *Variable*

### chainer.functions.huber_loss

```python
chainer.functions.huber_loss(x, t, delta, reduce='sum_along_second_axis')
```

Loss function which is less sensitive to outliers in data than MSE.

Loss function which is less sensitive to outliers in data than MSE.
\[ a = x - t \]

and

\[ L_\delta(a) = \begin{cases} 
\frac{1}{2}a^2 & \text{if } |a| \leq \delta \\
\delta(|a| - \frac{1}{2}\delta) & \text{otherwise},
\end{cases} \]

The output is a variable whose value depends on the value of the option reduce. If it is 'no', it holds the elementwise loss values. If it is 'sum_along_second_axis', loss values are summed up along the second axis (i.e. axis=1).

**Parameters**

- \( x \) (Variable) – Input variable. The shape of \( x \) should be \((N, K)\).
- \( t \) (Variable) – Target variable for regression. The shape of \( t \) should be \((N, K)\).
- \( \delta \) (float) – Constant variable for huber loss function as used in definition.
- \( \text{reduce} \) (str) – Reduction option. Its value must be either 'sum_along_second_axis' or 'no'. Otherwise, ValueError is raised.

**Returns** A variable object holding a scalar array of the huber loss \( L_\delta \). If reduce is 'no', the output variable holds array whose shape is same as one of (hence both of) input variables. If it is 'sum_along_second_axis', the shape of the array is same as the input variables, except the second axis is removed.

**Return type** Variable

See: Huber loss - Wikipedia.

**chainer.functions.mean_absolute_error**

\[ \text{chainer.functions.mean_absolute_error}(x0, x1) \]

Mean absolute error function.

This function computes mean absolute error between two variables. The mean is taken over the minibatch.

**chainer.functions.mean_squared_error**

\[ \text{chainer.functions.mean_squared_error}(x0, x1) \]

Mean squared error function.

This function computes mean squared error between two variables. The mean is taken over the minibatch. Note that the error is not scaled by 1/2.

**Parameters**

- \( x0 \) (Variable or numpy.ndarray or cupy.ndarray) – Input variable.
- \( x1 \) (Variable or numpy.ndarray or cupy.ndarray) – Input variable.

**Returns** A variable holding an array representing the mean squared error of two inputs.

**Return type** Variable
chainer.functions.negative_sampling

chainer.functions.negative_sampling(x, t, W, sampler, sample_size, reduce='sum')
Negative sampling loss function.

In natural language processing, especially language modeling, the number of words in a vocabulary can be very large. Therefore, you need to spend a lot of time calculating the gradient of the embedding matrix.

By using the negative sampling trick you only need to calculate the gradient for a few sampled negative examples.

The objective function is below:

\[
f(x, p) = \log \sigma(x^T w_p) + k E_{i \sim P(i)}[\log \sigma(-x^T w_i)],
\]

where \( \sigma(\cdot) \) is a sigmoid function, \( w_i \) is the weight vector for the word \( i \), and \( p \) is a positive example. It is approximated with \( k \) examples \( N \) sampled from probability \( P(i) \), like this:

\[
f(x, p) \approx \log \sigma(x^T w_p) + \sum_{n \in N} \log \sigma(-x^T w_n).
\]

Each sample of \( N \) is drawn from the word distribution \( P(w) \). This is calculated as \( P(w) = \frac{1}{Z} c(w)\alpha \), where \( c(w) \) is the unigram count of the word \( w \), \( \alpha \) is a hyper-parameter, and \( Z \) is the normalization constant.

Parameters

- \( x \) (Variable) – Batch of input vectors.
- \( t \) (Variable) – Vector of ground truth labels.
- \( W \) (Variable) – Weight matrix.
- \( \text{sampler} \) (FunctionType) – Sampling function. It takes a shape and returns an integer array of the shape. Each element of this array is a sample from the word distribution. A WalkerAlias object built with the power distribution of word frequency is recommended.
- \( \text{sample_size} \) (int) – Number of samples.
- \( \text{reduce} \) (str) – Reduction option. Its value must be either 'sum' or 'no'. Otherwise, ValueError is raised.

Returns A variable holding the loss value(s) calculated by the above equation. If \( \text{reduce} \) is 'no', the output variable holds array whose shape is same as one of (hence both of) input variables. If it is 'sum', the output variable holds a scalar value.

Return type Variable

See: Distributed Representations of Words and Phrases and their Compositionality

See also:
NegativeSampling.

chainer.functions.sigmoid_cross_entropy

chainer.functions.sigmoid_cross_entropy(x, t, normalize=True, reduce='mean')
Computes cross entropy loss for pre-sigmoid activations.

Parameters

- \( x \) (Variable) – A variable object holding a matrix whose (i, j)-th element indicates the unnormalized log probability of the j-th unit at the i-th example.
• \(t\) (Variable) – Variable holding an int32 vector of ground truth labels. If \(t[i] == -1\), corresponding \(x[i]\) is ignored. Loss is zero if all ground truth labels are -1.

• normalize (bool) – Variable holding a boolean value which determines the normalization constant. If true, this function normalizes the cross entropy loss across all instances. If else, it only normalizes along a batch size.

• reduce (str) – Variable holding a str which determines whether to reduce the shape of the input. If it is 'mean', it computes the sum of cross entropy and normalize it according to normalize option. If is is 'no', this function computes cross entropy for each instance and does not normalize it (normalize option is ignored). In this case, the loss value of the ignored instance, which has -1 as its target value, is set to 0.

**Returns** A variable object holding an array of the cross entropy. If reduce is 'mean', it is a scalar array. If reduce is 'no', the shape is same as \(x\).

**Return type** Variable

Note: This function is differentiable only by \(x\).

### chainer.functions.softmax_cross_entropy

chainer.functions.softmax_cross_entropy \((x, t, \text{normalize=True, cache_score=True, class_weight=None, ignore_label=-1, reduce='mean')}\)

Computes cross entropy loss for pre-softmax activations.

**Parameters**

• \(x\) (Variable) – Variable holding a multidimensional array whose element indicates unnormalized log probability: the first axis of the variable represents the number of samples, and the second axis represents the number of classes. While this function computes a usual softmax cross entropy if the number of dimensions is equal to 2, it computes a cross entropy of the replicated softmax if the number of dimensions is greater than 2.

• \(t\) (Variable) – Variable holding an int32 vector of ground truth labels. If \(t[i] == \text{ignore_label}\), corresponding \(x[i]\) is ignored.

• normalize (bool) – If True, this function normalizes the cross entropy loss across all instances. If False, it only normalizes along a batch size.

• cache_score (bool) – When it is True, the function stores result of forward computation to use it on backward computation. It reduces computational cost though consumes more memory.

• class_weight (ndarray or ndarray) – An array that contains constant weights that will be multiplied with the loss values along with the second dimension. The shape of this array should be \((x.\text{shape}[1],)\). If this is not None, each class weight \(\text{class_weight}[i]\) is actually multiplied to \(y[:, i]\) that is the corresponding log-softmax output of \(x\) and has the same shape as \(x\) before calculating the actual loss value.

• ignore_label (int) – Label value you want to ignore. Its default value is -1. See description of the argument \(t\).

• reduce (str) – A string that determines whether to reduce the loss values. If it is 'mean', it computes the sum of the individual cross entropy and normalize it according
to normalize option. If it is 'no', this function computes cross entropy for each instance and does not normalize it (normalize option is ignored). In this case, the loss value of the ignored instance, which has ignore_label as its target value, is set to 0.

**Returns** A variable holding a scalar array of the cross entropy loss. If reduce is 'mean', it is a scalar array. If reduce is 'no', the shape is same as that of x.

**Return type** *Variable*

---

**Note:** This function is differentiable only by x.

---

### chainer.functions.squared_error

**chainer.functions.squared_error**(x0, x1)

Squared error function.

This function computes the squared error between two variables:

\[
(x_0 - x_1)^2
\]

where operation is done in elementwise manner. Note that the error is not scaled by 1/2:

**Parameters**

- **x0** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Input variable.
- **x1** (*Variable* or *numpy.ndarray* or *cupy.ndarray*) – Input variable.

**Returns** A variable holding an array representing the squared error of two inputs.

**Return type** *Variable*

---

### chainer.functions.triplet

**chainer.functions.triplet**(anchor, positive, negative, margin=0.2, reduce='mean')

Computes triplet loss.

It takes a triplet of variables as inputs, a, p and n: anchor, positive example and negative example respectively. The triplet defines a relative similarity between samples. Let N and K denote mini-batch size and the dimension of input variables, respectively. The shape of all input variables should be (N, K).

\[
L(a, p, n) = \frac{1}{N} \left( \sum_{i=1}^{N} \max\{d(a_i, p_i) - d(a_i, n_i) + \text{margin}, 0\} \right)
\]

where \(d(x_i, y_i) = \|x_i - y_i\|_2^2\).

The output is a variable whose value depends on the value of the option reduce. If it is 'no', it holds the elementwise loss values. If it is 'mean', this function takes a mean of loss values.

**Parameters**

- **anchor** (*Variable*) – The anchor example variable. The shape should be (N, K), where N denotes the minibatch size, and K denotes the dimension of the anchor.
- **positive** (*Variable*) – The positive example variable. The shape should be the same as anchor.
• **negative** *(Variable)* – The negative example variable. The shape should be the same as anchor.

• **margin** *(float)* – A parameter for triplet loss. It should be a positive value.

• **reduce** *(str)* – Reduction option. Its value must be either 'mean' or 'no'. Otherwise, `ValueError` is raised.

**Returns** A variable holding a scalar that is the loss value calculated by the above equation. If `reduce` is 'no', the output variable holds array whose shape is same as one of (hence both of) input variables. If it is 'mean', the output variable holds a scalar value.

**Return type** *Variable*

---

**Note:** This cost can be used to train triplet networks. See Learning Fine-grained Image Similarity with Deep Ranking for details.

---

### Mathematical functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.functions.arccos</code></td>
<td>Elementwise arccosine function.</td>
</tr>
<tr>
<td><code>chainer.functions.arcsin</code></td>
<td>Elementwise arcsine function.</td>
</tr>
<tr>
<td><code>chainer.functions.arctan</code></td>
<td>Elementwise arctangent function.</td>
</tr>
<tr>
<td><code>chainer.functions.argmax</code></td>
<td>Returns index which holds maximum of array elements over a given axis.</td>
</tr>
<tr>
<td><code>chainer.functions.argmin</code></td>
<td>Returns index which holds minimum of array elements over a given axis.</td>
</tr>
<tr>
<td><code>chainer.functions.average</code></td>
<td>Calculate weighted average of array elements over a given axis.</td>
</tr>
<tr>
<td><code>chainer.functions.batch_inv</code></td>
<td>Computes the inverse of a batch of square matrices.</td>
</tr>
<tr>
<td><code>chainer.functions.batch_l2_norm_squared</code></td>
<td>L2 norm (a.k.a.</td>
</tr>
<tr>
<td><code>chainer.functions.batch_matmul</code></td>
<td>Computes the batch matrix multiplications of two sets of arrays.</td>
</tr>
<tr>
<td><code>chainer.functions.bias</code></td>
<td>Elementwise summation with broadcasting.</td>
</tr>
<tr>
<td><code>chainer.functions.ceil</code></td>
<td>Elementwise ceil function.</td>
</tr>
<tr>
<td><code>chainer.functions.clip</code></td>
<td>Clips (limits) elements of input variable.</td>
</tr>
<tr>
<td><code>chainer.functions.cos</code></td>
<td>Elementwise cos function.</td>
</tr>
<tr>
<td><code>chainer.functions.cosh</code></td>
<td>Elementwise hyperbolic cosine function.</td>
</tr>
<tr>
<td><code>chainer.functions.exp</code></td>
<td>Elementwise exponential function.</td>
</tr>
<tr>
<td><code>chainer.functions.fmod</code></td>
<td>Elementwise mod function.</td>
</tr>
<tr>
<td><code>chainer.functions.floor</code></td>
<td>Elementwise floor function.</td>
</tr>
<tr>
<td><code>chainer.functions.identity</code></td>
<td>Just returns input variables.</td>
</tr>
<tr>
<td><code>chainer.functions.inv</code></td>
<td>Computes the inverse of square matrix.</td>
</tr>
<tr>
<td><code>chainer.functions.linear_interpolate</code></td>
<td>Elementwise linear-interpolation function.</td>
</tr>
<tr>
<td><code>chainer.functions.log</code></td>
<td>Elementwise natural logarithm function.</td>
</tr>
<tr>
<td><code>chainer.functions.log10</code></td>
<td>Elementwise logarithm function to the base 10.</td>
</tr>
<tr>
<td><code>chainer.functions.log1p</code></td>
<td>Elementwise natural logarithm plus one function.</td>
</tr>
<tr>
<td><code>chainer.functions.log2</code></td>
<td>Elementwise logarithm function to the base 2.</td>
</tr>
<tr>
<td><code>chainer.functions.logsumexp</code></td>
<td>Log-sum-exp of array elements over a given axis.</td>
</tr>
<tr>
<td><code>chainer.functions.matmul</code></td>
<td>Computes the matrix multiplication of two arrays.</td>
</tr>
<tr>
<td><code>chainer.functions.max</code></td>
<td>Maximum of array elements over a given axis.</td>
</tr>
<tr>
<td><code>chainer.functions.maximum</code></td>
<td>Element-wise maximum of input variables.</td>
</tr>
</tbody>
</table>

---

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.functions.mean</code></td>
<td>Calculate weighted average of array elements over a given axis.</td>
</tr>
<tr>
<td><code>chainer.functions.min</code></td>
<td>Minimum of array elements over a given axis.</td>
</tr>
<tr>
<td><code>chainer.functions.minimum</code></td>
<td>Element-wise minimum of input variables.</td>
</tr>
<tr>
<td><code>chainer.functions.rsqrt</code></td>
<td>Computes elementwise reciprocal of square root of input $x_i$.</td>
</tr>
<tr>
<td><code>chainer.functions.scale</code></td>
<td>Elementwise product with broadcasting.</td>
</tr>
<tr>
<td><code>chainer.functions.sin</code></td>
<td>Elementwise sin function.</td>
</tr>
<tr>
<td><code>chainer.functions.sinh</code></td>
<td>Elementwise hyperbolic sine function.</td>
</tr>
<tr>
<td><code>chainer.functions.sqrt</code></td>
<td>Elementwise square root function.</td>
</tr>
<tr>
<td><code>chainer.functions.square</code></td>
<td>Elementwise square function.</td>
</tr>
<tr>
<td><code>chainer.functions.squared_difference</code></td>
<td>Squared difference of input variables.</td>
</tr>
<tr>
<td><code>chainer.functions.sum</code></td>
<td>Sum of array elements over a given axis.</td>
</tr>
<tr>
<td><code>chainer.functions.tanh</code></td>
<td>Elementwise hyperbolic tangent function.</td>
</tr>
<tr>
<td><code>chainer.functions.tan</code></td>
<td>Elementwise tan function.</td>
</tr>
</tbody>
</table>

**chainer.functions.arccos**

**chainer.functions.arccos**($x$)  
Elementwise arccosine function.

\[ y_i = \arccos x_i. \]

**Parameters** $x$ (Variable) – Input variable.

**Returns** Output variable.

**Return type** Variable

**chainer.functions.arcsin**

**chainer.functions.arcsin**($x$)  
Elementwise arcsine function.

\[ y_i = \arcsin x_i. \]

**Parameters** $x$ (Variable) – Input variable.

**Returns** Output variable.

**Return type** Variable

**chainer.functions.arctan**

**chainer.functions.arctan**($x$)  
Elementwise arctangent function.

\[ y_i = \arctan x_i. \]

**Parameters** $x$ (Variable) – Input variable.

**Returns** Output variable.

**Return type** Variable
chainer.functions.argmax

chainer.functions.argmax(x, axis=None)

Returns index which holds maximum of array elements over a given axis.

Parameters

- x (Variable) – Array to find maximum elements.
- axis (None or int) – Axis over which a max is performed. The default (axis = None)
  is perform a max over all the dimensions of the input array.

Returns Output variable.
Return type Variable

chainer.functions.argmin

chainer.functions.argmin(x, axis=None)

Returns index which holds minimum of array elements over a given axis.

Parameters

- x (Variable) – Array to find minimum elements.
- axis (None or int) – Axis over which a min is performed. The default (axis = None)
  is perform a min over all the dimensions of the input array.

Returns Output variable.
Return type Variable

chainer.functions.average

chainer.functions.average(x, axis=None, weights=None, keepdims=False)

Calculate weighted average of array elements over a given axis.

Parameters

- x (Variable) – Elements to sum.
- axis (None or int) – Axis which the method is performed. With the default (axis = None) it performs a mean over all the dimensions of the input array.
- weights (None or chainer.Variable) – An array holding weights to calculate weighted average. If it is None, all weights are assumed to be one. When axis is None, weights must have the same shape of x. And when axis is int, it must be 1-D array satisfying weights.shape == (x.shape[axis],).
- keepdims (bool) – If True, the specified axes are remained as axes of length one.

Returns Output variable.
Return type Variable

chainer.functions.batch_inv

chainer.functions.batch_inv(a)

Computes the inverse of a batch of square matrices.
Parameters **a** *(Variable)* – Input array to compute the inverse for. Shape of the array should be $(m, n, n)$ where $m$ is the number of matrices in the batch, and $n$ is the dimensionality of a square matrix.

**Returns** Inverse of every matrix in the batch of matrices.

**Return type** *Variable*

chainer.functions.batch_l2_norm_squared

chainer.functions.batch_l2_norm_squared($x$)

L2 norm (a.k.a. Euclidean norm) squared.

This function implements the square of L2 norm on a vector. No reduction along batch axis is done.

**Parameters**
- **x** *(Variable)* – Input variable. The first dimension is assumed to be the *minibatch dimension*. If $x$ has more than two dimensions all but the first dimension are flattened to one dimension.

**Returns** Two dimensional output variable.

**Return type** *Variable*

chainer.functions.batch_matmul

chainer.functions.batch_matmul($a$, $b$, *transa=False, transb=False*)

Computes the batch matrix multiplications of two sets of arrays.

**Parameters**
- **$a$** *(Variable)* – The left operand of the batch matrix multiplications. A 2-D array of shape $(B, N)$ is considered as $B \times N \times 1$ matrices. A 3-D array of shape $(B, M, N)$ is considered as $B \times M \times N$ matrices.
- **$b$** *(Variable)* – The right operand of the batch matrix multiplications. Its array is treated as matrices in the same way as $a$’s array.
- **transa** *(bool)* – If True, transpose each matrix in $a$.
- **transb** *(bool)* – If True, transpose each matrix in $b$.

**Returns**

The result of the batch matrix multiplications as a 3-D array.

**Return type** *Variable*

chainer.functions.bias

chainer.functions.bias($x$, $y$, *axis=1*)

Elementwise summation with broadcasting.

Computes a elementwise summation of two input variables, with the shape of the latter variable broadcasted to match the shape of the former. *axis* is the first axis of the first variable along which the second variable is applied.

The term “broadcasting” here comes from Caffe’s bias layer so the “broadcasting” with the following arguments:
is equivalent to the following numpy broadcasting:

\[
\begin{align*}
x & : 100 \times 3 \times 40 \times 60 \\
y & : 1 \times 3 \times 40 \times 1 \\
axis & : 1
\end{align*}
\]

Note that how the `axis` indicates to which axis of `x` we apply `y`.

**Parameters**
- `x (Variable)` – Input variable to be summed.
- `y (Variable)` – Input variable to sum, broadcasted.
- `axis (int)` – The first axis of `x` along which `y` is applied.

**Returns** Output variable.

**Return type** `Variable`

### chainer.functions.ceil

**chainer.functions.ceil**(x)

Elementwise ceil function.

\[
y_i = \lceil x_i \rceil
\]

**Parameters**
- `x (Variable)` – Input variable.

**Returns** Output variable.

**Return type** `Variable`

### chainer.functions.clip

**chainer.functions.clip**(x, x_min, x_max)

Clips (limits) elements of input variable.

Given an interval `[x_min, x_max]`, elements outside the interval are clipped to the interval edges.

**Parameters**
- `x (Variable)` – Input variable to be clipped.
- `x_min (float)` – Minimum value.
- `x_max (float)` – Maximum value.

**Returns** Output variable.

**Return type** `Variable`

### chainer.functions.cos

**chainer.functions.cos**(x)

Elementwise cos function.
chainer.functions.cosh

chainer.functions.cosh(x)
Elementwise hyperbolic cosine function.

\[ y_i = \cosh x_i. \]

Parameters  
- x (Variable) – Input variable.

Returns  Output variable.

Return type  Variable

chainer.functions.exp

chainer.functions.exp(x)
Elementwise exponential function.

chainer.functions.fmod

chainer.functions.fmod(x, divisor)
Elementwise mod function.

\[ y_i = x_i \mod \text{divisor}. \]

Parameters  
- x (Variable) – Input variable.
- divisor (Variable) – Input divisor.

Returns  Output variable.

Return type  Variable

chainer.functions.floor

chainer.functions.floor(x)
Elementwise floor function.

\[ y_i = \lfloor x_i \rfloor \]

Parameters  
- x (Variable) – Input variable.

Returns  Output variable.

Return type  Variable

chainer.functions.identity

chainer.functions.identity(*inputs)
Just returns input variables.
chainer.functions.inv

chainer.functions.inv(a)
Computes the inverse of square matrix.

Parameters a (Variable) – Input array to compute the inverse for. Shape of the array should be (n, n) where n is the dimensionality of a square matrix.

Returns Matrix inverse of a.

Return type Variable

chainer.functions.linear_interpolate

chainer.functions.linear_interpolate(p, x, y)
Elementwise linear-interpolation function.

This function is defined as

\[ f(p, x, y) = px + (1-p)y. \]

Parameters

• p (Variable) – Input variable.
• x (Variable) – Input variable.
• y (Variable) – Input variable.

Returns Output variable.

Return type Variable

chainer.functions.log

chainer.functions.log(x)
Elementwise natural logarithm function.

chainer.functions.log10

chainer.functions.log10(x)
Elementwise logarithm function to the base 10.

\[ y_i = \log_{10} x_i. \]

Parameters x (Variable) – Input variable.

Returns Output variable.

Return type Variable

chainer.functions.log1p

chainer.functions.log1p(x)
Elementwise natural logarithm plus one function.
**chainer.functions.log2**

**chainer.functions.log2**(x)

Elementwise logarithm function to the base 2.

\[ y_i = \log_2 x_i. \]

**Parameters**
- **x** (**Variable**): Input variable.

**Returns**
Output variable.

**Return type**
**Variable**

---

**chainer.functions.logsumexp**

**chainer.functions.logsumexp**(x, axis=None)

Log-sum-exp of array elements over a given axis.

This function calculates logarithm of sum of exponential of array elements.

\[ y_i = \log \left( \sum_j \exp(x_{ij}) \right) \]

**Parameters**
- **x** (**Variable**): Elements to log-sum-exp.
- **axis** (**None, int, or tuple of int**): Axis which a sum is performed. The default (axis = None) is perform a sum over all the dimensions of the input array.

**Returns**
Output variable.

**Return type**
**Variable**

---

**chainer.functions.matmul**

**chainer.functions.matmul**(a, b, transa=False, transb=False)

Computes the matrix multiplication of two arrays.

**Parameters**
- **a** (**Variable**): The left operand of the matrix multiplication. A 1-D array of shape \((N,)\) is considered as an \(N \times 1\) matrix. A 2-D array of shape \((M, N)\) is considered as an \(M \times N\) matrix.
- **b** (**Variable**): The right operand of the matrix multiplication. Its array is treated as a matrix in the same way as a’s array.
- **transa** (**bool**): If True, transpose a.
- **transb** (**bool**): If True, transpose b.

**Returns**
The result of the matrix multiplication as a 2-D array.

**Return type**
**Variable**
**chainer.functions.max**

chainer.functions.max(x, axis=None, keepdims=False)

Maximum of array elements over a given axis.

**Parameters**

- **x** *(Variable)* – Array to be maximized.
- **axis** *(None, int, or tuple of int)* – Axis over which a max is performed. The default (axis = None) is perform a max over all the dimensions of the input array.

**Returns** Output variable.

**Return type** Variable

**chainer.functions.maximum**

chainer.functions.maximum(x1, x2)

Element-wise maximum of input variables.

**Parameters**

- **x1** *(Variable)* – Input variables to be compared.
- **x2** *(Variable)* – Input variables to be compared.

**Returns** Output variable.

**Return type** Variable

**chainer.functions.mean**

chainer.functions.mean(x, axis=None, weights=None, keepdims=False)

Calculate weighted average of array elements over a given axis.

**Parameters**

- **x** *(Variable)* – Elements to sum.
- **axis** *(None or int)* – Axis which the method is performed. With the default (axis = None) it performs a mean over all the dimensions of the input array.
- **weights** *(None or chainer.Variable)* – An array holding weights to calculate weighted average. If it is None, all weights are assumed to be one. When axis is None, weights must have the same shape of x. And when axis is int, it must be 1-D array satisfying weights.shape == (x.shape[axis],).
- **keepdims** *(bool)* – If True, the specified axes are remained as axes of length one.

**Returns** Output variable.

**Return type** Variable

**chainer.functions.min**

chainer.functions.min(x, axis=None, keepdims=False)

Minimum of array elements over a given axis.

**Parameters**
• \textbf{x (Variable)} – Array to be minimized.
• \textbf{axis (None, int, or tuple of int)} – Axis over which a min is performed. The default (axis = None) is perform a min over all the dimensions of the input array.

\textbf{Returns} Output variable.
\textbf{Return type} \textit{Variable}

\textbf{chainer.functions.minimum}

\texttt{chainer.functions.minimum(x1, x2)}

Element-wise minimum of input variables.

\textbf{Parameters}
• \textbf{x1 (Variable)} – Input variables to be compared.
• \textbf{x2 (Variable)} – Input variables to be compared.

\textbf{Returns} Output variable.
\textbf{Return type} \textit{Variable}

\textbf{chainer.functions.rsqrt}

\texttt{chainer.functions.rsqrt(x)}

Computes elementwise reciprocal of square root of input $x_i$.

\[ y_i = \frac{1}{\sqrt{x_i}}. \]

\textbf{Parameters} \textbf{x (Variable)} – Input variable.
\textbf{Returns} Output variable.
\textbf{Return type} \textit{Variable}

\textbf{See also:} \texttt{sqrt()}

\textbf{chainer.functions.scale}

\texttt{chainer.functions.scale(x, y, axis=1)}

Elementwise product with broadcasting.

Computes a elementwise product of two input variables, with the shape of the latter variable broadcasted to match the shape of the former. \texttt{axis} is the first axis of the first variable along which the second variable is applied.

The term “broadcasting” here comes from Caffe’s scale layer so the “broadcasting” with the following arguments:

\begin{verbatim}
x : 100 x 3 x 40 x 60 y : 3 x 40 axis : 1
\end{verbatim}

is equivalent to the following numpy broadcasting:
\[ x : 100 \times 3 \times 40 \times 60 \]
\[ y : 1 \times 3 \times 40 \times 1 \]

Note that how the axis indicates to which axis of \( x \) we apply \( y \).

Parameters

- \( x \) (Variable) – Input variable to be scaled.
- \( y \) (Variable) – Input variable to scale, broadcasted.
- \( \text{axis} \) (int) – The first axis of \( x \) along which \( y \) is applied.

Returns  Output variable.

Return type  \textit{Variable}

\texttt{chainer.functions.sin}

\texttt{chainer.functions.sin}(x)

Elementwise sin function.

\texttt{chainer.functions.sinh}

\texttt{chainer.functions.sinh}(x)

Elementwise hyperbolic sine function.

\[ y_i = \sinh x_i. \]

Parameters  \( x \) (Variable) – Input variable.

Returns  Output variable.

Return type  \textit{Variable}

\texttt{chainer.functions.sqrt}

\texttt{chainer.functions.sqrt}(x)

Elementwise square root function.

\[ y_i = \sqrt{x_i}. \]

If the value of \( x_i \) is negative, it returns \texttt{Nan} for \( y_i \) respect to underlying numpy and cupy specification.

Parameters  \( x \) (Variable) – Input variable.

Returns  Output variable.

Return type  \textit{Variable}

\texttt{chainer.functions.square}

\texttt{chainer.functions.square}(x)

Elementwise square function.

\[ y_i = x_i^2. \]
Parameters \( \mathbf{x} \) (chainer.Variable or numpy.ndarray or cupy.ndarray) – Input variable.

Returns Output variable.

Return type Variable

**chainer.functions.squared_difference**

chainer.functions.squared_difference\((\mathbf{x}_1, \mathbf{x}_2)\)

Squared difference of input variables.

Parameters

- \( \mathbf{x}_1 \) (Variable) – Input variables to be compared.
- \( \mathbf{x}_2 \) (Variable) – Input variables to be compared.

Returns \((\mathbf{x}_1 - \mathbf{x}_2)^2\) element-wise.

Return type Variable

**chainer.functions.sum**

chainer.functions.sum\((\mathbf{x}, axis=None, keepdims=False)\)

Sum of array elements over a given axis.

Parameters

- \( \mathbf{x} \) (Variable) – Elements to sum.
- \( \text{axis}(\text{None, int, or tuple of int}) \) – Axis which a sum is performed. The default (axis = None) is perform a sum over all the dimensions of the input array.
- \( \text{keepdims} \) (bool) – If True, the specified axes are remained as axes of length one.

Returns Output variable.

Return type Variable

**chainer.functions.tan**

chainer.functions.tan\((\mathbf{x})\)

Elementwise tan function.

**Noise injections**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.functions.dropout</td>
<td>Drops elements of input variable randomly.</td>
</tr>
<tr>
<td>chainer.functions.gaussian</td>
<td>Gaussian sampling function.</td>
</tr>
<tr>
<td>chainer.functions.simplified_dropconnect</td>
<td>Linear unit regularized by simplified dropconnect.</td>
</tr>
</tbody>
</table>

**chainer.functions.dropout**

chainer.functions.dropout\((\mathbf{x}, ratio=.5)\)

Drops elements of input variable randomly.

This function drops input elements randomly with probability ratio and scales the remaining elements by
factor $1 / (1 - \text{ratio})$. In testing mode, it does nothing and just returns $x$.

**Warning:** train argument is not supported anymore since v2. Instead, use chainer.
using_config('train', train). See chainer.using_config().

**Parameters**
- $x$ (Variable) – Input variable.
- $\text{ratio}$ (float) – Dropout ratio.

**Returns** Output variable.

**Return type** Variable

See the paper by G. Hinton: Improving neural networks by preventing co-adaptation of feature detectors.

**chainer.functions.gaussian**

chainer.functions.gaussian ($\text{mean}$, $\text{ln}_{\text{var}}$)
Gaussian sampling function.
It takes mean $\mu$ and logarithm of variance $\log(\sigma^2)$ as input and output a sample drawn from gaussian $\mathcal{N}(\mu, \sigma)$.

**Parameters**
- $\text{mean}$ (Variable) – Input variable representing mean $\mu$.
- $\text{ln}_{\text{var}}$ (Variable) – Input variable representing logarithm of variance $\log(\sigma^2)$.

**Returns** Output variable.

**Return type** Variable

**chainer.functions.simplified_dropconnect**

chainer.functions.simplified_dropconnect ($x$, $W$, $b$=None, $\text{ratio}$=0.5, $\text{train}$=True, $\text{mask}$=None)
Linear unit regularized by simplified dropconnect.
Simplified dropconnect drops weight matrix elements randomly with probability $\text{ratio}$ and scales the remaining elements by factor $1 / (1 - \text{ratio})$. Which element is dropped depends on each sample. It accepts two or three arguments: an input minibatch $x$, a weight matrix $W$, and optionally a bias vector $b$. It computes $Y = xW^\top + b$.

In testing mode, zero will be used as simplified dropconnect ratio instead of $\text{ratio}$.

Notice: This implementation cannot be used for reproduction of the paper. There is a difference between the current implementation and the original one. The original version uses sampling with gaussian distribution before passing activation function, whereas the current implementation averages before activation.

**Parameters**
- $x$ (chainer.Variable or numpy.ndarray or cupy.ndarray) – Input variable. Its first dimension $n$ is assumed to be the minibatch dimension. The other dimensions are treated as concatenated one dimension whose size must be $N$.
- $W$ (Variable) – Weight variable of shape $(M, N)$.
- $b$ (Variable) – Bias variable (optional) of shape $(M,)$.
• **ratio** (*float*) – Dropconnect ratio.

• **train** (*bool*) – If True, executes simplified dropconnect. Otherwise, simplified dropconnect function works as a linear function.

• **mask**  
  
  (None or `chainer.Variable` or `numpy.ndarray` or `cupy.ndarray`) – If None, randomized dropconnect mask is generated. Otherwise, The mask must be (n, M, N) shaped array. Main purpose of this option is debugging. *mask* array will be used as a dropconnect mask.

**Returns**  
Output variable.

**Return type**  
`Variable`

**See also:**  
Dropconnect

**See also:**  

### Normalization functions

| **chainer.functions.batch_normalization** | Batch normalization function. |
| **chainer.functions.fixed_batch_normalization** | Batch normalization function with fixed statistics. |
| **chainer.Functions.local_response_normalization** | Local response normalization across neighboring channels. |
| **chainer.functions.normalize** | L2 norm squared (a.k.a.) |

**chainer.functions.batch_normalization**

**chainer.functions.batch_normalization**(*x, gamma, beta, eps=2e-5, running_mean=None, running_var=None, decay=0.9*)

Batch normalization function.

It takes the input variable *x* and two parameter variables `gamma` and `beta`. The parameter variables must both have the same dimensionality, which is referred to as the channel shape. This channel shape corresponds to the dimensions in the input which are not averaged over. Since the first dimension of the input corresponds to the batch size, the second dimension of *x* will correspond to the first dimension of the channel shape, the third dimension of *x* will correspond to the second channel dimension (if it exists) and so on. Therefore, the dimensionality of the input must be at least one plus the number of channel dimensions. The total effective “batch size” will then be considered to be the product of all dimensions in *x* except for the channel dimensions.

As an example, if the input is four dimensional and the parameter variables are one dimensional, then it is assumed that the first dimension of the input is the batch size, the second dimension is the channel size, and the remaining two dimensions are considered to be spatial dimensions that will be averaged over along with the batch size in the batch normalization computations. That is, the total batch size will be considered to be the product of all input dimensions except the second dimension.

Note: If this function is called, it will not be possible to access the updated running mean and variance statistics, because they are members of the function object, which cannot be accessed by the caller. If it is desired to access the updated running statistics, it is necessary to get a new instance of the function object, call the object, and then access the *running_mean* and/or *running_var* attributes. See the corresponding Link class for an example of how to do this.
Warning: train argument is not supported anymore since v2. Instead, use chainer.
using_config('train', train). See chainer.using_config().

Parameters

- **x** (*Variable*) – Input variable.
- **gamma** (*Variable*) – Scaling parameter of normalized data.
- **beta** (*Variable*) – Shifting parameter of scaled normalized data.
- **eps** (*float*) – Epsilon value for numerical stability.
- **running_mean** (*array*) – Running average of the mean. This is a running average of the mean over several mini-batches using the decay parameter. If None, the running average is not computed. If this is None, then **running_var** must also be None.
- **running_var** (*array*) – Running average of the variance. This is a running average of the variance over several mini-batches using the decay parameter. If None, the running average is not computed. If this is None, then **running_mean** must also be None.
- **decay** (*float*) – Decay rate of moving average. It is used during training.

See: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

See also:

chainer.functions.fixed_batch_normalization

chainer.functions.fixed_batch_normalization(*x, gamma, beta, mean, var, eps=2e-05*)

Batch normalization function with fixed statistics.

This is a variant of batch normalization, where the mean and variance statistics are given by the caller as fixed variables. This is used on testing mode of the batch normalization layer, where batch statistics cannot be used for prediction consistency.

Parameters

- **x** (*Variable*) – Input variable.
- **gamma** (*Variable*) – Scaling parameter of normalized data.
- **beta** (*Variable*) – Shifting parameter of scaled normalized data.
- **mean** (*Variable*) – Shifting parameter of input.
- **var** (*Variable*) – Square of scaling parameter of input.
- **eps** (*float*) – Epsilon value for numerical stability.

See also:

functions.batch_normalization(), links.BatchNormalization

chainer.functions.local_response_normalization

chainer.functions.local_response_normalization(*x, n=5, k=2, alpha=0.0001, beta=0.75*)

Local response normalization across neighboring channels.
This function implements normalization across channels. Let \( x \) an input image with \( N \) channels. Then, this function computes an output image \( y \) by following formula:

\[
y_i = \frac{x_i}{\left( k + \alpha \sum_{j=\max(1,i-n/2)}^{\min(N,i+n/2)} x_j^2 \right)^\beta}
\]

**Parameters**

- \( x \) (Variable) – Input variable.
- \( n \) (int) – Normalization window width.
- \( k \) (float) – Smoothing parameter.
- \( \alpha \) (float) – Normalizer scaling parameter.
- \( \beta \) (float) – Normalizer power parameter.

**Returns** Output variable.

**Return type** Variable

See: Section 3.3 of ImageNet Classification with Deep Convolutional Neural Networks

**chainer.functions.normalize**

\[
\text{chainer.functions.normalize}(x, \text{eps}=1e-05, \text{axis}=1)
\]

L2 norm squared (a.k.a. Euclidean norm).

This function implements L2 normalization on a vector along the given axis. No reduction is done along the normalization axis.

In the case when \( \text{axis}=1 \) and \( x \) is a vector of dimension \((N, K)\), where \( N \) and \( K \) denote mini-batch size and the dimension of the input variable, this function computes an output vector \( y \) by the following equation:

\[
y_i = \frac{x_i}{\|x_i\|_2 + \epsilon}
\]

\( \epsilon \) is used to avoid division by zero when norm of \( x \) along the given axis is zero.

The default value of \( \text{axis} \) is determined for backward compatibility.

**Parameters**

- \( x \) (Variable) – Two dimensional output variable. The first dimension is assumed to be the mini-batch dimension.
- \( \epsilon \) (float) – Epsilon value for numerical stability.
- \( \text{axis} \) (int) – Axis along which to normalize.

**Returns** The output variable which has the same shape as \( x \).

**Return type** Variable

**Spatial pooling**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.functions.average_pooling_2d</td>
<td>Spatial average pooling function.</td>
</tr>
<tr>
<td>chainer.functions.average_pooling_nd</td>
<td>N-dimensionally spatial average pooling function.</td>
</tr>
<tr>
<td>chainer.functions.max_pooling_2d</td>
<td>Spatial max pooling function.</td>
</tr>
</tbody>
</table>

Continued on next page
chainer.functions.average_pooling_2d

chainer.functions.average_pooling_2d(x, ksize, stride=None, pad=0)

Spatial average pooling function.

This function acts similarly to Convolution2D, but it computes the average of input spatial patch for each channel without any parameter instead of computing the inner products.

**Parameters**

- **x** (*Variable*) – Input variable.
- **ksize** (*int or pair of ints*) – Size of pooling window. ksize=k and ksize=(k, k) are equivalent.
- **stride** (*int or pair of ints or None*) – Stride of pooling applications. stride=s and stride=(s, s) are equivalent. If None is specified, then it uses same stride as the pooling window size.
- **pad** (*int or pair of ints*) – Spatial padding width for the input array. pad=p and pad=(p, p) are equivalent.

**Returns** Output variable.

**Return type** *Variable*

**Note:** This function currently does not support cover_all mode as max_pooling_2d(). Average pooling runs in non-cover-all mode.

chainer.functions.average_pooling_nd

chainer.functions.average_pooling_nd(x, ksize, stride=None, pad=0)

N-dimensionally spatial average pooling function.

This function provides a N-dimensionally generalized version of average_pooling_2d(). This acts similarly to ConvolutionND, but it computes the average of input spatial patch for each channel without any parameter instead of computing the inner products.

**Parameters**

- **x** (*Variable*) – Input variable.
- **ksize** (*int or tuple of ints*) – Size of pooling window. ksize=k and ksize=(k, k, ..., k) are equivalent.
- **stride** (*int or tuple of ints or None*) – Stride of pooling applications. stride=s and stride=(s, s, ..., s) are equivalent. If None is specified, then it uses same stride as the pooling window size.
• pad (int or tuple of ints) – Spatial padding width for the input array. pad=p and pad=(p, p, ..., p) are equivalent.

Returns Output variable.

Return type Variable

Note: This function currently does not support cover_all mode as max_pooling_nd(). Average pooling runs in non-cover-all mode.

chainer.functions.max_pooling_2d

chainer.functions.max_pooling_2d(x, ksize, stride=None, pad=0, cover_all=True)

Spatial max pooling function.

This function acts similarly to Convolution2D, but it computes the maximum of input spatial patch for each channel without any parameter instead of computing the inner products.

Parameters

• x (Variable) – Input variable.
• ksize (int or pair of ints) – Size of pooling window. ksize=k and ksize=(k, k) are equivalent.
• stride (int or pair of ints or None) – Stride of pooling applications. stride=s and stride=(s, s) are equivalent. If None is specified, then it uses same stride as the pooling window size.
• pad (int or pair of ints) – Spatial padding width for the input array. pad=p and pad=(p, p) are equivalent.
• cover_all (bool) – If True, all spatial locations are pooled into some output pixels. It may make the output size larger.

Returns Output variable.

Return type Variable

chainer.functions.max_pooling_nd

chainer.functions.max_pooling_nd(x, ksize, stride=None, pad=0, cover_all=True)

N-dimensionally spatial max pooling function.

This function provides a N-dimensionally generalized version of max_pooling_2d(). This acts similarly to ConvolutionND, but it computes the maximum of input spatial patch for each channel without any parameter instead of computing the inner products.

Parameters

• x (Variable) – Input variable.
• ksize (int or tuple of ints) – Size of pooling window. ksize=k and ksize=(k, k, ..., k) are equivalent.
• stride (int or tuple of ints or None) – Stride of pooling applications. stride=s and stride=(s, s, ..., s) are equivalent. If None is specified, then it uses same stride as the pooling window size.
• **pad**(int or tuple of ints) – Spatial padding width for the input array. \( \text{pad}=p \) and \( \text{pad}=(p, p, \ldots, p) \) are equivalent.

• **cover_all**(bool) – If True, all spatial locations are pooled into some output pixels. It may make the output size larger.

**Returns** Output variable.

**Return type** Variable

### chainer.functions.roi_pooling_2d

chainer.functions.roi_pooling_2d(x, rois, outh, outw, spatial_scale)

Spatial Region of Interest (ROI) pooling function.

This function acts similarly to MaxPooling2D, but it computes the maximum of input spatial patch for each channel with the region of interest.

**Parameters**

- **x**(Variable) – Input variable. The shape is expected to be 4 dimensional: (n: batch, c: channel, h, height, w: width).

- **rois**(Variable) – Input roi variable. The shape is expected to be (n: data size, 5), and each datum is set as below: (batch_index, x_min, y_min, x_max, y_max).

- **outh**(int) – Height of output image after pooled.

- **outw**(int) – Width of output image after pooled.

- **spatial_scale**(float) – Scale of the roi is resized.

**Returns** Output variable.

**Return type** Variable

See the original paper proposing ROIPooling: Fast R-CNN.

### chainer.functions.spatial_pyramid_pooling_2d

chainer.functions.spatial_pyramid_pooling_2d(x, pyramid_height, pooling_class)

Spatial pyramid pooling function.

It outputs a fixed-length vector regardless of input feature map size.

It performs pooling operation to the input 4D-array \( x \) with different kernel sizes and padding sizes, and then flattens all dimensions except first dimension of all pooling results, and finally concatenates them along second dimension.

At \( i \)-th pyramid level, the kernel size \( (k_h^{(i)}, k_w^{(i)}) \) and padding size \( (p_h^{(i)}, p_w^{(i)}) \) of pooling operation are calculated as below:

\[
\begin{align*}
  k_h^{(i)} &= \left\lfloor \frac{b_h}{2^i} \right\rfloor, \\
  k_w^{(i)} &= \left\lfloor \frac{b_w}{2^i} \right\rfloor, \\
  p_h^{(i)} &= \left( 2^i k_h^{(i)} - b_h \right) / 2, \\
  p_w^{(i)} &= \left( 2^i k_w^{(i)} - b_w \right) / 2,
\end{align*}
\]

where \( \left\lfloor \cdot \right\rfloor \) denotes the ceiling function, and \( b_h, b_w \) are height and width of input variable \( x \), respectively. Note that index of pyramid level \( i \) is zero-based.

Parameters

- **x** *(Variable)* – Input variable. The shape of x should be *(batchsize, # of channels, height, width).*
- **pyramid_height** *(int)* – Number of pyramid levels
- **pooling_class** *(MaxPooling2D or AveragePooling2D)* – Only MaxPooling2D class can be available for now.

Returns

Output variable. The shape of the output variable will be *(batchsize, \(c \sum_{h=0}^{H-1} 2^h \cdot 1, 1)\), where \(c\) is the number of channels of input variable \(x\) and \(H\) is the number of pyramid levels.

Return type *Variable*

---

**Note:** This function uses some pooling classes as components to perform spatial pyramid pooling. Now it supports only MaxPooling2D as elemental pooling operator so far.

---

**chainer.functions.unpooling_2d**

chainer.functions.unpooling_2d(x, ksize, stride=None, pad=0, outsize=None, cover_all=True)

Inverse operation of pooling for 2d array.

This function acts similarly to *Deconvolution2D*, but it spreads input 2d array’s value without any parameter instead of computing the inner products.

Parameters

- **x** *(Variable)* – Input variable.
- **ksize** *(int or pair of ints)* – Size of pooling window. ksize=k and ksize=(k, k) are equivalent.
- **stride** *(int, pair of ints or None)* – Stride of pooling applications. stride=s and stride=(s, s) are equivalent. If None is specified, then it uses same stride as the pooling window size.
- **pad** *(int or pair of ints)* – Spatial padding width for the input array. pad=p and pad=(p, p) are equivalent.
- **outsize** *(None or pair of ints)* – Expected output size (height, width) of array after the operation. If None, the size (height or width) is estimated from the size of input array in first batch with *get_deconv_outsize()* . If outsize is not None, the result of outsize applied to *get_conv_outsize()* must be equal to the shape of the 2d array in the input batch x.
- **cover_all** *(bool)* – If True, the output size may be smaller than the size if cover_all is False. This flag serves to align behavior to the pooling functions which can cover all input locations, see *max_pooling_2d()* and *convolution_2d()*.

Returns Output variable.

Return type *Variable*
chainer.functions.upsampling_2d

chainer.functions.upsampling_2d(x, indexes, ksize, stride=None, pad=0, outsize=None, cover_all=True)

Upsampling using pooling indices.

This function produces an upsampled image using pooling indices.

Example

It should be noted that you need to turn off chainer.config.use_cudnn flag when you perform max_pooling_2d() function which will make a pooling indicies for this upsampling_2d(). It is because indexes is never created and stored in the MaxPooling2D object when cuDNN is used for it.

```python
>>> x = np.arange(1, 37).reshape(1, 1, 6, 6).astype('f')
>>> x = chainer.Variable(x)
>>> x.data
array([[[ 1.,  2.,  3.,  4.,  5.,  6.],
        [ 7.,  8.,  9., 10., 11., 12.],
        [13., 14., 15., 16., 17., 18.],
        [19., 20., 21., 22., 23., 24.],
        [25., 26., 27., 28., 29., 30.],
        [31., 32., 33., 34., 35., 36.]]], dtype=float32)
```

This is the original x before max pooling.

```python
>>> p = F.MaxPooling2D(2, 2)
>>> with chainer.using_config('use_cudnn', 'never'):
...   pooled_x = p(x)
>>> pooled_x.data
array([[[  8.,  10.,  12.],
        [ 20.,  22.,  24.],
        [ 32.,  34.,  36.]]], dtype=float32)
```

This is the output of the max pooling operation. upsampling_2d() needs indexes array stored in the max pooling object p.

```python
>>> upsampled_x = F.upsampling_2d(  
...   pooled_x, p.indexes, p.kh, p.sy, p.ph, x.shape[2:])
>>> upsampled_x.shape
(1, 1, 6, 6)
>>> upsampled_x.data
array([[[ 0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  8.,  0., 10.,  0., 12.],
        [ 0.,  0.,  0.,  0.,  0.,  0.],
        [ 0., 20.,  0., 22.,  0., 24.],
        [ 0.,  0.,  0.,  0.,  0.,  0.],
        [ 0., 32.,  0., 34.,  0., 36.]]], dtype=float32)
```

Parameters

- **x (Variable)** – Input variable.
- **indexes (ndarray or ndarray)** – Index array that was used to calculate x with MaxPooling2D.
- **ksize (int or (int, int))** – ksize attribute of MaxPooling2D object that is used to calculate x
• **stride** *(int or (int, int)) – stride attribute of MaxPooling2D object that is used to calculate x*

• **pad** *(int or (int, int)) – pad attribute of MaxPooling2D object that is used to calculate x*

• **outsize** *(int, int) – Expected output size (height, width).*

• **cover_all** *(bool) – Whether cover_all is used in the MaxPooling2D object or not.*

**Returns** Output variable.

**Return type** *Variable*

### Utility functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.functions.forget</code></td>
<td>Call a function without storing internal results.</td>
</tr>
</tbody>
</table>

**`chainer.functions.forget`**

`chainer.functions.forget(func, *xs)`  
Call a function without storing internal results.

On a forward propagation Chainer stores all internal results of `Function` on a computational graph as they are required on backward-propagation. These results consume too much memory when the internal results are too large. This method `forgets` such internal results on forward propagation, and still supports back-propagation with recalculation.

In a forward propagation, this method calls a given function with given variables without creating a computational graph. That means, no internal results are stored. In a backward propagation this method calls the given function again to create a computational graph to execute back-propagation.

This method reduces internal memory usage. Instead it requires more calculation time as it calls the function twice.

**Example**

Let $f$ be a function defined as:

```python
>>> def f(a, b):
...     return a + b + a
```

and, $x$ and $y$ be `Variable`:

```python
>>> x = chainer.Variable(np.random.uniform(-1, 1, 5).astype('f'))
>>> y = chainer.Variable(np.random.uniform(-1, 1, 5).astype('f'))
```

When $z$ is calculated as $z = f(x, y)$, its internal result $x + y$ is stored in memory. Instead if you call $f$ with `forget()`:

```python
>>> z = F.forget(f, x, y)
```

internal $x + y$ is forgotten.
Note: The method does not support functions behaving randomly, such as `dropout()` and `negative_sampling()`. It is because first results of these functions differ from the second one.

Parameters

- **func (callable)** – A function to call. It needs to be called with `Variable` object(s) and to return a `Variable` object or a tuple of `Variable` objects.

- **xs (Variable)** – Argument variables of the function.

Returns A variable `func` returns. If it returns a tuple, the method returns a tuple too.

Return type `Variable`

### Standard Link implementations

Chainer provides many Link implementations in the `chainer.links` package.

**Note:** Some of the links are originally defined in the `chainer.functions` namespace. They are still left in the namespace for backward compatibility, though it is strongly recommended to use them via the `chainer.links` package.

### Learnable connections

<table>
<thead>
<tr>
<th>Link</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.links.Bias</code></td>
<td>Broadcasted elementwise summation with learnable parameters.</td>
</tr>
<tr>
<td><code>chainer.links.Bilinear</code></td>
<td>Bilinear layer that performs tensor multiplication.</td>
</tr>
<tr>
<td><code>chainer.links.Convolution2D</code></td>
<td>Two-dimensional convolutional layer.</td>
</tr>
<tr>
<td><code>chainer.links.ConvolutionND</code></td>
<td>N-dimensional convolution layer.</td>
</tr>
<tr>
<td><code>chainer.links.Deconvolution2D</code></td>
<td>Two dimensional deconvolution function.</td>
</tr>
<tr>
<td><code>chainer.links.DeconvolutionND</code></td>
<td>N-dimensional deconvolution function.</td>
</tr>
<tr>
<td><code>chainer.links.DepthwiseConvolution2D</code></td>
<td>Two-dimensional depthwise convolutional layer.</td>
</tr>
<tr>
<td><code>chainer.links.DilatedConvolution2D</code></td>
<td>Two-dimensional dilated convolutional layer.</td>
</tr>
<tr>
<td><code>chainer.links.EmbedID</code></td>
<td>Efficient linear layer for one-hot input.</td>
</tr>
<tr>
<td><code>chainer.links.GRU</code></td>
<td>Stateful Gated Recurrent Unit function (GRU)</td>
</tr>
<tr>
<td><code>chainer.links.Highway</code></td>
<td>Highway module.</td>
</tr>
<tr>
<td><code>chainer.links.Inception</code></td>
<td>Inception module of GoogLeNet.</td>
</tr>
<tr>
<td><code>chainer.links.InceptionBN</code></td>
<td>Inception module of the new GoogLeNet with BatchNormalization.</td>
</tr>
<tr>
<td><code>chainer.links.Linear</code></td>
<td>Linear layer (a.k.a.</td>
</tr>
<tr>
<td><code>chainer.links.LSTM</code></td>
<td>Fully-connected LSTM layer.</td>
</tr>
<tr>
<td><code>chainer.links.MLPConvolution2D</code></td>
<td>Two-dimensional MLP convolution layer of Network in Network.</td>
</tr>
<tr>
<td><code>chainer.links.NStepBiGRU</code></td>
<td>Stacked Bi-directional GRU for sequences.</td>
</tr>
<tr>
<td><code>chainer.links.NStepBiLSTM</code></td>
<td>Stacked Bi-directional LSTM for sequences.</td>
</tr>
<tr>
<td><code>chainer.links.NStepBiRNNReLU</code></td>
<td>Stacked Bi-directional RNN for sequences.</td>
</tr>
<tr>
<td><code>chainer.links.NStepBiRNNTanh</code></td>
<td>Stacked Bi-directional RNN for sequences.</td>
</tr>
</tbody>
</table>
Table 3.40 – continued from previous page

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.links.NStepGRU</td>
<td>Stacked Uni-directional GRU for sequnces.</td>
</tr>
<tr>
<td>chainer.links.NStepLSTM</td>
<td>Stacked Uni-directional LSTM for sequnces.</td>
</tr>
<tr>
<td>chainer.links.NStepRNNReLU</td>
<td>Stacked Uni-directional RNN for sequnces.</td>
</tr>
<tr>
<td>chainer.links.NStepRNNTanh</td>
<td>Stacked Uni-directional RNN for sequnces.</td>
</tr>
<tr>
<td>chainer.links.Scale</td>
<td>Broadcasted elementwise product with learnable parameters.</td>
</tr>
<tr>
<td>chainer.links.StatefulGRU</td>
<td>Stateful Gated Recurrent Unit function (GRU).</td>
</tr>
<tr>
<td>chainer.links.StatefulPeepholeLSTM</td>
<td>Fully-connected LSTM layer with peephole connections.</td>
</tr>
<tr>
<td>chainer.links.StatelessLSTM</td>
<td>Stateless LSTM layer.</td>
</tr>
</tbody>
</table>

**chainer.links.Bias**

```python
class chainer.links.Bias(axis=1, shape=None)
```

Broadcasted elementwise summation with learnable parameters.

Computes a elementwise summation as `bias()` function does except that its second input is a learnable bias parameter $b$ the link has.

**Parameters**

- **axis** (*int*) – The first axis of the first input of `bias()` function along which its second input is applied.
- **shape** (*tuple of ints*) – Shape of the learnable bias parameter. If `None`, this link does not have learnable parameters so an explicit bias needs to be given to its `__call__` method’s second input.

**See also:**

See `bias()` for details.

**Variables**

- **b** (*Variable*) – Bias parameter if `shape` is given. Otherwise, no attributes.

**Methods**

```python
__call__(*xs)
```

Applies broadcasted elementwise summation.

**Parameters**

- **xs** (*list of Variables*) – Input variables whose length should be one if the link has a learnable bias parameter, otherwise should be two.

```python
add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
```

Registers a parameter to the link.

Deprecated since version 2.0.0: Assign a `Parameter` object directly to an attribute within an `initialization scope` instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**
• **name** (*str*) – Name of the parameter. This name is also used as the attribute name.

• **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• **dtype** – Data type of the parameter array.

• **initializer** – If it is not **None**, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, **dtype** argument is ignored.

### `add_persistent(name, value)`
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

• **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.

• **value** – Value to be registered.

### `addgrads(link)`
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

• **link** (*Link*) – Source link object.

### `children()`
Returns a generator of all child links.

**Returns** A generator object that generates all child links.

### `cleargrads()`
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

### `copy()`
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns** Copied link object.

**Return type** *Link*

### `copyparams(link)`
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

• **link** (*Link*) – Source link object.

### `disable_update()`
Disables update rules of all parameters under the link hierarchy.
This method sets the `attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

**enable_update**

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

**init_scope** (*args, **kwds*)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** (`skipself=False`)

Returns a generator of all links under the hierarchy.

**Parameters skipself** (`bool`) – If `True`, then the generator skips this link and starts with the first child link.

**Returns** A generator object that generates all links.

**namedlinks** (`skipself=False`)

Returns a generator of all (path, link) pairs under the hierarchy.

**Parameters skipself** (`bool`) – If `True`, then the generator skips this link and starts with the first child link.

**Returns** A generator object that generates all (path, link) pairs.

**namedparams** (`include_uninit=True`)

Returns a generator of all (path, param) pairs under the hierarchy.

**Parameters include_uninit** (`bool`) – If `True`, it also generates uninitialized parameters.

**Returns** A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params** (`include_uninit=True`)

Returns a generator of all parameters under the link hierarchy.

**Parameters include_uninit** (`bool`) – If `True`, it also generates uninitialized parameters.

**Returns** A generator object that generates all parameters.

**register_persistent** (`name`)

Registers an attribute of a given name as a persistent value.
This is a convenient method to register an existing attribute as a persistent value. If \texttt{name} has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**

\texttt{name} (str) – Name of the attribute to be registered.

\texttt{serialize} (serializer)

Serializes the link object.

**Parameters**

\texttt{serializer} (AbstractSerializer) – Serializer object.

\texttt{to_cpu}()

Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

Returns: self

\texttt{to_gpu}(device=None)

Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

**Parameters**

\texttt{device} – Target device specifier. If omitted, the current device is used.

Returns: self

\texttt{zerograds}()

Initializes all gradient arrays by zero.

This method can be used for the same purpose of \texttt{cleargrads}, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use \texttt{cleargrads()} instead.

**Attributes**

\texttt{update_enabled}

True if at least one parameter has an update rule enabled.

\texttt{within_init_scope}

True if the current code is inside of an initialization scope.

See \texttt{init_scope()} for the details of the initialization scope.

\texttt{xp}

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns \texttt{numpy} or \texttt{cupy}.

**chainer.links.Bilinear**

\texttt{class} chainer.links.Bilinear (left_size, right_size, out_size, nobias=False, initialW=None, initial_bias=None)

Bilinear layer that performs tensor multiplication.

Bilinear is a primitive link that wraps the \texttt{bilinear()} functions. It holds parameters \(W, V_1, V_2\), and \(b\) corresponding to the arguments of \texttt{bilinear()}.

**Parameters**
• **left_size** (*int*) – Dimension of input vector \( e^1 (J) \)
• **right_size** (*int*) – Dimension of input vector \( e^2 (K) \)
• **out_size** (*int*) – Dimension of output vector \( y (L) \)
• **nobias** (*bool*) – If True, parameters \( V_1, V_2, \) and \( b \) are omitted.
• **initialW** (*3-D array*) – Initial value of \( W \). Shape of this argument must be \((\text{left_size}, \text{right_size}, \text{out_size})\). If None, the default initializer is used. May also be a callable that takes \numpy.ndarray or \cupy.ndarray and edits its value.
• **initial_bias** (*tuple*) – Initial values of \( V_1, V_2 \) and \( b \). The length of this argument must be 3. Each element of this tuple must have the shapes of \((\text{left_size}, \text{out_size})\), \((\text{right_size}, \text{out_size})\), and \((\text{out_size},)\), respectively. If None, \( V_1 \) and \( V_2 \) are initialized by the default initializer and \( b \) is set to 0. May also be a tuple of callables that take \numpy.ndarray or \cupy.ndarray and edit its value.

See also:
See *chainer.functions.bilinear()* for details.

Variables

• **\( W \)** (*Variable*) – Bilinear weight parameter.
• **\( V_1 \)** (*Variable*) – Linear weight parameter for the first argument.
• **\( V_2 \)** (*Variable*) – Linear weight parameter for the second argument.
• **\( b \)** (*Variable*) – Bias parameter.

Methods

**__call__** (*\( e_1, e_2 \)*)
Applies the bilinear function to inputs and the internal parameters.

Parameters

• **\( e_1 \)** (*Variable*) – Left input.
• **\( e_2 \)** (*Variable*) – Right input.

Returns Output variable.

Return type **Variable**

**add_param** (*name, shape=None, dtype=<type 'numpy.float32'>, initializer=None*)
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters
• **name** (*str*) – Name of the parameter. This name is also used as the attribute name.

• **shape** (*int* or *tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• **dtype** – Data type of the parameter array.

• **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, **dtype** argument is ignored.

### add_persistent (*name*, *value*)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

• **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.

• **value** – Value to be registered.

### addgrads (*link*)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

• **link** (*Link*) – Source link object.

### children()
Returns a generator of all child links.

**Returns** A generator object that generates all child links.

### cleargrads()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

### copy()
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns** Copied link object.

**Return type** *Link*

### copyparams (*link*)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

• **link** (*Link*) – Source link object.

### disable_update()
Disables update rules of all parameters under the link hierarchy.

---

### 3.5. Standard Link implementations

215
This method sets the `attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

**enable_update()**

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

**init_scope(*args, **kwds)**

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links**(skipself=False)

Returns a generator of all links under the hierarchy.

*Parameters skipself (bool) – If True, then the generator skips this link and starts with the first child link.*

*Returns* A generator object that generates all links.

**namedlinks**(skipself=False)

Returns a generator of all (path, link) pairs under the hierarchy.

*Parameters skipself (bool) – If True, then the generator skips this link and starts with the first child link.*

*Returns* A generator object that generates all (path, link) pairs.

**namedparams**(include_uninit=True)

Returns a generator of all (path, param) pairs under the hierarchy.

*Parameters include_uninit (bool) – If True, it also generates uninitialized parameters.*

*Returns* A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params**(include_uninit=True)

Returns a generator of all parameters under the link hierarchy.

*Parameters include_uninit (bool) – If True, it also generates uninitialized parameters.*

*Returns* A generator object that generates all parameters.

**registerPersistent**(name)

Registers an attribute of a given name as a persistent value.
This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**

- **name** *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*

Serializes the link object.

**Parameters**

- **serializer** *(AbstractSerializer)* – Serializer object.

**to_cpu**

Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

Returns: self

**to_gpu** *(device=None)*

Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

**Parameters**

- **device** – Target device specifier. If omitted, the current device is used.

Returns: self

**zero_grads**

**zerograds**

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads()` instead.

**Attributes**

- **update_enabled**

  True if at least one parameter has an update rule enabled.

- **within_init_scope**

  True if the current code is inside of an initialization scope.

  See `init_scope()` for the details of the initialization scope.

- **xp**

  Array module for this link.

  Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

```python
class chainer.links.Convolution2D:

    Two-dimensional convolutional layer.

    This link wraps the `convolution_2d()` function and holds the filter weight and bias vector as parameters.
```
The output of this function can be non-deterministic when it uses cuDNN. If `chainer.configuration.config.deterministic` is True and cuDNN version is >= v3, it forces cuDNN to use a deterministic algorithm.

**Warning:** deterministic argument is not supported anymore since v2. Instead, use `chainer.using_config('cudnn_deterministic', value)` (value is either True or False). See `chainer.using_config()`.

**Parameters**

- **in_channels** *(int or None)* – Number of channels of input arrays. If None, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_channels** *(int)* – Number of channels of output arrays.
- **ksize** *(int or pair of ints)* – Size of filters (a.k.a. kernels). ksize=k and ksize=(k, k) are equivalent.
- **stride** *(int or pair of ints)* – Stride of filter applications. stride=s and stride=(s, s) are equivalent.
- **pad** *(int or pair of ints)* – Spatial padding width for input arrays. pad=p and pad=(p, p) are equivalent.
- **nobias** *(bool)* – If True, then this link does not use the bias term.
- **initialW** *(4-D array)* – Initial weight value. If None, the default initializer is used. May also be a callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value.
- **initial_bias** *(1-D array)* – Initial bias value. If None, the bias is set to 0. May also be a callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value.

**See also:**

See `chainer.functions.convolution_2d()` for the definition of two-dimensional convolution.

**Variables**

- **W** *(Variable)* – Weight parameter.
- **b** *(Variable)* – Bias parameter.

**Example**

There are several ways to make a Convolution2D link.

Let an input vector \( x \) be:

```python
>>> x = np.arange(1 * 3 * 10 * 10, dtype='f').reshape(1, 3, 10, 10)
```

1. Give the first three arguments explicitly:

```python
>>> l = L.Convolution2D(3, 7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```
2. Omit `in_channels` or fill it with `None`:

The below two cases are the same.

```python
>>> l = L.Convolution2D(7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)

>>> l = L.Convolution2D(None, 7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```

When you omit the first argument, you need to specify the other subsequent arguments from `stride` as keyword arguments. So the below two cases are the same.

```python
>>> l = L.Convolution2D(7, 5, stride=1, pad=0)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)

>>> l = L.Convolution2D(None, 7, 5, 1, 0)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```

### Methods

**`__call__`(x)**

Applies the convolution layer.

**Parameters**

- `x` (*Variable*) – Input image.

**Returns**

Output of the convolution.

**Return type**

*Variable*

**add_param** *(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)*

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a *Parameter* object directly to an attribute within an *initialization scope* instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- `name` (*str*) – Name of the parameter. This name is also used as the attribute name.
• **shape** *(int or tuple of ints)* – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• **dtype** – Data type of the parameter array.

• **initializer** – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

**add_persistent** *(name, value)*

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

• **name** *(str)* – Name of the persistent value. This name is also used for the attribute name.

• **value** – Value to be registered.

**addgrads** *(link)*

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

• **link** *(Link)* – Source link object.

**children** *

Returns a generator of all child links.

**Returns** A generator object that generates all child links.

**cleargrads** *

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy** *

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns** Copied link object.

**Return type** *Link*

**copyparams** *(link)*

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

• **link** *(Link)* – Source link object.

**disable_update** *

Disables update rules of all parameters under the link hierarchy.

This method sets the `attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.
enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope(*args, **kwds)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))

links(skipself=False)

Returns a generator of all links under the hierarchy.

Parameters skipself (bool) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

dnamedlinks(skipself=False)

Returns a generator of all (path, link) pairs under the hierarchy.

Parameters skipself (bool) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

namedparams(include_uninit=True)

Returns a generator of all (path, param) pairs under the hierarchy.

Parameters include_uninit (bool) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(include_uninit=True)

Returns a generator of all parameters under the link hierarchy.

Parameters include_uninit (bool) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

register_persistent(name)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

3.5. Standard Link implementations
Parameters

**name**(str) – Name of the attribute to be registered.

**serialize**(serializer)

Serializes the link object.

Parameters

**serializer**(AbstractSerializer) – Serializer object.

**to_cpu**

Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

Returns: self

**to_gpu**(device=None)

Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

Parameters

**device** – Target device specifier. If omitted, the current device is used.

Returns: self

**zerograds**

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads()` instead.

**Attributes**

**update_enabled**

True if at least one parameter has an update rule enabled.

**within_init_scope**

True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

**xp**

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

`chainer.links.ConvolutionND`

```python
class chainer.links.ConvolutionND(ndim, in_channels, out_channels, ksize, stride=1, pad=0, nobias=False, initialW=None, initial_bias=None, cover_all=False)
```

N-dimensional convolution layer.

This link wraps the `convolution_nd()` function and holds the filter weight and bias vector as parameters.

Parameters

- **ndim**(int) – Number of spatial dimensions.
- **in_channels**(int) – Number of channels of input arrays.
• **out_channels** (*int*) – Number of channels of output arrays.

• **ksize** (*int* or *tuple of ints*) – Size of filters (a.k.a. kernels). \(ksize=k\) and \(ksize=(k, k, \ldots, k)\) are equivalent.

• **stride** (*int* or *tuple of ints*) – Stride of filter application. \(stride=s\) and \(stride=(s, s, \ldots, s)\) are equivalent.

• **pad** (*int* or *tuple of ints*) – Spatial padding width for input arrays. \(pad=p\) and \(pad=(p, p, \ldots, p)\) are equivalent.

• **nobias** (*bool*) – If True, then this function does not use the bias.

• **initialW** (*array*) – Initial weight array. If None, the default initializer is used. May be a callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value.

• **initial_bias** (*array*) – Initial bias vector. If None, the bias is set to zero. May be a callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value.

• **cover_all** (*bool*) – If True, all spatial locations are convoluted into some output pixels. It may make the output size larger. `cover_all` needs to be False if you want to use cuDNN.

See also:
See `convolution_nd()` for the definition of N-dimensional convolution. See `convolution_2d()` for the definition of two-dimensional convolution.

Variables

• **W** (*Variable*) – Weight parameter.

• **b** (*Variable*) – Bias parameter. If `initial_bias` is None, set to None.

Methods

```python
__call__(x)
```
Applies N-dimensional convolution layer.

**Parameters**

- **x** (*Variable*) – Input image.

**Returns**
Output of convolution.

**Return type**
*Variable*

```python
add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
```
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a `Parameter` object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
• **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• **dtype** – Data type of the parameter array.

• **initializer** – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

### add_persistent

`add_persistent(name, value)`

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

### addgrads

`addgrads(link)`

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

- **link** (*Link*) – Source link object.

### children

`children()`

Returns a generator of all child links.

**Returns** A generator object that generates all child links.

### cleargrads

`cleargrads()`

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

### copy

`copy()`

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns** Copied link object.

**Return type** *Link*

### copyparams

`copyparams(link)`

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

- **link** (*Link*) – Source link object.

### disable_update

`disable_update()`

Disables update rules of all parameters under the link hierarchy.

This method sets the `attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`. 
**enable_update()**
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

**init_scope(*args, **kwds)**
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

**Example**
In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links(skipself=False)**
Returns a generator of all links under the hierarchy.

- **Parameters** `skipself(bool)` – If `True`, then the generator skips this link and starts with the first child link.
- **Returns** A generator object that generates all links.

**namedlinks(skipself=False)**
Returns a generator of all (path, link) pairs under the hierarchy.

- **Parameters** `skipself(bool)` – If `True`, then the generator skips this link and starts with the first child link.
- **Returns** A generator object that generates all (path, link) pairs.

**namedparams(include_uninit=True)**
Returns a generator of all (path, param) pairs under the hierarchy.

- **Parameters** `include_uninit(bool)` – If `True`, it also generates uninitialized parameters.
- **Returns** A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params(include_uninit=True)**
Returns a generator of all parameters under the link hierarchy.

- **Parameters** `include_uninit(bool)` – If `True`, it also generates uninitialized parameters.
- **Returns** A generator object that generates all parameters.

**register_persistent(name)**
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.
Parameters name (str) – Name of the attribute to be registered.

serialize (serializer)
    Serializes the link object.
    Parameters serializer (AbstractSerializer) – Serializer object.

to_cpu()
   Copies parameter variables and persistent values to CPU.
    This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.
    Returns: self

to_gpu (device=None)
   Copies parameter variables and persistent values to GPU.
    This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.
    Parameters device – Target device specifier. If omitted, the current device is used.
    Returns: self

zerograd()
   Initializes all gradient arrays by zero.
    This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.
    Deprecated since version v1.15: Use cleargrads() instead.

Attributes

update_enabled
    True if at least one parameter has an update rule enabled.

within_init_scope
    True if the current code is inside of an initialization scope.
    See init_scope() for the details of the initialization scope.

xp
    Array module for this link.
    Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.Deconvolution2D

class chainer.links.Deconvolution2D (self, in_channels, out_channels, ksize=None, stride=1, pad=0, nobias=False, outsize=None, initialW=None, initial_bias=None)

Two dimensional deconvolution function.
This link wraps the deconvolution_2d() function and holds the filter weight and bias vector as parameters.

Warning: deterministic argument is not supported anymore since v2. Instead, use chainer.using_config('cudnn_deterministic', value) (value is either True or False). See chainer.using_config().
Parameters

- **in_channels** *(int or None)* – Number of channels of input arrays. If None, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.

- **out_channels** *(int)* – Number of channels of output arrays.

- **ksize** *(int or pair of ints)* – Size of filters (a.k.a. kernels). ksize=k and ksize=(k, k) are equivalent.

- **stride** *(int or pair of ints)* – Stride of filter applications. stride=s and stride=(s, s) are equivalent.

- **pad** *(int or pair of ints)* – Spatial padding width for input arrays. pad=p and pad=(p, p) are equivalent.

- **nobias** *(bool)* – If True, then this function does not use the bias term.

- **outsize** *(tuple)* – Expected output size of deconvolutional operation. It should be pair of height and width *(out_H, out_W)*. Default value is None and the outsize is estimated by input size, stride and pad.

- **initialW** *(4-D array)* – Initial weight value. If None, the default initializer is used. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.

- **initial_bias** *(1-D array)* – Initial bias value. If None, the bias vector is set to zero. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.

The filter weight has four dimensions *(c_I, c_O, k_H, k_W)* which indicate the number of input channels, output channels, height and width of the kernels, respectively. The filter weight is initialized with i.i.d. Gaussian random samples, each of which has zero mean and deviation \(\sqrt{1/(c_I k_H k_W)}\) by default.

The bias vector is of size c_O. Its elements are initialized by bias argument. If nobias argument is set to True, then this function does not hold the bias parameter.

The output of this function can be non-deterministic when it uses cuDNN. If chainer.configuration.config.cudnn_deterministic is True and cuDNN version is >= v3, it forces cuDNN to use a deterministic algorithm.

See also:

See chainer.functions.deconvolution_2d() for the definition of two-dimensional convolution.

See also:

See chainer.links.Convolution2D() for the examples of ways to give arguments to this link.

Example

There are several ways to make a Deconvolution2D link.

Let an input vector x be:

```python
>>> x = np.arange(1 * 3 * 10 * 10, dtype='f').reshape(1, 3, 10, 10)
```

1. Give the first three arguments explicitly:

   In this case, all the other arguments are set to the default values.
>>> l = L.Deconvolution2D(3, 7, 4)
>>> y = l(x)
>>> y.shape
(1, 7, 13, 13)

2. Omit `in_channels` or fill it with `None`:

The below two cases are the same.

```python
>>> l = L.Deconvolution2D(7, 4)
>>> y = l(x)
>>> y.shape
(1, 7, 13, 13)
```

```python
>>> l = L.Deconvolution2D(None, 7, 4)
>>> y = l(x)
>>> y.shape
(1, 7, 13, 13)
```

When you omit the first argument, you need to specify the other subsequent arguments from `stride` as keyword arguments. So the below two cases are the same.

```python
>>> l = L.Deconvolution2D(None, 7, 4, 2, 1)
>>> y = l(x)
>>> y.shape
(1, 7, 20, 20)
```

```python
>>> l = L.Deconvolution2D(7, 4, stride=2, pad=1)
>>> y = l(x)
>>> y.shape
(1, 7, 20, 20)
```

## Methods

**__call__**(x)

**add_param**(name, shape=None, dtype=<type ‘numpy.float32’>, initializer=None)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a `Parameter` object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

### Parameters

- **name**(str) – Name of the parameter. This name is also used as the attribute name.
• **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• **dtype** – Data type of the parameter array.

• **initializer** – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

**add_persistent** (*name, value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

• **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.

• **value** – Value to be registered.

**addgrads** (*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

• **link** (*Link*) – Source link object.

**children**()

Returns a generator of all child links.

**Returns** A generator object that generates all child links.

**cleargrads**()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy**()

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns** Copied link object.

**Return type** *Link*

**copyparams** (*link*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

• **link** (*Link*) – Source link object.

**disable_update**()

Disables update rules of all parameters under the link hierarchy.

This method sets the `attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

### 3.5. Standard Link implementations
**enable_update()**
Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

**init_scope(*args, **kwds)**
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

**Example**
In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links**(skipself=False)
Returns a generator of all links under the hierarchy.

Parameters **skipself**(bool) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all links.

**namedlinks**(skipself=False)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters **skipself**(bool) – If True, then the generator skips this link and starts with the first child link.

Returns A generator object that generates all (path, link) pairs.

**namedparams**(include_uninit=True)
Returns a generator of all (path, param) pairs under the hierarchy.

Parameters **include_uninit**(bool) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params**(include_uninit=True)
Returns a generator of all parameters under the link hierarchy.

Parameters **include_uninit**(bool) – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

**register_persistent**(name)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.
Parameters

**name** *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*

Serializes the link object.

**Parameters**

**serializer** *(AbstractSerializer)* – Serializer object.

**to_cpu()**

Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

Returns: self

**to_gpu**(device=None)

Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

**Parameters**

**device** – Target device specifier. If omitted, the current device is used.

Returns: self

**zerograds()**

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use **cleargrads()** instead.

**Attributes**

**update_enabled**

True if at least one parameter has an update rule enabled.

**within_init_scope**

True if the current code is inside of an initialization scope.

See **init_scope()** for the details of the initialization scope.

**xp**

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns **numpy** or **cupy**.

**chainer.links.DeconvolutionND**

**class** chainer.links.DeconvolutionND *(ndim, in_channels, out_channels, ksize, stride=1, pad=0, nobias=False, outsize=None, initialW=None, initial_bias=None)*

N-dimensional deconvolution function.

This link wraps **deconvolution_nd()** function and holds the filter weight and bias vector as its parameters.

**Parameters**

- **ndim** *(int)* – Number of spatial dimensions.
- **in_channels** *(int)* – Number of channels of input arrays.
• **out_channels** (*int*) – Number of channels of output arrays.

• **ksize** (*int or tuple of ints*) – Size of filters (a.k.a. kernels). ksize=k and ksize=(k, k, ..., k) are equivalent.

• **stride** (*int or tuple of ints*) – Stride of filter application. stride=s and stride=(s, s, ..., s) are equivalent.

• **pad** (*int or tuple of ints*) – Spatial padding width for input arrays. pad=p and pad=(p, p, ..., p) are equivalent.

• **nobias** (*bool*) – If True, then this function does not use the bias.

• **outsize** (*tuple of ints*) – Expected output size of deconvolutional operation. It should be a tuple of ints that represents the output size of each dimension. Default value is None and the outsize is estimated with input size, stride and pad.

• **initialW** (*array*) – Initial weight array. If None, the default initializer is used. May be an initializer instance of another value that the init_weight() function can take.

• **initial_bias** (*array*) – Initial bias array. If None, the bias vector is set to zero. May be an initializer instance of another value that the init_weight() function can take.

See also:

deeconvolution_nd()

**Variables**

• **W** (*Variable*) – Weight parameter.

• **b** (*Variable*) – Bias parameter. If initial_bias is None, set to None.

**Methods**

`__call__(x)`

`add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)`

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

• **name** (*str*) – Name of the parameter. This name is also used as the attribute name.

• **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• **dtype** – Data type of the parameter array.
• **initializer** – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

**add_persistent**(name, value)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name**(str) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads**(link)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

- **link**(Link) – Source link object.

**children**()

Returns a generator of all child links.

**Returns** A generator object that generates all child links.

**cleargrads**()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy**()

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns** Copied link object.

**Return type** Link

**copyparams**(link)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

- **link**(Link) – Source link object.

**disable_update**()

Disables update rules of all parameters under the link hierarchy.

This method sets the `chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

**enable_update**()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`. 3.5. Standard Link implementations

---

233
**init_scope** (*args, **kwds*)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the **init_scope** method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** (skipself=False)

Returns a generator of all links under the hierarchy.

- **Parameters skipself** (bool) – If True, then the generator skips this link and starts with the first child link.

- **Returns** A generator object that generates all links.

**namedlinks** (skipself=False)

Returns a generator of all (path, link) pairs under the hierarchy.

- **Parameters skipself** (bool) – If True, then the generator skips this link and starts with the first child link.

- **Returns** A generator object that generates all (path, link) pairs.

**namedparams** (include_uninit=True)

Returns a generator of all (path, param) pairs under the hierarchy.

- **Parameters include_uninit** (bool) – If True, it also generates uninitialized parameters.

- **Returns** A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params** (include_uninit=True)

Returns a generator of all parameters under the link hierarchy.

- **Parameters include_uninit** (bool) – If True, it also generates uninitialized parameters.

- **Returns** A generator object that generates all parameters.

**register_persistent** (name)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

- **Parameters name** (str) – Name of the attribute to be registered.

**serialize** (serializer)

Serializes the link object.
Parameters serializer (AbstractSerializer) – Serializer object.

**to_cpu()**
Copies parameter variables and persistent values to CPU.
This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.
Returns: self

**to_gpu()**
Copies parameter variables and persistent values to GPU.
This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

**Parameters device** – Target device specifier. If omitted, the current device is used.
Returns: self

**zerograds()**
Initializes all gradient arrays by zero.
This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.
Deprecated since version v1.15: Use cleargrads() instead.

**Attributes**

**update_enabled**
True if at least one parameter has an update rule enabled.

**within_init_scope**
True if the current code is inside of an initialization scope.
See init_scope() for the details of the initialization scope.

**xp**
Array module for this link.
Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

**chainer.links.DepthwiseConvolution2D**

class chainer.links.DepthwiseConvolution2D (in_channels, channel_multiplier, ksize, stride=1, pad=0, nobias=False, initialW=None, initial_bias=None)

Two-dimensional depthwise convolutional layer.
This link wraps the depthwise_convolution_2d() function and holds the filter weight and bias vector as parameters.

**Parameters**

- **in_channels (int)** – Number of channels of input arrays. If None, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.

- **channel_multiplier (int)** – Channel multiplier number. Number of output arrays equal in_channels * channel_multiplier.
• **ksize** *(int or pair of ints)* – Size of filters (a.k.a. kernels). ksize=k and ksize=(k, k) are equivalent.

• **stride** *(int or pair of ints)* – Stride of filter applications. stride=s and stride=(s, s) are equivalent.

• **pad** *(int or pair of ints)* – Spatial padding width for input arrays. pad=p and pad=(p, p) are equivalent.

• **nobias** *(bool)* – If True, then this link does not use the bias term.

• **initialW** *(4-D array)* – Initial weight value. If None, the default initializer is used. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.

• **initial_bias** *(1-D array)* – Initial bias value. If None, the bias is set to 0. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.

See also:

See `chainer.functions.depthwise_convolution_2d()`.

**Variables**

• **W** *(Variable)* – Weight parameter.

• **b** *(Variable)* – Bias parameter.

**Methods**

```python
__call__(x)
```
Applies the depthwise convolution layer.

**Parameters**

• **x** *(chainer.Variable or numpy.ndarray or cupy.ndarray)* – Input image.

**Returns**

Output of the depthwise convolution.

**Return type**

Variable

```python
add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
```
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```
can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

• **name** *(str)* – Name of the parameter. This name is also used as the attribute name.

• **shape** *(int or tuple of ints)* – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• **dtype** – Data type of the parameter array.
• **initializer** – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

**add_persistent**(name, value)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name**(str) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads**(link)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

- **link**(Link) – Source link object.

**children**()

Returns a generator of all child links.

**Returns** A generator object that generates all child links.

**cleargrads**()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy**()

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns** Copied link object.

**Return type** Link

**copyparams**(link)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

- **link**(Link) – Source link object.

**disable_update**()

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update**()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.
**init_scope** (*args, **kwargs*)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** (skipself=False)

Returns a generator of all links under the hierarchy.

Parameters

- **skipself (bool)** – If True, then the generator skips this link and starts with the first child link.

Returns

A generator object that generates all links.

**namedlinks** (skipself=False)

Returns a generator of all (path, link) pairs under the hierarchy.

Parameters

- **skipself (bool)** – If True, then the generator skips this link and starts with the first child link.

Returns

A generator object that generates all (path, link) pairs.

**namedparams** (include_uninit=True)

Returns a generator of all (path, param) pairs under the hierarchy.

Parameters

- **include_uninit (bool)** – If True, it also generates uninitialized parameters.

Returns

A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params** (include_uninit=True)

Returns a generator of all parameters under the link hierarchy.

Parameters

- **include_uninit (bool)** – If True, it also generates uninitialized parameters.

Returns

A generator object that generates all parameters.

**register_persistent** (name)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters

- **name (str)** – Name of the attribute to be registered.

**serialize** (serializer)

Serializes the link object.
Parameters `serializer`\(^1\) – Serializer object.

`to_cpu()`
Copies parameter variables and persistent values to CPU.
This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

Returns: self

`to_gpu(device=None)`
Copies parameter variables and persistent values to GPU.
This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

Parameters `device` – Target device specifier. If omitted, the current device is used.

Returns: self

`zerograds()`
Initializes all gradient arrays by zero.
This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads()` instead.

Attributes

`update_enabled`
True if at least one parameter has an update rule enabled.

`within_init_scope`
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

`xp`
Array module for this link.
Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

`chainer.links.DilatedConvolution2D`

class chainer.links.DilatedConvolution2D\(^2\)(in_channels, out_channels, ksize=None, stride=1, pad=0, dilate=1, nobias=False, initialW=None, initial_bias=None)

Two-dimensional dilated convolutional layer.

This link wraps the `dilated_convolution_2d()` function and holds the filter weight and bias vector as parameters.

Parameters

- `in_channels`\(^3\)\(^4\)\(^5\)\(^6\)\(^7\)\(^8\)\(^9\)\(^10\)(int or None) – Number of channels of input arrays. If None, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- `out_channels`\(^3\)\(^4\)\(^5\)\(^6\)\(^7\)\(^8\)\(^9\)\(^10\)(int) – Number of channels of output arrays.
• **ksize** (*int or pair of ints*) – Size of filters (a.k.a. kernels). ksize=k and ksize=(k, k) are equivalent.

• **stride** (*int or pair of ints*) – Stride of filter applications. stride=s and stride=(s, s) are equivalent.

• **pad** (*int or pair of ints*) – Spatial padding width for input arrays. pad=p and pad=(p, p) are equivalent.

• **dilate** (*int or pair of ints*) – Dilation factor of filter applications. dilate=d and dilate=(d, d) are equivalent.

• **nobias** (*bool*) – If True, then this link does not use the bias term.

• **initialW** (*4-D array*) – Initial weight value. If None, the default initializer is used. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.

• **initial_bias** (*1-D array*) – Initial bias value. If None, the default initializer is used. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.

See also:

See `chainer.functions.dilated_convolution_2d()` for the definition of two-dimensional dilated convolution.

**Variables**

• **W** (*Variable*) – Weight parameter.

• **b** (*Variable*) – Bias parameter.

**Example**

There are several ways to make a DilatedConvolution2D link.

Let an input vector \( x \) be:

```python
>>> x = np.arange(1 * 3 * 10 * 10, dtype='f').reshape(1, 3, 10, 10)
```

1. Give the first three arguments explicitly:

```python
>>> l = L.DilatedConvolution2D(3, 7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```

2. Omit `in_channels` or fill it with None:

The below two cases are the same.

```python
>>> l = L.DilatedConvolution2D(7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)

>>> l = L.DilatedConvolution2D(None, 7, 5)
>>> y = l(x)
>>> y.shape
(1, 7, 6, 6)
```
When you omit the first argument, you need to specify the other subsequent arguments from `stride` as keyword arguments. So the below two cases are the same.

```python
>>> l = L.DilatedConvolution2D(None, 7, 5, 1, 0, 2)
>>> y = l(x)
>>> y.shape
(1, 7, 2, 2)

>>> l = L.DilatedConvolution2D(7, 5, stride=1, pad=0, dilate=2)
>>> y = l(x)
>>> y.shape
(1, 7, 2, 2)
```

## Methods

**__call__(x)**

Applies the convolution layer.

- **Parameters**
  - `x` ([Variable](#)) – Input image.
- **Returns**
  - Output of the convolution.
- **Return type**
  - [Variable](#)

**add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)**

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a [Parameter](#) object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

- **Parameters**
  - `name` ([str](#)) – Name of the parameter. This name is also used as the attribute name.
  - `shape` ([int](#) or [tuple of ints](#)) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
  - `dtype` – Data type of the parameter array.
  - `initializer` – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

**addpersistent(name, value)**

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

- **Parameters**
• **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.

• **value** – Value to be registered.

**addgrads** (*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

- **link** (*Link*) – Source link object.

**children**()

Returns a generator of all child links.

**Returns**

A generator object that generates all child links.

**cleargrads**()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy**()

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns**

Copied link object.

**Return type** *Link*

**copyparams** (*link*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

- **link** (*Link*) – Source link object.

**disable_update**()

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update**()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

**init_scope**(*args, **kws*)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

**Example**
In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

---

**links** *(skipself=False)*

Returns a generator of all links under the hierarchy.

- **Parameters** `skipself` *(bool)* — If `True`, then the generator skips this link and starts with the first child link.

- **Returns** A generator object that generates all links.

**namedlinks** *(skipself=False)*

Returns a generator of all (path, link) pairs under the hierarchy.

- **Parameters** `skipself` *(bool)* — If `True`, then the generator skips this link and starts with the first child link.

- **Returns** A generator object that generates all (path, link) pairs.

**namedparams** *(include_uninit=True)*

Returns a generator of all (path, param) pairs under the hierarchy.

- **Parameters** `include_uninit` *(bool)* — If `True`, it also generates uninitialized parameters.

- **Returns** A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params** *(include_uninit=True)*

Returns a generator of all parameters under the link hierarchy.

- **Parameters** `include_uninit` *(bool)* — If `True`, it also generates uninitialized parameters.

- **Returns** A generator object that generates all parameters.

**register_persistent**(name)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has already been registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

- **Parameters** `name` *(str)* — Name of the attribute to be registered.

**serialize**(serializer)

Serializes the link object.

- **Parameters** `serializer` *(AbstractSerializer)* — Serializer object.

**to_cpu**()

Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

- **Returns**: self
to_gpu(device=None)

Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

Parameters

- **device** – Target device specifier. If omitted, the current device is used.

Returns: self

zerograds()

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.

Attributes

- **update_enabled** – True if at least one parameter has an update rule enabled.

- **within_init_scope** – True if the current code is inside of an initialization scope.

  See init_scope() for the details of the initialization scope.

- **xp** – Array module for this link.

  Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.EmbedID

**class** chainer.links.EmbedID(in_size, out_size, initialW=None, ignore_label=None)

Efficient linear layer for one-hot input.

This is a link that wraps the embed_id() function. This link holds the ID (word) embedding matrix \( W \) as a parameter.

Parameters

- **in_size** (int) – Number of different identifiers (a.k.a. vocabulary size).

- **out_size** (int) – Size of embedding vector.

- **initialW** (2-D array) – Initial weight value. If None, then the matrix is initialized from the standard normal distribution. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.

- **ignore_label** (int or None) – If ignore_label is an int value, \( i \)-th column of return value is filled with 0.

See also:

chainer.functions.embed_id()

Variables

- **W** (Variable) – Embedding parameter matrix.
Methods

__call__ (x)
Extracts the word embedding of given IDs.

Parameters

- **x** (Variable) – Batch vectors of IDs.

Returns

Batch of corresponding embeddings.

Return type

Variable

add_param (name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code can be replaced by the following assignment.

```python
link.add_param('W', shape=(5, 3))
```

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- **name**(str) – Name of the parameter. This name is also used as the attribute name.
- **shape**(int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent (name, value)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name**(str) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (link)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters

- **link**(Link) – Source link object.

children ()
Returns a generator of all child links.

Returns

A generator object that generates all child links.
cleargrads()
Cleareds all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy()
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Returns Copied link object.
Return type Link
copyparams(link)
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

Parameters link (Link) – Source link object.
disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope(*args, **kwds)
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))

links(skipself=False)
Returns a generator of all links under the hierarchy.

Parameters skipself (bool) – If True, then the generator skips this link and starts with the first child link.
Returns A generator object that generates all links.

**namedlinks** *(skipself=False)*

Returns a generator of all (path, link) pairs under the hierarchy.

**Parameters** skipself *(bool)* – If True, then the generator skips this link and starts with the first child link.

**Returns** A generator object that generates all (path, link) pairs.

**namedparams** *(include_uninit=True)*

Returns a generator of all (path, param) pairs under the hierarchy.

**Parameters** include_uninit *(bool)* – If True, it also generates uninitialized parameters.

**Returns** A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params** *(include_uninit=True)*

Returns a generator of all parameters under the link hierarchy.

**Parameters** include_uninit *(bool)* – If True, it also generates uninitialized parameters.

**Returns** A generator object that generates all parameters.

**register_persistent** *(name)*

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters** name *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*

Serializes the link object.

**Parameters** serializer *(AbstractSerializer)* – Serializer object.

**to_cpu** *(device=None)*

Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

**Returns** self

**to_gpu** *(device=None)*

Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

**Parameters** device – Target device specifier. If omitted, the current device is used.

**Returns** self

**zerograds** *(device=None)*

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.
Attributes

ignore_label = None

update_enabled
  True if at least one parameter has an update rule enabled.

within_init_scope
  True if the current code is inside of an initialization scope.
  See init_scope() for the details of the initialization scope.

xp
  Array module for this link.
  Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.GRU

class chainer.links.GRU (in_size, out_size, init=None, inner_init=None, bias_init=0)
  Stateful Gated Recurrent Unit function (GRU)
  This is an alias of “~chainer.links.StatefulGRU”. Its documented API is identical to the function.

Warning: In Chainer v1, GRU was stateless, as opposed to the current implementation. To align with the naming convension of LSTM links, we have changed the naming convension from Chainer v2 so that the shorthand name points the stateful links. You can use StatelessGRU for stateless version, whose implementation is identical to chainer.linksGRU in v1.

See also:

   GRU

Methods

__call__ (self, x)
  Does forward propagation.

__getitem__ (name)
  Equivalent to getattr.

add_link (name, link)
  Registers a child link to this chain.
  Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code
  
  chain.add_link('ll', L.Linear(3, 5))

  can be replaced by the following line.

  with self.init_scope():
    chain.ll = L.Linear(3, 5)

  The latter one is easier for IDEs to keep track of the attribute’s type.
Parameters

- **name (str)** – Name of the child link. This name is also used as the attribute name.
- **link (Link)** – The link object to be registered.

**add_param** *(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)*

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- **name (str)** – Name of the parameter. This name is also used as the attribute name.
- **shape (int or tuple of ints)** – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

**add_persistent** *(name, value)*

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name (str)** – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads** *(link)*

**children** *

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy** *

**copyparams** *(link)*

**disable_update** *

Disables update rules of all parameters under the link hierarchy.

This method sets the ‘attr:~chainer.UpdateRule.enabled’ flag of the update rule of each parameter variable to False.
enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope(*args, **kwds)
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example
In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (skipself=False)

namedlinks (skipself=False)

namedparams (include_uninit=True)

params (include_uninit=True)

register_persistent (name)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters name (str) – Name of the attribute to be registered.

reset_state ()

serialize (serializer)

set_state (h)

to_cpu ()

to_gpu (device=None)

zerograds ()
Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.
Attributes

update_enabled
True if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See init_scope() for the details of the initialization scope.

xp
Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.Highway
class chainer.links.Highway (in_out_size, nobias=False, activate=<function relu>, init_Wh=None, init_Wt=None, init_bh=None, init_bt=-1)
Highway module.

In highway network, two gates are added to the ordinal non-linear transformation \( H(x) = activate(W_h x + b_h) \). One gate is the transform gate \( T(x) = \sigma(W_t x + b_t) \), and the other is the carry gate \( C(x) \). For simplicity, the author defined \( C = 1 - T \). Highway module returns \( y \) defined as

\[
y = activate(W_h x + b_h) \odot \sigma(W_t x + b_t) + x \odot (1 - \sigma(W_t x + b_t))
\]

The output array has the same spatial size as the input. In order to satisfy this, \( W_h \) and \( W_t \) must be square matrices.

Parameters

- **in_out_size** (int) – Dimension of input and output vectors.
- **nobias** (bool) – If True, then this function does not use the bias.
- **activate** – Activation function of plain array. \( \tanh \) is also available.
- **init_Wh** (2-D array) – Initial weight value of plain array. If None, the default initializer is used. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.
- **init_Wt** (2-D array) – Initial weight value of transform array. If None, the default initializer is used. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.
- **init_bh** (1-D array) – Initial bias value of plain array. If None, then \( b_h \) is initialized to zero. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.
- **init_bt** (1-D array) – Initial bias value of transform array. Default value is -1 vector. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value. Negative value is recommended by the author of the paper. (e.g. -1, -3, ...).

See: Highway Networks.
Methods

__call__(x)

Computes the output of the Highway module.

Parameters x (Variable) – Input variable.

Returns Output variable. Its array has the same spatial size and the same minibatch size as the input array.

Return type Variable

__getitem__(name)

Equivalent to getattr.

add_link(name, link)

Registers a child link to this chain.

Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('l1', L.Linear(3, 5))
```

can be replaced by the following line.

```python
with self.init_scope():
    chain.l1 = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- name (str) – Name of the child link. This name is also used as the attribute name.
- link (Link) – The link object to be registered.

add_param(name, shape=None, dtype=<type ‘numpy.float32’>, initializer=None)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- name (str) – Name of the parameter. This name is also used as the attribute name.
- shape (int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- dtype – Data type of the parameter array.
- initializer – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.
add_persistent (name, value)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute
of the link.

Parameters
- name (str) – Name of the persistent value. This name is also used for the attribute name.
- value – Value to be registered.

addgrads (link)

children ()
cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy ()
copyparams (link)
disable_update ()
Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable
to False.

enable_update ()
Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope (*args, **kwds)
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain)
by an assignment. A Parameter object can be automatically registered by assigning it to an attribute
under this context manager.

Example
In most cases, the parameter registration is done in the initializer method. Using the init_scope
method, we can simply assign a Parameter object to register it to the link.

class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
register_persistent (name)
   Registers an attribute of a given name as a persistent value.

   This is a convenient method to register an existing attribute as a persistent value. If \texttt{name} has been already
   registered as a parameter, this method removes it from the list of parameter names and re-registers it as a
   persistent value.

   \textbf{Parameters} \texttt{name (str)} – Name of the attribute to be registered.

serialize (serializer)

to_cpu ()

to_gpu (device=\texttt{None})

zerograds ()
   Initializes all gradient arrays by zero.

   This method can be used for the same purpose of cleargrads, but less efficient. This method is left for
   backward compatibility.

   Deprecated since version v1.15: Use \texttt{cleargrads()} instead.

\textbf{Attributes}

update_enabled
   True if at least one parameter has an update rule enabled.

within_init_scope
   True if the current code is inside of an initialization scope.

   See \texttt{init_scope()} for the details of the initialization scope.

xp
   Array module for this link.

   Depending on which of CPU/GPU this link is on, this property returns \texttt{numpy} or \texttt{cupy}.

\texttt{chainer.links.Inception}

class \texttt{chainer.links.Inception}(\texttt{in_channels, out1, proj3, out3, proj5, out5, proj_pool,}
\texttt{conv_init=\texttt{None}, bias_init=\texttt{None}})
   Inception module of GoogLeNet.

   It applies four different functions to the input array and concatenates their outputs along the channel dimension.
   Three of them are 2D convolutions of sizes 1x1, 3x3 and 5x5. Convolution paths of 3x3 and 5x5 sizes have 1x1
   convolutions (called projections) ahead of them. The other path consists of 1x1 convolution (projection) and
   3x3 max pooling.

   The output array has the same spatial size as the input. In order to satisfy this, Inception module uses appropriate
   padding for each convolution and pooling.

   See: Going Deeper with Convolutions.

   \textbf{Parameters}

   \begin{itemize}
   \item \texttt{in_channels (int or \texttt{None})} – Number of channels of input arrays.
   \item \texttt{out1 (int)} – Output size of 1x1 convolution path.
   \item \texttt{proj3 (int)} – Projection size of 3x3 convolution path.
   \end{itemize}
- **out3** (*int*) – Output size of 3x3 convolution path.
- **proj5** (*int*) – Projection size of 5x5 convolution path.
- **out5** (*int*) – Output size of 5x5 convolution path.
- **proj_pool** (*int*) – Projection size of max pooling path.
- **conv_init** – A callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value. It is used for initialization of the convolution matrix weights. Maybe be `None` to use default initialization.
- **bias_init** – A callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value. It is used for initialization of the convolution bias weights. Maybe be `None` to use default initialization.

**Methods**

__call__ (*x*)
Computes the output of the Inception module.

**Parameters**
- **x** (*Variable*) – Input variable.

**Returns**
Output variable. Its array has the same spatial size and the same minibatch size as the input array. The channel dimension has size `out1 + out3 + out5 + proj_pool`.

**Return type** *Variable*

__getitem__ (*name*)
Equivalent to getattr.

add_link (*name, link*)
Registers a child link to this chain.

Depreciated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('l1', L.Linear(3, 5))
```

can be replaced by the following line.

```python
with self.init_scope():
    chain.l1 = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**
- **name** (*str*) – Name of the child link. This name is also used as the attribute name.
- **link** (*Link*) – The link object to be registered.

add_param (*name, shape=None, dtype=<type 'numpy.float32'>, initializer=None*)
Registers a parameter to the link.

Depreciated since version v2.0.0: Assign a `Parameter` object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- **name (str)** – Name of the parameter. This name is also used as the attribute name.
- **shape (int or tuple of ints)** – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent (name, value)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name (str)** – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (link)

children()

cleargrads()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy()

copyparams (link)

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope (*args, **kwds)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for :class:`Chain`) by an assignment. A :class:`Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example
In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

- **links** `(skipself=False)`: 
- **namedlinks** `(skipself=False)`: 
- **namedparams** `(include_uninit=True)`: 
- **params** `(include_uninit=True)`: 
- **register_persistent** `(name)`: Registers an attribute of a given name as a persistent value.
  
  This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

  **Parameters**
  
  - **name**: `str` – Name of the attribute to be registered.

- **serialize** `(serializer)`: 

- **to_cpu()**: 

- **to_gpu** `(device=None)`: 

- **zerograds()**: 

  Initializes all gradient arrays by zero.

  This method can be used for the same purpose of `cleargrads`, but less efficient. This method is left for backward compatibility.

  Deprecated since version v1.15: Use `cleargrads()` instead.

**Attributes**

- **update_enabled**: 

  True if at least one parameter has an update rule enabled.

- **within_init_scope**: 

  True if the current code is inside of an initialization scope.

  See `init_scope()` for the details of the initialization scope.

- **xp**: 

  Array module for this link.

  Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.
### chainer.links.InceptionBN

**class** `chainer.links.InceptionBN` *(in_channels, out1, proj3, out3, proj33, out33, pooltype, proj_pool=None, stride=1, conv_init=None, dtype=<type 'numpy.float32'>)*

Inception module of the new GoogLeNet with BatchNormalization.

This chain acts like *Inception*, while InceptionBN uses the *BatchNormalization* on top of each convolution, the 5x5 convolution path is replaced by two consecutive 3x3 convolution applications, and the pooling method is configurable.

**See:** Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

**Parameters**

- **in_channels** *(int or None)* – Number of channels of input arrays.
- **out1** *(int)* – Output size of the 1x1 convolution path.
- **proj3** *(int)* – Projection size of the single 3x3 convolution path.
- **out3** *(int)* – Output size of the single 3x3 convolution path.
- **proj33** *(int)* – Projection size of the double 3x3 convolutions path.
- **out33** *(int)* – Output size of the double 3x3 convolutions path.
- **pooltype** *(str)* – Pooling type. It must be either 'max' or 'avg'.
- **proj_pool** *(int or None)* – Projection size in the pooling path. If None, no projection is done.
- **stride** *(int)* – Stride parameter of the last convolution of each path.
- **conv_init** – A callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value. It is used for initialization of the convolution matrix weights. Maybe be None to use default initialization.
- **dtype** *(numpy.dtype)* – Type to use in `~batch_normalization.BatchNormalization`.

**See also:**

*Inception*

**Methods**

- **__call__(x)**
- **__getitem__(name)**
  
  Equivalent to getattr.

- **add_link**(name, link)**
  
  Registers a child link to this chain.

  Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

  ```python
  chain.add_link('l1', L.Linear(3, 5))
  ```

  can be replaced by the following line.
with self.init_scope():
    chain.l1 = L.Linear(3, 5)

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- name (str) – Name of the child link. This name is also used as the attribute name.
- link (Link) – The link object to be registered.

add_param (name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- name (str) – Name of the parameter. This name is also used as the attribute name.
- shape (int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- dtype – Data type of the parameter array.
- initializer – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent (name, value)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- name (str) – Name of the persistent value. This name is also used for the attribute name.
- value – Value to be registered.

addgrads (link)

children ()

cleargrads ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy ()

copyparams (link)
disable_update()
Disables update rules of all parameters under the link hierarchy.
This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

enable_update()
Enables update rules of all parameters under the link hierarchy.
This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope(*args, **kwds)
Creates an initialization scope.
This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example
In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

class MyLink(chainer.Link):
  def __init__(self):
    super().__init__()
    with self.init_scope():
      self.W = chainer.Parameter(0, (10, 5))
      self.b = chainer.Parameter(0, (5,))

links(skipself=False)
namedlinks(skipself=False)
namedparams(include_uninit=True)
params(include_uninit=True)
register_persistent(name)
Registers an attribute of a given name as a persistent value.
This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters

name (str) – Name of the attribute to be registered.
serialize(name (str) – Name of the attribute to be registered.

to_cpu()
to_gpu(device=None)
zerograds()
Initializes all gradient arrays by zero.
This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.
Deprecation since version v1.15: Use cleargrads() instead.
Attributes

**update_enabled**
True if at least one parameter has an update rule enabled.

**within_init_scope**
True if the current code is inside of an initialization scope.
See `init_scope()` for the details of the initialization scope.

**xp**
Array module for this link.
Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

### chainer.links.Linear

class chainer.links.Linear(`in_size`, `out_size=None`, `nobias=False`, `initialW=None`, `initial_bias=None`)
Linear layer (a.k.a. fully-connected layer).
This is a link that wraps the `linear()` function, and holds a weight matrix \( W \) and optionally a bias vector \( b \) as parameters.
The weight matrix \( W \) is initialized with i.i.d. Gaussian samples, each of which has zero mean and deviation \( \sqrt{1/\text{in}_\text{size}} \)

Parameters

- **in_size** (*int or None*) – Dimension of input vectors. If `None`, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_size** (*int*) – Dimension of output vectors.
- **nobias** (*bool*) – If `True`, then this function does not use the bias.
- **initialW** (*2-D array*) – Initial weight value. If `None`, then the default initializer is used. May also be a callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value.
- **initial_bias** (*1-D array*) – Initial bias value. If `None`, the bias vector is initialized to zero. May also be a callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value.

See also:
linear()

Variables

- \( W \) (Variable) – Weight parameter.
- \( b \) (Variable) – Bias parameter.

### Example

There are several ways to make a Linear link.

Define an input vector \( x \) as:

```python
>>> x = np.array([[0, 1, 2, 3, 4]], 'f')
```

1. Give the first two arguments explicitly:
Those numbers are considered as the input size and the output size.
2. **Omit in_size (give the output size only as the first argument)**

   or fill it with None:

   In this case, the size of second axis of \( x \) is used as the input size. So the below two cases are the same.

   ```python
   >>> l = L.Linear(10)
   >>> y = l(x)
   >>> y.shape
   (1, 10)
   ```

   ```python
   >>> l = L.Linear(None, 10)
   >>> y = l(x)
   >>> y.shape
   (1, 10)
   ```

   When you omit the first argument, you need to specify the other subsequent arguments from nobias as keyword arguments. So the below two cases are the same.

   ```python
   >>> l = L.Linear(None, 10, False, None, 0)
   >>> y = l(x)
   >>> y.shape
   (1, 10)
   ```

   ```python
   >>> l = L.Linear(10, nobias=False, initialW=None, initial_bias=0)
   >>> y = l(x)
   >>> y.shape
   (1, 10)
   ```

---

**Methods**

```python
def __call__(x)
    Applies the linear layer.

    Parameters
    ----------
    x : Variable
        Batch of input vectors.

    Returns
    -------
    Output of the linear layer.

    Return type
    -----------
    Variable
```

```python
def add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
    Registers a parameter to the link.

    Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

    ```python
    link.add_param('W', shape=(5, 3))
    ```

    can be replaced by the following assignment.
```
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

• **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
• **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
• **dtype** – Data type of the parameter array.
• **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

**add_persistent** (*name, value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

• **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
• **value** – Value to be registered.

**addgrads** (*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters **link** (*Link*) – Source link object.

**children**()

Returns a generator of all child links.

**Returns** A generator object that generates all child links.

**cleargrads**()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy**()

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns** Copied link object.

**Return type** *Link*

**copyparams** (*link*)

Copies all parameters from given link.
This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

`link (Link)` – Source link object.

**disable_update()**

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`-chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

**enable_update()**

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to `True`.

**init_scope(*args, **kwds)**

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links(skipself=False)**

Returns a generator of all links under the hierarchy.

**Parameters**

`skipself (bool)` – If `True`, then the generator skips this link and starts with the first child link.

**Returns** A generator object that generates all links.

**namedlinks(skipself=False)**

Returns a generator of all (path, link) pairs under the hierarchy.

**Parameters**

`skipself (bool)` – If `True`, then the generator skips this link and starts with the first child link.

**Returns** A generator object that generates all (path, link) pairs.

**namedparams(include_uninit=True)**

Returns a generator of all (path, param) pairs under the hierarchy.

**Parameters**

`include_uninit (bool)` – If `True`, it also generates uninitialized parameters.

**Returns** A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params(include_uninit=True)**

Returns a generator of all parameters under the link hierarchy.
Parameters `include_uninit (bool)` – If True, it also generates uninitialized parameters.

Returns A generator object that generates all parameters.

`register_persistent (name)`
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters `name (str)` – Name of the attribute to be registered.

`serialize (serializer)`
Serializes the link object.

Parameters `serializer (AbstractSerializer)` – Serializer object.

`to_cpu ()`
Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

Returns: self

`to_gpu (device=None)`
Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

Parameters `device` – Target device specifier. If omitted, the current device is used.

Returns: self

`zerograds ()`
Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads ()` instead.

Attributes

`update_enabled`
True if at least one parameter has an update rule enabled.

`within_init_scope`
True if the current code is inside of an initialization scope.

See `init_scope ()` for the details of the initialization scope.

`xp`
Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.
chainer.links.LSTM

class chainer.links.LSTM(in_size, out_size=None, **kwargs)

Fully-connected LSTM layer.

This is a fully-connected LSTM layer as a chain. Unlike the lstm() function, which is defined as a stateless activation function, this chain holds upward and lateral connections as child links.

It also maintains states, including the cell state and the output at the previous time step. Therefore, it can be used as a stateful LSTM.

This link supports variable length inputs. The mini-batch size of the current input must be equal to or smaller than that of the previous one. The mini-batch size of $c$ and $h$ is determined as that of the first input $x$. When mini-batch size of $i$-th input is smaller than that of the previous input, this link only updates $c[0:len(x)]$ and $h[0:len(x)]$ and doesn’t change the rest of $c$ and $h$. So, please sort input sequences in descending order of lengths before applying the function.

Parameters

- **in_size** (int) – Dimension of input vectors. If it is None or omitted, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_size** (int) – Dimensionality of output vectors.
- **lateral_init** – A callable that takes numpy.ndarray or cupy.ndarray and edits its value. It is used for initialization of the lateral connections. May be None to use default initialization.
- **upward_init** – A callable that takes numpy.ndarray or cupy.ndarray and edits its value. It is used for initialization of the upward connections. May be None to use default initialization.
- **bias_init** – A callable that takes numpy.ndarray or cupy.ndarray and edits its value. It is used for initialization of the biases of cell input, input gate and output gate and gates of the upward connection. May be a scalar, in that case, the bias is initialized by this value. If it is None, the cell-input bias is initialized to zero.
- **forget_bias_init** – A callable that takes numpy.ndarray or cupy.ndarray and edits its value. It is used for initialization of the biases of the forget gate of the upward connection. May be a scalar, in that case, the bias is initialized by this value. If it is None, the forget bias is initialized to one.

Variables

- **upward** (Linear) – Linear layer of upward connections.
- **lateral** (Linear) – Linear layer of lateral connections.
- **c** (Variable) – Cell states of LSTM units.
- **h** (Variable) – Output at the previous time step.

Methods

__call__(x)

Updates the internal state and returns the LSTM outputs.

Parameters **x** (Variable) – A new batch from the input sequence.

Returns Outputs of updated LSTM units.
Return type  Variable

__getitem__ (name)
Equivalent to getattr.

add_link (name, link)
Registers a child link to this chain.

Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```
chain.add_link('l1', L.Linear(3, 5))
```

can be replaced by the following line.

```
with self.init_scope():
    chain.l1 = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters
- **name** (str) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

add_param (name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters
- **name** (str) – Name of the parameter. This name is also used as the attribute name.
- **shape** (int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent (name, value)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters
- **name** (str) – Name of the persistent value. This name is also used for the attribute name.
• value – Value to be registered.

addgrads (link)

children ()

cleargrads ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy ()

copyparams (link)

disable_update ()

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable
to False.

enable_update ()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope (*args, **kwds)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`)by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (skipself=False)

namedlinks (skipself=False)

namedparams (include_uninit=True)

params (include_uninit=True)

register_persistent (name)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been alreadyregistered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters name (str) – Name of the attribute to be registered.
reset_state()
Resets the internal state.
It sets None to the c and h attributes.

serialize(serializer)

set_state(c, h)
Sets the internal state.
It sets the c and h attributes.

Parameters
- c (Variable) – A new cell states of LSTM units.
- h (Variable) – A new output at the previous time step.

to_cpu()
to_gpu(device=None)
zerograds()
Initializes all gradient arrays by zero.
This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.
Deprecated since version v1.15: Use cleargrads() instead.

Attributes

update_enabled
True if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.
See init_scope() for the details of the initialization scope.

xp
Array module for this link.
Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.MLPConvolution2D

class chainer.links.MLPConvolution2D(self, in_channels, out_channels, ksize=None, stride=1, pad=0, activation=relu.relu, conv_init=None, bias_init=None)

Two-dimensional MLP convolution layer of Network in Network.
This is an “mlpconv” layer from the Network in Network paper. This layer is a two-dimensional convolution layer followed by 1x1 convolution layers and interleaved activation functions.
Note that it does not apply the activation function to the output of the last 1x1 convolution layer.

Parameters
- in_channels (int or None) – Number of channels of input arrays. If it is None or omitted, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
• **out_channels** *(tuple of ints)* – Tuple of number of channels. The i-th integer indicates the number of filters of the i-th convolution.

• **ksize** *(int or pair of ints)* – Size of filters (a.k.a. kernels) of the first convolution layer. ksize=k and ksize=(k, k) are equivalent.

• **stride** *(int or pair of ints)* – Stride of filter applications at the first convolution layer. stride=s and stride=(s, s) are equivalent.

• **pad** *(int or pair of ints)* – Spatial padding width for input arrays at the first convolution layer. pad=p and pad=(p, p) are equivalent.

• **activation** *(function)* – Activation function for internal hidden units. Note that this function is not applied to the output of this link.

• **conv_init** – An initializer of weight matrices passed to the convolution layers. This option must be specified as a keyword argument.

• **bias_init** – An initializer of bias vectors passed to the convolution layers. This option must be specified as a keyword argument.

See: Network in Network.

**Variables**

**activation** *(function)* – Activation function.

**Methods**

**__call__(x)**

Computes the output of the mlpconv layer.

- **Parameters**
  - x *(Variable)* – Input image.

- **Returns**
  - Output of the mlpconv layer.

- **Return type**
  - Variable

**__getitem__(index)**

Returns the child at given index.

- **Parameters**
  - index *(int)* – Index of the child in the list.

- **Returns**
  - The index-th child link.

- **Return type**
  - Link

**__len__()**

Returns the number of children.

**__iter__()**

**add_link(link)**

Registers a child link and adds it to the tail of the list.

- **Parameters**
  - link *(Link)* – The link object to be registered.

**add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)**

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```
can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** *(str)* – Name of the parameter. This name is also used as the attribute name.
- **shape** *(int or tuple of ints)* – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

**add_persistent**(name, value)

 Registers a persistent value to the link.

 The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

 **Parameters**

- **name** *(str)* – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads**(link)

**append**(link)

 Registers a child link and adds it to the tail of the list.

 This is equivalent to `add_link()`. This method has been added to emulate the list interface.

 **Parameters**

- **link** *(Link)* – The link object to be registered.

**children**()

**cleargrads**()

 Clears all gradient arrays.

 This method should be called before the backward computation at every iteration of the optimization.

**copy**()

**copyparams**(link)

**disable_update**()

 Disables update rules of all parameters under the link hierarchy.

 This method sets the `attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update**()

 Enables update rules of all parameters under the link hierarchy.

 This method sets the `enabled` flag of the update rule of each parameter variable to True.

**init_scope**(args, **kwds)

 Creates an initialization scope.

3.5. Standard Link implementations
This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** *(skipself=False)*

**namedlinks** *(skipself=False)*

**namedparams** *(include_uninit=True)*

**params** *(include_uninit=True)*

**register_persistent**(name)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**

- **name**(str) – Name of the attribute to be registered.

**serialize**(serializer)

**to_cpu**()

**to_gpu**(device=None)

**zerograds**()

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.

**Attributes**

**update_enabled**

True if at least one parameter has an update rule enabled.

**within_init_scope**

True if the current code is inside of an initialization scope.

See init_scope() for the details of the initialization scope.

**xp**

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.
chainer.links.NStepBiGRU

class chainer.links.NStepBiGRU(self, n_layers, in_size, out_size, dropout)
Stacked Bi-directional GRU for sequences.

This link is stacked version of Bi-directional GRU for sequences. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike chainer.functions.n_step_bigru(), this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of chainer.Variable holding sequences.

Warning: use_cudnn argument is not supported anymore since v2. Instead, use chainer.using_config('use_cudnn', use_cudnn). See chainer.using_config().

Parameters

• n_layers (int) – Number of layers.
• in_size (int) – Dimensionality of input vectors.
• out_size (int) – Dimensionality of hidden states and output vectors.
• dropout (float) – Dropout ratio.

See also:
chainer.functions.n_step_bigru()

Methods

__call__(self, hx, xs)
Calculate all hidden states and cell states.

Warning: train argument is not supported anymore since v2. Instead, use chainer.using_config('train', train). See chainer.using_config().

Parameters

• hx (Variable or None) – Initial hidden states. If None is specified zero-vector is used.
• xs (list of ~chianer.Variable) – List of input sequences. Each element xs[i] is a chainer.Variable holding a sequence.

__getitem__(index)
Returns the child at given index.

Parameters index (int) – Index of the child in the list.

Returns The index-th child link.

Return type Link

__len__()
Returns the number of children.
__iter__()  

add_link(link)  
Registers a child link and adds it to the tail of the list.

Parameters
link (Link) – The link object to be registered.

add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)  
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters
• name (str) – Name of the parameter. This name is also used as the attribute name.
• shape (int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
• dtype – Data type of the parameter array.
• initializer – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent(name, value)  
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters
• name (str) – Name of the persistent value. This name is also used for the attribute name.
• value – Value to be registered.

addgrads(link)  
append(link)  
Registers a child link and adds it to the tail of the list.

This is equivalent to add_link(). This method has been added to emulate the list interface.

Parameters
link (Link) – The link object to be registered.

cleargrads()  
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy()  
copyparams(link)
disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

init_hx(xs)

init_scope(*args, **kwds)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(skipself=False)

namedlinks(skipself=False)

namedparams(include_uninit=True)

params(include_uninit=True)

register_persistent(name)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters name (str) – Name of the attribute to be registered.

serialize(name)

to_cpu()

to_gpu(device=None)

zerograds()

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.
Attributes

**update_enabled**
True if at least one parameter has an update rule enabled.

**within_init_scope**
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

**xp**
Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

**chainer.links.NStepBiLSTM**

**class** `chainer.links.NStepBiLSTM(self, n_layers, in_size, out_size, dropout)`

Stacked Bi-directional LSTM for sequences.

This link is stacked version of Bi-directional LSTM for sequences. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike `chainer.functions.n_step_bilstm()`, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of `chainer.Variable` holding sequences.

**Warning:** use_cudnn argument is not supported anymore since v2. Instead, use `chainer.using_config('use_cudnn', use_cudnn)`.

See also:
`chainer.functions.n_step_bilstm()`

**Methods**

```
__call__(self, hx, cx, xs)
```

Calculate all hidden states and cell states.

**Warning:** train argument is not supported anymore since v2. Instead, use `chainer.using_config('train', train)`.

Parameters
• **hx** *(Variable or None)* — Initial hidden states. If None is specified zero-vector is used.

• **cx** *(Variable or None)* — Initial cell states. If None is specified zero-vector is used.

• **xs** *(list of ~chainer.Variable)* — List of input sequences. Each element xs[i] is a chainer.Variable holding a sequence.

__getitem__(index)

Returns the child at given index.

Parameters **index** *(int)* — Index of the child in the list.

Returns The index-th child link.

Return type **Link**

__len__()

Returns the number of children.

__iter__()

add_link(link)

Registers a child link and adds it to the tail of the list.

Parameters **link** *(Link)* — The link object to be registered.

add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

• **name** *(str)* — Name of the parameter. This name is also used as the attribute name.

• **shape** *(int or tuple of ints)* — Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• **dtype** — Data type of the parameter array.

• **initializer** — If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent(name, value)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

• **name** *(str)* — Name of the persistent value. This name is also used for the attribute name.
• **value** – Value to be registered.

**addgrads** *(link)*

**append** *(link)*

Registers a child link and adds it to the tail of the list.

This is equivalent to *add_link()* method. This method has been added to emulate the *list* interface.

**Parameters**

- **link** *(Link)* – The link object to be registered.

**children** *

**cleargrads** *

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy** *

**copyparams** *(link)*

**disable_update** *

Disables update rules of all parameters under the link hierarchy.

This method sets the *chainer.UpdateRule.enabled* flag of the update rule of each parameter variable
to False.

**enable_update** *

Enables update rules of all parameters under the link hierarchy.

This method sets the *enabled* flag of the update rule of each parameter variable to True.

**init_hx** *(xs)*

**init_scope** *(*args, **kwds)*

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*)
by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute
under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the *init_scope*
method, we can simply assign a *Parameter* object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** *(skipself=False)*

**namedlinks** *(skipself=False)*

**namedparams** *(include_uninit=True)*

**params** *(include_uninit=True)*
**register_persistent** *(name)*

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If *name* has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**

- **name** *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*

to_cpu()

to_gpu *(device=None)*

zerograds()

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.

**Attributes**

- **update_enabled**

  True if at least one parameter has an update rule enabled.

- **within_init_scope**

  True if the current code is inside of an initialization scope.

  See init_scope() for the details of the initialization scope.

- **xp**

  Array module for this link.

  Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

**chainer.links.NStepBiRNNReLU**

**class** chainer.links.NStepBiRNNReLU *(self, n_layers, in_size, out_size, dropout)*

Stacked Bi-directional RNN for sequences.

This link is stacked version of Bi-directional RNN for sequences. Note that the activation function is relu. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike chainer.functions.n_step_birnn(), this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of chainer.Variable holding sequences.

**Warning:** use_cudnn argument is not supported anymore since v2. Instead, use chainer.using_config('use_cudnn', use_cudnn). See chainer.using_config().

**Parameters**

- **n_layers** *(int)* – Number of layers.
- **in_size** *(int)* – Dimensionality of input vectors.
• **out_size** (*int*) – Dimensionality of hidden states and output vectors.
• **dropout** (*float*) – Dropout ratio.

See also:

chainer.functions.n_step_birnn()

**Methods**

__call__ (self, hx, xs)

Calculate all hidden states and cell states.

**Warning:** train argument is not supported anymore since v2. Instead, use chainer.
using_config('train', train). See chainer.using_config().

**Parameters**

• **hx** (*Variable or None*) – Initial hidden states. If None is specified zero-vector is
  used.

• **xs** (*list of ~chainer.Variable*) – List of input sequences. Each element
  xs[i] is a chainer.Variable holding a sequence.

__getitem__ (index)

Returns the child at given index.

**Parameters**

**index** (*int*) – Index of the child in the list.

**Returns**

The index-th child link.

**Return type** Link

__len__()

Returns the number of children.

__iter__()

add_link (link)

Registers a child link and adds it to the tail of the list.

**Parameters**

**link** (Link) – The link object to be registered.

add_param (name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an
initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**
- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, *dtype* argument is ignored.

### add_persistent (*name, value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

### addgrads (*link*)

**append (*link*)**

Registers a child link and adds it to the tail of the list.

This is equivalent to *add_link().* This method has been added to emulate the list interface.

**Parameters**

- **link** (*Link*) – The link object to be registered.

### children()

**cleargrads()**

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

### copy()

**copyparams (*link*)**

**disable_update()**

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update()**

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

### init_hx (*xs*)

**init_scope (**args, **kwds**)**

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

**Example**
In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

`links` *(skipself=False)*

`namedlinks` *(skipself=False)*

`namedparams` *(include_uninit=True)*

`params` *(include_uninit=True)*

`register_persistent` *(name)*

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**

- `name` *(str)* – Name of the attribute to be registered.

`serialize` *(serializer)*

`to_cpu()`

`to_gpu` *(device=None)*

`zero_grads()`

Initializes all gradient arrays by zero.

This method can be used for the same purpose of `clear_grads`, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `clear_grads()` instead.

**Attributes**

- `update_enabled`  
  True if at least one parameter has an update rule enabled.

- `within_init_scope`  
  True if the current code is inside of an initialization scope.

  See `init_scope()` for the details of the initialization scope.

- `xp`  
  Array module for this link.

  Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

**chainer.links.NStepBiRNNTanh**

```python
class chainer.links.NStepBiRNNTanh(chainer.Link):  
    def __init__(self, n_layers, in_size, out_size, dropout):
        Stacked Bi-directional RNN for sequences.
```
This link is stacked version of Bi-directional RNN for sequences. Note that the activation function is \( \text{tanh} \). It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike `chainer.functions.n_step_birnn()`, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of `chainer.Variable` holding sequences.

---

**Warning:** use_cudnn argument is not supported anymore since v2. Instead, use `chainer.using_config('use_cudnn', use_cudnn)`. See `chainer.using_config()`.

---

**Parameters**

- `n_layers (int)` – Number of layers.
- `in_size (int)` – Dimensionality of input vectors.
- `out_size (int)` – Dimensionality of hidden states and output vectors.
- `dropout (float)` – Dropout ratio.
- `use_cudnn (bool)` – Use cuDNN.

See also:

`chainer.functions.n_step_birnn()`

**Methods**

**`__call__`**: (self, hx, xs)

Calculate all hidden states and cell states.

---

**Warning:** train argument is not supported anymore since v2. Instead, use `chainer.using_config('train', train)`. See `chainer.using_config()`.

---

**Parameters**

- `hx (Variable or None)` – Initial hidden states. If `None` is specified zero-vector is used.
- `xs (list of ~chainer.Variable)` – List of input sequences. Each element `xs[i]` is a `chainer.Variable` holding a sequence.

**`__getitem__`**: (index)

Returns the child at given index.

Parameters

- `index (int)` – Index of the child in the list.

Returns

The `index`-th child link.

Return type **Link**

**`__len__`**: ()

Returns the number of children.

**`__iter__`**: ()

**add_link**(link)

Registers a child link and adds it to the tail of the list.
The link object to be registered.

Parameters

- **`add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)`**
  Registers a parameter to the link.
  
  Deprecated since version v2.0.0: Assign a `Parameter` object directly to an attribute within an initialization scope instead. For example, the following code can be replaced by the following assignment.

```python
link.add_param('W', shape=(5, 3))
```

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- **`name (str)`** – Name of the parameter. This name is also used as the attribute name.
- **`shape (int or tuple of ints)`** – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **`dtype`** – Data type of the parameter array.
- **`initializer`** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **`name (str)`** – Name of the persistent value. This name is also used for the attribute name.
- **`value`** – Value to be registered.

### `addgrads(link)`

### `append(link)`

Registers a child link and adds it to the tail of the list.

This is equivalent to `add_link()`. This method has been added to emulate the `list` interface.

Parameters

- **`link (Link)`** – The link object to be registered.

### `children()`

Cleans all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

### `copy()`

### `copyparams(link)`

### `disable_update()`

Disables update rules of all parameters under the link hierarchy.
This method sets the `attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

**enable_update()**

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

**init_hx(xs)**

**init_scope(*args, **kwds)**

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links**(skipself=False)

**namedlinks**(skipself=False)

**namedparams** (include_uninit=True)

**params** (include_uninit=True)

**register_persistent**(name)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters

- **name**(str) – Name of the attribute to be registered.

**serialize**(serializer)

**to_cpu()**

**to_gpu**(device=None)

**zerograds()**

Initializes all gradient arrays by zero.

This method can be used for the same purpose of `cleargrads`, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads()` instead.
Attributes

**update_enabled**
True if at least one parameter has an update rule enabled.

**within_init_scope**
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

**xp**
Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

### chainer.links.NStepGRU

#### class chainer.links.NStepGRU

```python
class chainer.links.NStepGRU(self, n_layers, in_size, out_size, dropout)
```

Stacked Uni-directional GRU for sequences.

This link is stacked version of Uni-directional GRU for sequences. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike `chainer.functions.n_step_gru()`, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of `chainer.Variable` holding sequences.

**Warning:** use_cudnn argument is not supported anymore since v2. Instead, use `chainer.using_config('use_cudnn', use_cudnn)`. See `chainer.using_config()`.

#### Parameters

- **n_layers** (`int`) – Number of layers.
- **in_size** (`int`) – Dimensionality of input vectors.
- **out_size** (`int`) – Dimensionality of hidden states and output vectors.
- **dropout** (`float`) – Dropout ratio.

See also:

`chainer.functions.n_step_gru()`

#### Methods

```python
__call__(self, hx, xs)
```

Calculate all hidden states and cell states.

**Warning:** train argument is not supported anymore since v2. Instead, use `chainer.using_config('train', train)`. See `chainer.using_config()`.

#### Parameters
• **hx** (*Variable or None*) – Initial hidden states. If None is specified zero-vector is used.

• **xs** (*list of ~chainer.Variable*) – List of input sequences. Each element xs[i] is a *chainer.Variable* holding a sequence.

**__getitem__**(index)
Returns the child at given index.

**Parameters**
*index* (*int*) – Index of the child in the list.

**Returns**
The index-th child link.

**Return type** *Link*

**__len__**( )
Returns the number of children.

**__iter__**( )

**add_link**(link)
Registers a child link and adds it to the tail of the list.

**Parameters**
*link* (*Link*) – The link object to be registered.

**add_param**(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a *Parameter* object directly to an attribute within an *initialization scope* instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

• **name** (*str*) – Name of the parameter. This name is also used as the attribute name.

• **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• **dtype** – Data type of the parameter array.

• **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

**add_persistent**(name, value)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

• **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.

• **value** – Value to be registered.
addgrads \texttt{(link)}

append \texttt{(link)}

Registers a child link and adds it to the tail of the list.

This is equivalent to \texttt{add\_link()}. This method has been added to emulate the \texttt{list} interface.

\textbf{Parameters} link \texttt{(Link)} – The link object to be registered.

children

cleargrads

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy

copyparams \texttt{(link)}

disable\_update

Disables update rules of all parameters under the link hierarchy.

This method sets the `attr:-chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to \texttt{False}.

enable\_update

Enables update rules of all parameters under the link hierarchy.

This method sets the \texttt{enabled} flag of the update rule of each parameter variable to \texttt{True}.

\textbf{Example}

In most cases, the parameter registration is done in the initializer method. Using the \texttt{init\_scope} method, we can simply assign a \texttt{Parameter} object to register it to the link.

\begin{verbatim}
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
\end{verbatim}

links \texttt{(skipself=False)}

namedlinks \texttt{(skipself=False)}

namedparams \texttt{(include_uninit=True)}

params \texttt{(include_uninit=True)}

register\_persistent \texttt{(name)}

Registers an attribute of a given name as a persistent value.
This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**

- `name` *(str)* – Name of the attribute to be registered.

- `serialize` *(serializer)*

- `to_cpu` *()*

- `to_gpu` *(device=None)*

- `zerograds` *()*

  Initializes all gradient arrays by zero.

  This method can be used for the same purpose of `cleargrads`, but less efficient. This method is left for backward compatibility.

  Deprecated since version v1.15: Use `cleargrads()` instead.

**Attributes**

- `update_enabled` *()*

  True if at least one parameter has an update rule enabled.

- `within_init_scope` *()*

  True if the current code is inside of an initialization scope.

  See `init_scope()` for the details of the initialization scope.

- `xp` *()*

  Array module for this link.

  Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

---

**chainer.links.NStepLSTM**

**class** `chainer.links.NStepLSTM` *(self, n_layers, in_size, out_size, dropout)*

Stacked Uni-directional LSTM for sequences.

This link is stacked version of Uni-directional LSTM for sequences. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike `chainer.functions.n_step_lstm()`, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of `chainer.Variable` holding sequences.

**Warning:** use_cudnn argument is not supported anymore since v2. Instead, use `chainer.using_config('use_cudnn', use_cudnn)`. See `chainer.using_config()`.

**Parameters**

- `n_layers` *(int)* – Number of layers.

- `in_size` *(int)* – Dimensionality of input vectors.

- `out_size` *(int)* – Dimensionality of hidden states and output vectors.

- `dropout` *(float)* – Dropout ratio.
See also:

`chainer.functions.n_step_lstm()`

Methods

`__call__`(self, hx, cx, xs)

Calculate all hidden states and cell states.

**Warning:** `train` argument is not supported anymore since v2. Instead, use `chainer.using_config('train', train)`. See `chainer.using_config()`.

**Parameters**

- **hx** (*Variable or None*) – Initial hidden states. If None is specified zero-vector is used.
- **cx** (*Variable or None*) – Initial cell states. If None is specified zero-vector is used.
- **xs** (*list of ~chainer.Variable*) – List of input sequences. Each element `xs[i]` is a `chainer.Variable` holding a sequence.

`__getitem__`(index)

Returns the child at given index.

**Parameters**

- **index** (*int*) – Index of the child in the list.

**Returns**

The `index`-th child link.

**Return type**

`Link`

`__len__`()

Returns the number of children.

`__iter__`()

`add_link`(link)

Registers a child link and adds it to the tail of the list.

**Parameters**

- **link** (*Link*) – The link object to be registered.

`add_param`(name, shape=None, dtype=\texttt{<type 'numpy.float32'>}, initializer=None)

Registers a parameter to the link.

**Deprecated since version v2.0.0:** Assign a `Parameter` object directly to an attribute within an initialization scope instead. For example, the following code can be replaced by the following assignment.

```python
link.add_param('W', shape=(5, 3))
```

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
• **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• **dtype** – Data type of the parameter array.

• **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, **dtype** argument is ignored.

**add_persistent** *(name, value)*

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

• **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.

• **value** – Value to be registered.

**addgrads** *(link)*

**append** *(link)*

Registers a child link and adds it to the tail of the list.

This is equivalent to **add_link** *(link)*. This method has been added to emulate the **list** interface.

**Parameters** **link** *(Link)* – The link object to be registered.

**children** *

**cleargrads** *

Cleans all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy** *

**copyparams** *(link)*

**disable_update** *

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update** *

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

**init_hx** *(xs)*

**init_scope** *(args, **kwds)*

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for **Chain**) by an assignment. A **Parameter** object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the **init_scope** method, we can simply assign a **Parameter** object to register it to the link.
```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** *(skipself=False)*

**namedlinks** *(skipself=False)*

**namedparams** *(include_uninit=True)*

**params** *(include_uninit=True)*

**register_persistent** *(name)*

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**

- **name** *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*

**to_cpu** *

**to_gpu** *(device=None)*

**zerograds** *

Initializes all gradient arrays by zero.

This method can be used for the same purpose of `cleargrads`, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads()` instead.

**Attributes**

- **update_enabled**
    - True if at least one parameter has an update rule enabled.

- **within_init_scope**
    - True if the current code is inside of an initialization scope.
    - See `init_scope()` for the details of the initialization scope.

- **xp**
    - Array module for this link.
    - Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

---

`chainer.links.NStepRNNReLU`

```python
class chainer.links.NStepRNNReLU(self, n_layers, in_size, out_size, dropout)
    Stacked Uni-directional RNN for sequences.
    This link is stacked version of Uni-directional RNN for sequences. Note that the activation function is `relu`. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.
```
Unlike `chainer.functions.n_step_rnn()`, this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of `chainer.Variable` holding sequences.

**Warning:** `use_cudnn` argument is not supported anymore since v2. Instead, use `chainer.using_config('use_cudnn', use_cudnn)`. See `chainer.using_config()`.

**Parameters**

- `n_layers (int)` – Number of layers.
- `in_size (int)` – Dimensionality of input vectors.
- `out_size (int)` – Dimensionality of hidden states and output vectors.
- `dropout (float)` – Dropout ratio.

**See also:**
`chainer.functions.n_step_rnn()`

**Methods**

**__call__**(self, hx, xs)

Calculate all hidden states and cell states.

**Warning:** `train` argument is not supported anymore since v2. Instead, use `chainer.using_config('train', train)`. See `chainer.using_config()`.

**Parameters**

- `hx (Variable or None)` – Initial hidden states. If `None` is specified zero-vector is used.
- `xs (list of ~chainer.Variable)` – List of input sequences. Each element `xs[i]` is a `chainer.Variable` holding a sequence.

**__getitem__**(index)

Returns the child at given index.

**Parameters** `index (int)` – Index of the child in the list.

**Returns** The `index`-th child link.

**Return type** `Link`

**__len__**()

Returns the number of children.

**__iter__**()

**add_link**(link)

Registers a child link and adds it to the tail of the list.

**Parameters** `link (Link)` – The link object to be registered.
add_param (name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- **name (str)** – Name of the parameter. This name is also used as the attribute name.
- **shape (int or tuple of ints)** – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent (name, value)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name (str)** – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads (link)
append (link)
Registers a child link and adds it to the tail of the list.

This is equivalent to add_link(). This method has been added to emulate the list interface.

Parameters **link** (Link) – The link object to be registered.

children ()
cleargrads ()
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.
copy ()
copyparams (link)
disable_update ()
Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.
**enable_update()**

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

**init_hx(xs)**

**init_scope(*args, **kwds)**

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links (skipself=False)**

**namedlinks (skipself=False)**

**namedparams (include_uninit=True)**

**params (include_uninit=True)**

**register_persistent (name)**

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**

- **name** (`str`) – Name of the attribute to be registered.

**serialize (serializer)**

**to_cpu()**

**to_gpu (device=None)**

**zerograds()**

Initializes all gradient arrays by zero.

This method can be used for the same purpose of `cleargrads`, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads()` instead.

**Attributes**

**update_enabled**

`True` if at least one parameter has an update rule enabled.
within_init_scope

True if the current code is inside of an initialization scope.

See init_scope() for the details of the initialization scope.

xp

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.NStepRNNTanh

class chainer.links.NStepRNNTanh (self, n_layers, in_size, out_size, dropout)

Stacked Uni-directional RNN for sequences.

This link is stacked version of Uni-directional RNN for sequences. Note that the activation function is tanh. It calculates hidden and cell states of all layer at end-of-string, and all hidden states of the last layer for each time.

Unlike chainer.functions.n_step_rnn(), this function automatically sort inputs in descending order by length, and transpose the sequence. Users just need to call the link with a list of chainer.Variable holding sequences.

Parameters

• n_layers (int) – Number of layers.
• in_size (int) – Dimensionality of input vectors.
• out_size (int) – Dimensionality of hidden states and output vectors.
• dropout (float) – Dropout ratio.

See also:

chainer.functions.n_step_rnn()

Methods

__call__ (self, hx, xs)

Calculate all hidden states and cell states.

Warning: train argument is not supported anymore since v2. Instead, use chainer.
using_config('train', train). See chainer.using_config().

Parameters

• hx (Variable or None) – Initial hidden states. If None is specified zero-vector is used.
• xs (list of ~chainer.Variable) – List of input sequences. Each element xs[i] is a chainer.Variable holding a sequence.
__getitem__(index)
Returns the child at given index.

Parameters  
index (int) – Index of the child in the list.

Returns  
The index-th child link.

Return type  
Link

__len__()
Returns the number of children.

__iter__()
add_link(link)
Registers a child link and adds it to the tail of the list.

Parameters  
link (Link) – The link object to be registered.

add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```
can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

• name (str) – Name of the parameter. This name is also used as the attribute name.

• shape (int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• dtype – Data type of the parameter array.

• initializer – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent(name, value)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

• name (str) – Name of the persistent value. This name is also used for the attribute name.

• value – Value to be registered.

addgrads(link)
append(link)
Registers a child link and adds it to the tail of the list.

This is equivalent to add_link(). This method has been added to emulate the list interface.
Parameters

**link** *(Link)* – The link object to be registered.

**children**

**cleargrads**

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy**

**copyparams** *(link)*

**disable_update**

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update**

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

**init_hx** *(xs)*

**init_scope** *(*args, **kwds)*

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A **Parameter** object can be automatically registered by assigning it to an attribute under this context manager.

---

**Example**

In most cases, the parameter registration is done in the initializer method. Using the **init_scope** method, we can simply assign a **Parameter** object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** *(skipself=False)*

**namedlinks** *(skipself=False)*

**namedparams** *(include_uninit=True)*

**params** *(include_uninit=True)*

**register_persistent** *(name)*

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If **name** has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters** **name** *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*
to_cpu()

to_gpu(device=None)

derived()  
Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for
backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.

Attributes

update_enabled
True if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.

See init_scope() for the details of the initialization scope.

xp
Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.Scale

class chainer.links.Scale(axis=1, W_shape=None, bias_term=False, bias_shape=None)

Broadcasted elementwise product with learnable parameters.

Computes a elementwise product as scale() function does except that its second input is a learnable weight
parameter $W$ the link has.

Parameters

• axis (int) – The first axis of the first input of scale() function along which its second
input is applied.

• W_shape (tuple of ints) – Shape of learnable weight parameter. If None, this link
does not have learnable weight parameter so an explicit weight needs to be given to its
__call__ method’s second input.

• bias_term (bool) – Whether to also learn a bias (equivalent to Scale link + Bias link).

• bias_shape (tuple of ints) – Shape of learnable bias. If W_shape is None, this
should be given to determine the shape. Otherwise, the bias has the same shape W_shape
with the weight parameter and bias_shape is ignored.

See also:
See scale() for details.

Variables

• W (Parameter) – Weight parameter if W_shape is given. Otherwise, no W attribute.

• bias (Bias) – Bias term if bias_term is True. Otherwise, no bias attribute.
Methods

__call__(xs)
Applies broadcasted elementwise product.

Parameters
xs (list of Variables) – Input variables whose length should be one if the
link has a learnable weight parameter, otherwise should be two.

__getitem__(name)
Equivalent to getattr.

add_link(name, link)
Registers a child link to this chain.

Deprecated since version v2.0.0: Assign the child link directly to an attribute within an
initialization scope, instead. For example, the following code

```
chain.add_link('ll', L.Linear(3, 5))
```

can be replaced by the following line.

```
with self.init_scope():
    chain.ll = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

• name (str) – Name of the child link. This name is also used as the attribute name.

• link (Link) – The link object to be registered.

add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an
initialization scope instead. For example, the following code

```
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

• name (str) – Name of the parameter. This name is also used as the attribute name.

• shape (int or tuple of ints) – Shape of the parameter array. If it is omitted,
  the parameter variable is left uninitialized.

• dtype – Data type of the parameter array.

• initializer – If it is not None, the data is initialized with the given initializer. If
  it is an array, the data is directly initialized by it. If it is callable, it is used as a weight
  initializer. Note that in these cases, dtype argument is ignored.

add_persistent(name, value)
Registers a persistent value to the link.
The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name (str)** – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads (link)**

**children ()**

**cleargrads ()**

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy ()**

**copyparams (link)**

**disable_update ()**

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update ()**

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to True.

**init_scope (*args, **kwds)**

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links (skipself=False)**

**namedlinks (skipself=False)**

**namedparams (include_uninit=True)**

**params (include_uninit=True)**

**register_persistent (name)**

Registers an attribute of a given name as a persistent value.
This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**

- `name (str)` – Name of the attribute to be registered.
- `serialize (serializer)`
- `to_cpu()`
- `to_gpu (device=None)`
- `zerograds ()`

Initializes all gradient arrays by zero.

This method can be used for the same purpose of `cleargrads`, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads ()` instead.

**Attributes**

- `update_enabled` – True if at least one parameter has an update rule enabled.
- `within_init_scope` – True if the current code is inside of an initialization scope.

See `init_scope ()` for the details of the initialization scope.

- `xp` – Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

---

**chainer.links.StatefulGRU**

**class chainer.links.StatefulGRU (in_size, out_size, init=None, inner_init=None, bias_init=0)**

Stateful Gated Recurrent Unit function (GRU).

Stateful GRU function has six parameters $W_r, W_z, W, U_r, U_z,$ and $U$. All these parameters are $n \times n$ matrices, where $n$ is the dimension of hidden vectors.

Given input vector $x$, Stateful GRU returns the next hidden vector $h'$ defined as

$$
r = \sigma(W_r x + U_r h),
$$
$$
z = \sigma(W_z x + U_z h),
$$
$$
\bar{h} = \tanh(W x + U (r \odot h)),
$$
$$
h' = (1 - z) \odot h + z \odot \bar{h},
$$

where $h$ is current hidden vector.

As the name indicates, `StatefulGRU` is *stateful*, meaning that it also holds the next hidden vector $h'$ as a state. Use `GRU` as a stateless version of GRU.
Parameters

- **in_size (int)** – Dimension of input vector $x$.
- **out_size (int)** – Dimension of hidden vector $h$.
- **init** – Initializer for GRU’s input units ($W$). It is a callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value. If it is `None`, the default initializer is used.
- **inner_init** – Initializer for the GRU’s inner recurrent units ($U$). It is a callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value. If it is `None`, the default initializer is used.
- **bias_init** – Bias initializer. It is a callable that takes `numpy.ndarray` or `cupy.ndarray` and edits its value. If None, the bias is set to zero.

Variables **h** (**Variable**) – Hidden vector that indicates the state of `StatefulGRU`.

See also:

GRU

Methods

**__call__** ($x$)

**__getitem__** (**name**)  
Equivalent to getattr.

**add_link** (**name**, **link**)  
Registers a child link to this chain.

Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('ll', L.Linear(3, 5))
```

can be replaced by the following line.

```python
with self.init_scope():
    chain.ll = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name (str)** – Name of the child link. This name is also used as the attribute name.
- **link** (**Link**) – The link object to be registered.

**add_param** (**name**, **shape=** `None`, **dtype=** `<type 'numpy.float32'>`, **initializer=** `None`)  
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a `Parameter` object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```
The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** *(str)* – Name of the parameter. This name is also used as the attribute name.
- **shape** *(int or tuple of ints)* – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

**add_persistent** *(name, value)*

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name** *(str)* – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads** *(link)*

**children** *

**cleargrads** *

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy** *

**copyparams** *(link)*

**disable_update** *

Disables update rules of all parameters under the link hierarchy.

This method sets the `attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

**enable_update** *

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

**init_scope** *(*args, **kwds)*

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))

links (skipself=False)
namedlinks (skipself=False)
namedparams (include_uninit=True)
params (include_uninit=True)
register_persistent (name)
    Registers an attribute of a given name as a persistent value.
    This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters name (str) – Name of the attribute to be registered.
reset_state ()
serialize (serializer)
set_state (h)
to_cpu ()
to_gpu (device=None)
zerograd ()
    Initializes all gradient arrays by zero.
    This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.
    Deprecated since version v1.15: Use cleargrads() instead.

Attributes

update_enabled
    True if at least one parameter has an update rule enabled.
within_init_scope
    True if the current code is inside of an initialization scope.
    See init_scope() for the details of the initialization scope.
xp
    Array module for this link.
    Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.
chainer.links.StatefulPeepholeLSTM

class chainer.links.StatefulPeepholeLSTM(in_size, out_size)

Fully-connected LSTM layer with peephole connections.

This is a fully-connected LSTM layer with peephole connections as a chain. Unlike the LSTM link, this chain holds peep_i, peep_f and peep_o as child links besides upward and lateral.

Given a input vector $x$, Peephole returns the next hidden vector $h'$ defined as

$$a = \tanh(upwardx + lateralth),$$

$$i = \sigma(upwardx + lateralth + peepic),$$

$$f = \sigma(upwardx + lateralth + peepfc),$$

$$c' = a \odot i + f \odot c,$$

$$o = \sigma(upwardx + lateralth + peepoc'),$$

$$h' = o \tanh(c'),$$

where $\sigma$ is the sigmoid function, $\odot$ is the element-wise product, $c$ is the current cell state, $c'$ is the next cell state and $h$ is the current hidden vector.

Parameters

- **in_size** (*int*) – Dimension of the input vector $x$.
- **out_size** (*int*) – Dimension of the hidden vector $h$.

Variables

- **upward** (*Linear*) – Linear layer of upward connections.
- **lateral** (*Linear*) – Linear layer of lateral connections.
- **peep_i** (*Linear*) – Linear layer of peephole connections to the input gate.
- **peep_f** (*Linear*) – Linear layer of peephole connections to the forget gate.
- **peep_o** (*Linear*) – Linear layer of peephole connections to the output gate.
- **c** (*Variable*) – Cell states of LSTM units.
- **h** (*Variable*) – Output at the current time step.

Methods

__call__(x)

Updates the internal state and returns the LSTM outputs.

Parameters **x** (*Variable*) – A new batch from the input sequence.

Returns Outputs of updated LSTM units.

Return type **Variable**
__getitem__(name)
Equivalent to getattr.

add_link(name, link)
 Registers a child link to this chain.

Deprecation since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('ll', L.Linear(3, 5))
```

can be replaced by the following line.

```python
with self.init_scope():
    chain.ll = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters
- **name** *(str)* – Name of the child link. This name is also used as the attribute name.
- **link** *(Link)* – The link object to be registered.

add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
 Registers a parameter to the link.

Deprecation since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters
- **name** *(str)* – Name of the parameter. This name is also used as the attribute name.
- **shape** *(int or tuple of ints)* – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent(name, value)
 Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters
- **name** *(str)* – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.
addgrads (link)

children()

cleargrads()
    Clears all gradient arrays.
    This method should be called before the backward computation at every iteration of the optimization.

copy()

coppypars (link)

disable_update()
    Disables update rules of all parameters under the link hierarchy.
    This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

enable_update()
    Enables update rules of all parameters under the link hierarchy.
    This method sets the enabled flag of the update rule of each parameter variable to True.

init_rate (*args, **kwds)
    Creates an initialization scope.
    This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```
It sets None to the c and h attributes.

```
serialize(serializer)
to_cpu()
to_gpu(device=None)
zerograds()
```

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads()` instead.

**Attributes**

```
update_enabled
``` True if at least one parameter has an update rule enabled.

```
within_init_scope
``` True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

```
xp
``` Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

---

**chainer.links.StatelessLSTM**

```python
class chainer.links.StatelessLSTM(in_size, out_size=None, lateral_init=None, upward_init=None, bias_init=0, forget_bias_init=1)
```

Stateless LSTM layer.

This is a fully-connected LSTM layer as a chain. Unlike the `lstm()` function, this chain holds upward and lateral connections as child links. This link doesn’t keep cell and hidden states.

**Parameters**

```
in_size (int or None) – Dimension of input vectors. If None, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
```  

```
out_size (int) – Dimensionality of output vectors.
```  

**Variables**

```
upward (chainer.links.Linear) – Linear layer of upward connections.
```  

```
lateral (chainer.links.Linear) – Linear layer of lateral connections.
```  

**Example**

There are several ways to make a StatelessLSTM link.

Let a two-dimensional input array `x`, a cell state array `h`, and the output array of the previous step `h` be:
1. Give both in_size and out_size arguments:

```python
>>> l = L.StatelessLSTM(10, 20)
>>> c_new, h_new = l(c, h, x)
>>> c_new.shape
(1, 20)
>>> h_new.shape
(1, 20)
```

2. Omit in_size argument or fill it with None:

The below two cases are the same.

```python
>>> l = L.StatelessLSTM(20)
>>> c_new, h_new = l(c, h, x)
>>> c_new.shape
(1, 20)
>>> h_new.shape
(1, 20)
```

```python
>>> l = L.StatelessLSTM(None, 20)
>>> c_new, h_new = l(c, h, x)
>>> c_new.shape
(1, 20)
>>> h_new.shape
(1, 20)
```

---

**Methods**

**__call__**(c, h, x)

Returns new cell state and updated output of LSTM.

**Parameters**

- c (Variable) – Cell states of LSTM units.
- h (Variable) – Output at the previous time step.
- x (Variable) – A new batch from the input sequence.

**Returns**

Returns (c_new, h_new), where c_new represents new cell state, and h_new is updated output of LSTM units.

**Return type** tuple of ~chainer.Variable

**__getitem__**(name)

Equivalent to getattr.

add_link**(name, link)**

Registers a child link to this chain.
Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('l1', L.Linear(3, 5))
```

can be replaced by the following line.

```python
with self.init_scope():
    chain.l1 = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** (str) – Name of the child link. This name is also used as the attribute name.
- **link** (Link) – The link object to be registered.

**add_param** *(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)*

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** (str) – Name of the parameter. This name is also used as the attribute name.
- **shape** (int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

**add_persistent** *(name, value)*

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name** (str) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads** *(link)*

children()
cleargrads()
    Clears all gradient arrays.

    This method should be called before the backward computation at every iteration of the optimization.

copy()

copyparams(link)

disable_update()
    Disables update rules of all parameters under the link hierarchy.

    This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

enable_update()
    Enables update rules of all parameters under the link hierarchy.

    This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope(*args, **kwds)
    Creates an initialization scope.

    This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example
In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))

links (skipself=False)

namedlinks (skipself=False)

namedparams (include_uninit=True)

params (include_uninit=True)

register_persistent (name)
    Registers an attribute of a given name as a persistent value.

    This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

    Parameters name (str) – Name of the attribute to be registered.

serialize (serializer)

to_cpu()

to_gpu (device=None)
**zerograds()**

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads()` instead.

**Attributes**

**update_enabled**

True if at least one parameter has an update rule enabled.

**within_init_scope**

True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

**xp**

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

**Activation/loss/normalization functions with parameters**

<table>
<thead>
<tr>
<th>Class/Link</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.links.BatchNormalization</code></td>
<td>Batch normalization layer on outputs of linear or convolution functions.</td>
</tr>
<tr>
<td><code>chainer.links.LayerNormalization</code></td>
<td>Layer normalization layer on outputs of linear functions.</td>
</tr>
<tr>
<td><code>chainer.links.BinaryHierarchicalSoftmax</code></td>
<td>Hierarchical softmax layer over binary tree.</td>
</tr>
<tr>
<td><code>chainer.links.BlackOut</code></td>
<td>BlackOut loss layer.</td>
</tr>
<tr>
<td><code>chainer.links.CRF1d</code></td>
<td>Linear-chain conditional random field loss layer.</td>
</tr>
<tr>
<td><code>chainer.links.SimplifiedDropconnect</code></td>
<td>Fully-connected layer with simplified dropconnect regularization.</td>
</tr>
<tr>
<td><code>chainer.links.PReLU</code></td>
<td>Parametric ReLU function as a link.</td>
</tr>
<tr>
<td><code>chainer.links.Maxout</code></td>
<td>Fully-connected maxout layer.</td>
</tr>
<tr>
<td><code>chainer.links.NegativeSampling</code></td>
<td>Negative sampling loss layer.</td>
</tr>
</tbody>
</table>

**chainer.links.BatchNormalization**

```python
class chainer.links.BatchNormalization(size, decay=0.9, eps=2e-05, dtype=<type 'numpy.float32'>, use_gamma=True, use_beta=True, initial_gamma=None, initial_beta=None)
```

Batch normalization layer on outputs of linear or convolution functions.

This link wraps the `batch_normalization()` and `fixed_batch_normalization()` functions.

It runs in three modes: training mode, fine-tuning mode, and testing mode.

In training mode, it normalizes the input by batch statistics. It also maintains approximated population statistics by moving averages, which can be used for instant evaluation in testing mode.

In fine-tuning mode, it accumulates the input to compute population statistics. In order to correctly compute the population statistics, a user must use this mode to feed mini-batches running through whole training dataset.

In testing mode, it uses pre-computed population statistics to normalize the input variable. The population
statistics is approximated if it is computed by training mode, or accurate if it is correctly computed by fine-
tuning mode.

Parameters

- **size** (*int or tuple of ints*) – Size (or shape) of channel dimensions.
- **decay** (*float*) – Decay rate of moving average. It is used on training.
- **eps** (*float*) – Epsilon value for numerical stability.
- **dtype** (*numpy.dtype*) – Type to use in computing.
- **use_gamma** (*bool*) – If True, use scaling parameter. Otherwise, use unit(1) which makes
  no effect.
- **use_beta** (*bool*) – If True, use shifting parameter. Otherwise, use unit(0) which makes
  no effect.

See: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

See also:

*batch_normalization()*, *fixed_batch_normalization()*

Variables

- **gamma** (*Variable*) – Scaling parameter.
- **beta** (*Variable*) – Shifting parameter.
- **avg_mean** (*numpy.ndarray or cupy.ndarray*) – Population mean.
- **avg_var** (*numpy.ndarray or cupy.ndarray*) – Population variance.
- **N** (*int*) – Count of batches given for fine-tuning.
- **decay** (*float*) – Decay rate of moving average. It is used on training.
- **eps** (*float*) – Epsilon value for numerical stability. This value is added to the batch
  variances.

Methods

**__call__**(self, x, finetune=False)

Invokes the forward propagation of BatchNormalization.

In training mode, the BatchNormalization computes moving averages of mean and variance for evaluatiing
during training, and normalizes the input using batch statistics.

**Warning:** test argument is not supported anymore since v2. Instead, use chainer.
using_config('train', train). See chainer.using_config().

Parameters

- **x** (*Variable*) – Input variable.
- **finetune** (*bool*) – If it is in the training mode and finetune is True, BatchNormal-
  ization runs in fine-tuning mode; it accumulates the input array to compute population
  statistics for normalization, and normalizes the input using batch statistics.
**add_param**(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name**(str) – Name of the parameter. This name is also used as the attribute name.
- **shape**(int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

**add_persistent**(name, value)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name**(str) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads**(link)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

- **link**(Link) – Source link object.

**children**()

Returns a generator of all child links.

**Returns**

A generator object that generates all child links.

**cleargrads**()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy**()

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.
The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns**  
Copied link object.

**Return type**  
`Link`

**copyparams**  
Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**  
link ([`Link`]) – Source link object.

**disable_update**  
Disables update rules of all parameters under the link hierarchy.

This method sets the `attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

**enable_update**  
Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

**init_scope**  
(*args, **kwds*)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

### Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links**  
(*skipself=**False***)

Returns a generator of all links under the hierarchy.

**Parameters**  
*skipself* ([`bool`]) – If `True`, then the generator skips this link and starts with the first child link.

**Returns**  
A generator object that generates all links.

**namedlinks**  
(*skipself=**False***)

Returns a generator of all (path, link) pairs under the hierarchy.

**Parameters**  
*skipself* ([`bool`]) – If `True`, then the generator skips this link and starts with the first child link.

**Returns**  
A generator object that generates all (path, link) pairs.
namedparams
(include_uninit=True)

Returns a generator of all (path, param) pairs under the hierarchy.

Parameters
include_uninit (bool) – If True, it also generates uninitialized parameters.

Returns
A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params
(include_uninit=True)

Returns a generator of all parameters under the link hierarchy.

Parameters
include_uninit (bool) – If True, it also generates uninitialized parameters.

Returns
A generator object that generates all parameters.

register_persistent
(name)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters
name (str) – Name of the attribute to be registered.

serialize
(serializer)

Serializes the link object.

Parameters
serializer (AbstractSerializer) – Serializer object.

start_finetuning
()

Resets the population count for collecting population statistics.

This method can be skipped if it is the first time to use the fine-tuning mode. Otherwise, this method should be called before starting the fine-tuning mode again.

to_cpu
()

Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

Returns: self

to_gpu
(device=None)

Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

Parameters
device – Target device specifier. If omitted, the current device is used.

Returns: self

zerograds
()

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.
Attributes

update_enabled
True if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.
See init_scope() for the details of the initialization scope.

xp
Array module for this link.
Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.LayerNormalization

class chainer.links.LayerNormalization(size=None, eps=1e-06, initial_gamma=None, initial_beta=None)
Layer normalization layer on outputs of linear functions.

This link implements a “layer normalization” layer which normalizes the input units by statistics that are computed along the second axis, scales and shifts them. Parameter initialization will be deferred until the first forward data pass at which time the size will be determined.

Parameters

• size (int) – Size of input units. If None, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
• eps (float) – Epsilon value for numerical stability of normalization.
• initial_gamma (Initializer) – Initializer for scaling vector. If None, then the vector is filled by 1. If a scalar, the vector is filled by it. If numpy.ndarray, the vector is set by it.
• initial_beta (Initializer) – Initializer for shifting vector. If None, then the vector is filled by 0. If a scalar, the vector is filled by it. If numpy.ndarray, the vector is set by it.

Variables

• gamma (Parameter) – Scaling parameter.
• beta (Parameter) – Shifting parameter.
• eps (float) – Epsilon value for numerical stability.

See: Layer Normalization

Methods

__call__(x)
Apply layer normalization to given input.

Parameters x (Variable) – Batch vectors. Shape of this value must be (batch_size, unit_size), e.g., the output of linear().

Returns Output of the layer normalization.

Return type Variable
add_param \((\text{name}, \text{shape}=\text{None}, \text{dtype}=<\text{type} \text{\{'numpy.float32\}'}, \text{initializer}=\text{None})\) Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

\[
\text{link}\.\text{add_param}\(\text{\'W\', \text{shape}=(5, 3)}\)
\]

can be replaced by the following assignment.

\[
\text{with self}.\text{init_scope}():
    \text{link}.\text{W} = \text{chainer}.\text{Parameter}(\text{None}, (5, 3))
\]

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** (str) – Name of the parameter. This name is also used as the attribute name.
- **shape** (int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent \((\text{name}, \text{value})\) Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name** (str) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads \((\text{link})\) Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

- **link** (Link) – Source link object.

children () Returns a generator of all child links.

**Returns** A generator object that generates all child links.

cleargrads () Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy () Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

3.5. Standard Link implementations
The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns**  Copied link object.

**Return type**  *Link*

### copyparams(*link*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**  

- **link** (*Link*) – Source link object.

### disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the `chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

### enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

### init_scope(*args, **kwds*)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

### links(*skipself=False*)

Returns a generator of all links under the hierarchy.

**Parameters**  

- **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

**Returns**  A generator object that generates all links.

### namedlinks(*skipself=False*)

Returns a generator of all (path, link) pairs under the hierarchy.

**Parameters**  

- **skipself** (*bool*) – If `True`, then the generator skips this link and starts with the first child link.

**Returns**  A generator object that generates all (path, link) pairs.
**namedparams** *(include_uninit=True)*

Returns a generator of all (path, param) pairs under the hierarchy.

- **Parameters**
  - **include_uninit** *(bool)* – If True, it also generates uninitialized parameters.

  - **Returns**
    - A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params** *(include_uninit=True)*

Returns a generator of all parameters under the link hierarchy.

- **Parameters**
  - **include_uninit** *(bool)* – If True, it also generates uninitialized parameters.

  - **Returns**
    - A generator object that generates all parameters.

**register_persistent** *(name)*

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

- **Parameters**
  - **name** *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*

Serializes the link object.

- **Parameters**
  - **serializer** *(AbstractSerializer)* – Serializer object.

**to_cpu** *

Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

- **Returns**
  - self

**to_gpu** *(device=None)*

Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

- **Parameters**
  - **device** *(str)* – Target device specifier. If omitted, the current device is used.

  - **Returns**
    - self

**zerograds** *

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.

**Attributes**

- **update_enabled** *(bool)*
  - True if at least one parameter has an update rule enabled.

- **within_init_scope** *(bool)*
  - True if the current code is inside of an initialization scope.

  See init_scope() for the details of the initialization scope.
xp
Array module for this link.
Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.BinaryHierarchicalSoftmax

class chainer.links.BinaryHierarchicalSoftmax(in_size, tree)
Hierarchical softmax layer over binary tree.

In natural language applications, vocabulary size is too large to use softmax loss. Instead, the hierarchical softmax uses product of sigmoid functions. It costs only $O(\log(n))$ time where $n$ is the vocabulary size in average.

At first a user need to prepare a binary tree whose each leaf is corresponding to a word in a vocabulary. When a word $x$ is given, exactly one path from the root of the tree to the leaf of the word exists. Let $\text{path}(x) = ((e_1, b_1), \ldots, (e_m, b_m))$ be the path of $x$, where $e_i$ is an index of $i$-th internal node, and $b_i \in \{-1, 1\}$ indicates direction to move at $i$-th internal node (-1 is left, and 1 is right). Then, the probability of $x$ is given as below:

$$P(x) = \prod_{(e_i, b_i) \in \text{path}(x)} P(b_i | e_i) = \prod_{(e_i, b_i) \in \text{path}(x)} \sigma(b_i x^\top w_{e_i}),$$

where $\sigma(\cdot)$ is a sigmoid function, and $w$ is a weight matrix.

This function costs $O(\log(n))$ time as an average length of paths is $O(\log(n))$, and $O(n)$ memory as the number of internal nodes equals $n - 1$.

Parameters

* in_size (int) – Dimension of input vectors.
* tree – A binary tree made with tuples like $(1, 2, 3)$.

Variables \(W\) (Variable) – Weight parameter matrix.

See: Hierarchical Probabilistic Neural Network Language Model [Morin+, AISTAT2005].

Methods

__call__(x, t)
Computes the loss value for given input and ground truth labels.

Parameters

* x (Variable) – Input to the classifier at each node.
* t (Variable) – Batch of ground truth labels.

Returns Loss value.

Return type Variable

add_param (name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code
link.add_param('W', shape=(5, 3))

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute's type.

### Parameters
- **name** *(str)* – Name of the parameter. This name is also used as the attribute name.
- **shape** *(int or tuple of ints)* – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

### add_persistent *(name, value)*
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

### Parameters
- **name** *(str)* – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

### addgrads *(link)*
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

### Parameters
- **link** *(Link)* – Source link object.

### children *
Returns a generator of all child links.

### cleargrads *
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

### copy *
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

### Returns
Copied link object.

### Return type
Link
**copyparams** *(link)*
Copies all parameters from given link.
This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

- **link** *(Link) – Source link object.*

**static create_huffman_tree** *(word_counts)*
Makes a Huffman tree from a dictionary containing word counts.
This method creates a binary Huffman tree, that is required for *BinaryHierarchicalSoftmax*. For example, `{0: 8, 1: 5, 2: 6, 3: 4}` is converted to `((3, 1), (2, 0))`.

**Parameters**

- **word_counts** *(dict of int key and int or float values)* – Dictionary representing counts of words.

**Returns**

Binary Huffman tree with tuples and keys of `word_counts`.

**disable_update** ()
Disables update rules of all parameters under the link hierarchy.
This method sets the `chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

**enable_update** ()
Enables update rules of all parameters under the link hierarchy.
This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

**init_scope** (*args, **kwds)*
Creates an initialization scope.
This method returns a context manager object that enables registration of parameters (and links for *Chain*) by an assignment. A *Parameter* object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a *Parameter* object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** *(skipself=False)*
Returns a generator of all links under the hierarchy.

**Parameters**

- **skipself** *(bool) – If True, then the generator skips this link and starts with the first child link.*

**Returns**

A generator object that generates all links.

**namedlinks** *(skipself=False)*
Returns a generator of all (path, link) pairs under the hierarchy.

**Parameters**

- **skipself** *(bool) – If True, then the generator skips this link and starts with the first child link.*
**Returns** A generator object that generates all (path, link) pairs.

**namedparams** *(include_uninit=True)*

Returns a generator of all (path, param) pairs under the hierarchy.

**Parameters** *include_uninit* *(bool)* – If True, it also generates uninitialized parameters.

**Returns** A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params** *(include_uninit=True)*

Returns a generator of all parameters under the link hierarchy.

**Parameters** *include_uninit* *(bool)* – If True, it also generates uninitialized parameters.

**Returns** A generator object that generates all parameters.

**register_persistent** *(name)*

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If *name* has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters** *name* *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*

Serializes the link object.

**Parameters** *serializer* *(AbstractSerializer)* – Serializer object.

**to_cpu** *

**to_gpu** *(device=None)*

**zerograds** *

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use *cleargrads()* instead.

**Attributes**

**update_enabled**

True if at least one parameter has an update rule enabled.

**within_init_scope**

True if the current code is inside of an initialization scope.

See *init_scope()* for the details of the initialization scope.

**xp**

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns *numpy* or *cupy*.

**chainer.links.BlackOut**

**class** *chainer.links.BlackOut*(in_size, counts, sample_size)*

BlackOut loss layer.
See also:
black_out() for more detail.

Parameters

- **in_size (int)** – Dimension of input vectors.
- **counts (int list)** – Number of each identifiers.
- **sample_size (int)** – Number of negative samples.

Variables

- **W (Parameter)** – Weight parameter matrix.

Methods

**__call__**(x, t)

Computes the loss value for given input and ground truth labels.

Parameters

- **x (Variable)** – Input of the weight matrix multiplication.
- **t (Variable)** – Batch of ground truth labels.

Returns

- **Loss value.**

Return type

- **Variable**

**add_param**(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- **name (str)** – Name of the parameter. This name is also used as the attribute name.
- **shape (int or tuple of ints)** – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

**add_persistent**(name, value)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters
• **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.

• **value** – Value to be registered.

**addgrads** (*link*)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

- **link** (*Link*) – Source link object.

**children**()

Returns a generator of all child links.

**Returns**

A generator object that generates all child links.

**cleargrads**()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy**()

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns**

Copied link object.

**Return type**

*Link*

**copyparams** (*link*)

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

- **link** (*Link*) – Source link object.

**disable_update**()

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update**()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

**init_scope**(*args, **kws*)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

**Example**
In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** *(skipself=False)*

Returns a generator of all links under the hierarchy.

**Parameters**

*skipself* *(bool)* – If `True`, then the generator skips this link and starts with the first child link.

**Returns** A generator object that generates all links.

**namedlinks** *(skipself=False)*

Returns a generator of all (path, link) pairs under the hierarchy.

**Parameters**

*skipself* *(bool)* – If `True`, then the generator skips this link and starts with the first child link.

**Returns** A generator object that generates all (path, link) pairs.

**namedparams** *(include_uninit=True)*

Returns a generator of all (path, param) pairs under the hierarchy.

**Parameters**

*include_uninit* *(bool)* – If `True`, it also generates uninitialized parameters.

**Returns** A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params** *(include_uninit=True)*

Returns a generator of all parameters under the link hierarchy.

**Parameters**

*include_uninit* *(bool)* – If `True`, it also generates uninitialized parameters.

**Returns** A generator object that generates all parameters.

**register_persistent** *(name)*

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**

*name* *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*

Serializes the link object.

**Parameters**

*serializer* *(AbstractSerializer)* – Serializer object.

**to_cpu** *

**to_gpu** *(device=None)*

**zero_grads** *

Initializes all gradient arrays by zero.

This method can be used for the same purpose of `cleargrads`, but less efficient. This method is left for backward compatibility.
Deprecation warning: Use `cleargrads()` instead.

**Attributes**

**update_enabled**
True if at least one parameter has an update rule enabled.

**within_init_scope**
True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

**xp**
Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

---

**chainer.links.CRF1d**

**class** `chainer.links.CRF1d(n_label)`

Linear-chain conditional random field loss layer.

This link wraps the `crf1d()` function. It holds a transition cost matrix as a parameter.

**Parameters**

- **n_label** (`int`) – Number of labels.

See also:

- `crf1d()` for more detail.

**Variables**

- **cost** (`Variable`) – Transition cost parameter.

**Methods**

**__call__(xs, ys, reduce='mean')**

**add_param**(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)

Registers a parameter to the link.

Deprecation warning: Assign a `Parameter` object directly to an attribute within an initialization scope instead. For example, the following code can be replaced by the following assignment.

```python
link.add_param('W', shape=(5, 3))

with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** (`str`) – Name of the parameter. This name is also used as the attribute name.

- **shape** (`int or tuple of ints`) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

- **dtype** – Data type of the parameter array.
• **initializer** – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

**add_persistent**(name, value)
Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**
- `name` *(str)* – Name of the persistent value. This name is also used for the attribute name.
- `value` – Value to be registered.

**addgrads**(link)
Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**
- `link` *(Link)* – Source link object.

**argmax**(xs)
Computes a state that maximizes a joint probability.

**Parameters**
- `xs` *(list of Variable)* – Input vector for each label.

**Returns**
- A tuple of Variable representing each log-likelihood and a list representing the argmax path.

**Return type**
- tuple

See also:
See `crf1d_argmax()` for more detail.

**children**
Returns a generator of all child links.

**Returns**
A generator object that generates all child links.

**cleargrads**
Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy**
Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns**
Copied link object.

**Return type**
- `Link`
**copyparams** *(link)*

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**
- **link** *(Link)* – Source link object.

**disable_update** *

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update** *

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

**init_scope** *(*args, **kwds)*

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the **init_scope** method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** *(skipself=False)*

Returns a generator of all links under the hierarchy.

**Parameters**
- **skipself** *(bool)* – If True, then the generator skips this link and starts with the first child link.

**Returns**
A generator object that generates all links.

**namedlinks** *(skipself=False)*

Returns a generator of all (path, link) pairs under the hierarchy.

**Parameters**
- **skipself** *(bool)* – If True, then the generator skips this link and starts with the first child link.

**Returns**
A generator object that generates all (path, link) pairs.

**namedparams** *(include_uninit=True)*

Returns a generator of all (path, param) pairs under the hierarchy.

**Parameters**
- **include_uninit** *(bool)* – If True, it also generates uninitialized parameters.

**Returns**
A generator object that generates all (path, parameter) pairs. The paths are relative from this link.
**params** *(include_uninit=True)*

Returns a generator of all parameters under the link hierarchy.

**Parameters** *include_uninit* *(bool)* – If True, it also generates uninitialized parameters.

**Returns** A generator object that generates all parameters.

**register_persistent** *(name)*

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters** *name* *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*

Serializes the link object.

**Parameters** *serializer* *(AbstractSerializer)* – Serializer object.

**to_cpu** *(*)

Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

**Returns**: self

**to_gpu** *(device=None)*

Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

**Parameters** *device* – Target device specifier. If omitted, the current device is used.

**Returns**: self

**zerograds** *(*)

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.

**Attributes**

**update_enabled**

True if at least one parameter has an update rule enabled.

**within_init_scope**

True if the current code is inside of an initialization scope.

See init_scope() for the details of the initialization scope.

**xp**

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.
chainer.links.SimplifiedDropconnect

class chainer.links.SimplifiedDropconnect(in_size, out_size, ratio=0.5, nobias=False, initialW=None, initial_bias=None)

Fully-connected layer with simplified dropconnect regularization.

Notice: This implementation cannot be used for reproduction of the paper. There is a difference between the current implementation and the original one. The original version uses sampling with gaussian distribution before passing activation function, whereas the current implementation averages before activation.

Parameters

- **in_size (int)** – Dimension of input vectors. If None, parameter initialization will be deferred until the first forward data pass at which time the size will be determined.
- **out_size (int)** – Dimension of output vectors.
- **nobias (bool)** – If True, then this link does not use the bias term.
- **initialW (3-D array or None)** – Initial weight value. If None, the default initializer is used. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.
- **initial_bias (2-D array, float or None)** – Initial bias value. If None, the bias is set to 0. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.

Variables

- **W (Variable)** – Weight parameter.
- **b (Variable)** – Bias parameter.

See also:

simplified_dropconnect()

See also:


Methods

__call__(x, train=True, mask=None)

Applies the simplified dropconnect layer.

Parameters

- **x (chainer.Variable or numpy.ndarray or cupy.ndarray)** – Batch of input vectors. Its first dimension n is assumed to be the minibatch dimension.
- **train (bool)** – If True, executes simplified dropconnect. Otherwise, simplified dropconnect link works as a linear unit.
- **mask (None or chainer.Variable or numpy.ndarray or cupy.ndarray)** – If None, randomized simplified dropconnect mask is generated. Otherwise, The mask must be (n, M, N) shaped array. Main purpose of this option is debugging. mask array will be used as a dropconnect mask.

Returns Output of the simplified dropconnect layer.

Return type Variable
add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name (str)** – Name of the parameter. This name is also used as the attribute name.
- **shape (int or tuple of ints)** – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent(name, value)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name (str)** – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(link)

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

- **link (Link)** – Source link object.

children()

Returns a generator of all child links.

**Returns** A generator object that generates all child links.

cleargrads()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

copy()

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.
The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Returns  Copied link object.

Return type  Link

copyparams (link)
Copies all parameters from given link.
This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

Parameters  link (Link) – Source link object.

disable_update()
Disables update rules of all parameters under the link hierarchy.
This method sets the `attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

enable_update()
Enables update rules of all parameters under the link hierarchy.
This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope (*args, **kwds)
Creates an initialization scope.
This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example
In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (skipself=False)
Returns a generator of all links under the hierarchy.

Parameters  skipself (bool) – If True, then the generator skips this link and starts with the first child link.

Returns  A generator object that generates all links.

namedlinks (skipself=False)
Returns a generator of all (path, link) pairs under the hierarchy.

Parameters  skipself (bool) – If True, then the generator skips this link and starts with the first child link.

Returns  A generator object that generates all (path, link) pairs.
**namedparams** (*include_uninit=True*)

Returns a generator of all (path, param) pairs under the hierarchy.

- **Parameters**
  - `include_uninit` (*bool*) – If True, it also generates uninitialized parameters.

- **Returns**
  - A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params** (*include_uninit=True*)

Returns a generator of all parameters under the link hierarchy.

- **Parameters**
  - `include_uninit` (*bool*) – If True, it also generates uninitialized parameters.

- **Returns**
  - A generator object that generates all parameters.

**register_persistent** (*name*)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

- **Parameters**
  - `name` (*str*) – Name of the attribute to be registered.

**serialize** (*serializer*)

Serializes the link object.

- **Parameters**
  - `serializer` (*AbstractSerializer*) – Serializer object.

**to_cpu** ()

Copies parameter variables and persistent values to CPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.

- **Returns**
  - self

**to_gpu** (*device=None*)

Copies parameter variables and persistent values to GPU.

This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

- **Parameters**
  - `device` – Target device specifier. If omitted, the current device is used.

- **Returns**
  - self

**zerograds** ()

Initializes all gradient arrays by zero.

This method can be used for the same purpose of `cleargrads()`, but less efficient. This method is left for backward compatibility.

- **Deprecated since version v1.15**: Use `cleargrads()` instead.

**Attributes**

- **update_enabled**
  - True if at least one parameter has an update rule enabled.

- **within_init_scope**
  - True if the current code is inside of an initialization scope.

  See `init_scope()` for the details of the initialization scope.
xp
Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

**chainer.links.PReLU**

```python
class chainer.links.PReLU(shape=(), init=0.25)
```

Parametric ReLU function as a link.

**Parameters**

- `shape (tuple of ints)` – Shape of the parameter array.
- `init (float)` – Initial parameter value.

See the paper for details: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.

**See also:**

```python
chainer.functions.prelu()
```

**Variables**

- `W (Parameter)` – Coefficient of parametric ReLU.

**Methods**

```python
__call__(x)
```

Applies the parametric ReLU activation function.

**Parameters**

- `x (Variable)` – Input variable.

**Returns**

Output of the parametric ReLU function.

**Return type**

`Variable`

```python
add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
```

Registers a parameter to the link.

**Deprecated since version v2.0.0:** Assign a `Parameter` object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- `name (str)` – Name of the parameter. This name is also used as the attribute name.
- `shape (int or tuple of ints)` – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- `dtype` – Data type of the parameter array.
• **initializer** – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

`add_persistent(name, value)`

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

• `name (str)` – Name of the persistent value. This name is also used for the attribute name.

• `value` – Value to be registered.

`addgrads(link)`

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

**Parameters**

• `link (Link)` – Source link object.

`children()`

Returns a generator of all child links.

**Returns**

A generator object that generates all child links.

`cleargrads()`

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

`copy()`

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

**Returns**

Copied link object.

**Return type** `Link`

`copyparams(link)`

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

**Parameters**

• `link (Link)` – Source link object.

`disable_update()`

Disables update rules of all parameters under the link hierarchy.

This method sets the `chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

`enable_update()`

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`. 
**init_scope** (*args, **kwds*)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

### Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** (skipself=False)

Returns a generator of all links under the hierarchy.

- **Parameters** skipself (bool) – If True, then the generator skips this link and starts with the first child link.

- **Returns** A generator object that generates all links.

**namedlinks** (skipself=False)

Returns a generator of all (path, link) pairs under the hierarchy.

- **Parameters** skipself (bool) – If True, then the generator skips this link and starts with the first child link.

- **Returns** A generator object that generates all (path, link) pairs.

**namedparams** (include_uninit=True)

Returns a generator of all (path, parameter) pairs under the hierarchy.

- **Parameters** include_uninit (bool) – If True, it also generates uninitialized parameters.

- **Returns** A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

**params** (include_uninit=True)

Returns a generator of all parameters under the link hierarchy.

- **Parameters** include_uninit (bool) – If True, it also generates uninitialized parameters.

- **Returns** A generator object that generates all parameters.

**register_persistent** (name)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

- **Parameters** name (str) – Name of the attribute to be registered.

**serialize** (serializer)

Serializes the link object.
Parameters `serializer` (AbstractSerializer) – Serializer object.

to_cpu()
Copies parameter variables and persistent values to CPU.
This method does not handle non-registered attributes. If some of such attributes must be copied to CPU, the link implementation must override this method to do so.
Returns: self
to_gpu(device=None)
Copies parameter variables and persistent values to GPU.
This method does not handle non-registered attributes. If some of such attributes must be copied to GPU, the link implementation must override this method to do so.

Parameters `device` – Target device specifier. If omitted, the current device is used.
Returns: self
zerograds()
Initializes all gradient arrays by zero.
This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.
Deprecated since version v1.15: Use `cleargrads()` instead.

Attributes

`update_enabled`
True if at least one parameter has an update rule enabled.

`within_init_scope`
True if the current code is inside of an initialization scope.
See `init_scope()` for the details of the initialization scope.

`xp`
Array module for this link.
Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

`chainer.links.Maxout`

class chainer.links.Maxout (in_size, out_size, pool_size, initialW=None, initial_bias=0)
Fully-connected maxout layer.

Let $M$, $P$ and $N$ be an input dimension, a pool size, and an output dimension, respectively. For an input vector $x$ of size $M$, it computes

$$Y_i = \max_j (W_{ij} x + b_{ij}).$$

Here $W$ is a weight tensor of shape $(M, P, N)$, $b$ an optional bias vector of shape $(M, P)$ and $W_{ij}$ is a sub-vector extracted from $W$ by fixing first and second dimensions to $i$ and $j$, respectively. Minibatch dimension is omitted in the above equation.

As for the actual implementation, this chain has a Linear link with a $(M \times P, N)$ weight matrix and an optional $M \times P$ dimensional bias vector.

Parameters
• **in_size** (int) – Dimension of input vectors.
• **out_size** (int) – Dimension of output vectors.
• **pool_size** (int) – Number of channels.

**Variables**

- **initialW** (3-D array or None) – Initial weight value. If None, the default initializer is used to initialize the weight matrix.
- **initial_bias** (2-D array, float or None) – Initial bias value. If it is float, initial bias is filled with this value. If None, bias is omitted.

**Variables**

- **linear** (Link) – The Linear link that performs affine transformation.

**See also:**

- **maxout()**

**See also:**


**Methods**

- **__call__(x)**
  
  Applies the maxout layer.

  **Parameters**
  
  - **x** (Variable) – Batch of input vectors.

  **Returns**
  
  Output of the maxout layer.

  **Return type** Variable

- **__getitem__(name)**

  Equivalent to getattr.

- **add_link(name, link)**

  Registers a child link to this chain.

  Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

  ```python
  chain.add_link('l1', L.Linear(3, 5))
  ```

  can be replaced by the following line.

  ```python
  with self.init_scope():
      chain.l1 = L.Linear(3, 5)
  ```

  The latter one is easier for IDEs to keep track of the attribute’s type.

  **Parameters**

  - **name** (str) – Name of the child link. This name is also used as the attribute name.

  - **link** (Link) – The link object to be registered.

- **add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)**

  Registers a parameter to the link.

  Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code
link.add_param('W', shape=(5, 3))

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute's type.

**Parameters**

- `name (str)` – Name of the parameter. This name is also used as the attribute name.
- `shape (int or tuple of ints)` – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- `dtype` – Data type of the parameter array.
- `initializer` – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

**add_persistent (name, value)**

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- `name (str)` – Name of the persistent value. This name is also used for the attribute name.
- `value` – Value to be registered.

**addgrads (link)**

**children ()**

**cleargrads ()**

Cleans all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy ()**

**copyparams (link)**

**disable_update ()**

Disables update rules of all parameters under the link hierarchy.

This method sets the `attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

**enable_update ()**

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

**init_scope (**`args`, **`kwds`)**

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.
Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

```python
links (skipself=False)
namedlinks (skipself=False)
namedparams (include_uninit=True)
params (include_uninit=True)
register_persistent (name)
    Registers an attribute of a given name as a persistent value.

    This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

    Parameters
    `name` (str) -- Name of the attribute to be registered.

serialize (serializer)
to_cpu ()
to_gpu (device=None)
zerograds ()
    Initializes all gradient arrays by zero.

    This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

    Deprecated since version v1.15: Use `cleargrads()` instead.

Attributes

update_enabled
    True if at least one parameter has an update rule enabled.

within_init_scope
    True if the current code is inside of an initialization scope.

    See `init_scope()` for the details of the initialization scope.

xp
    Array module for this link.

    Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

3.5. Standard Link implementations 343
chainer.links.NegativeSampling

class chainer.links.NegativeSampling(in_size, counts, sample_size, power=0.75)

Negative sampling loss layer.

This link wraps the negative_sampling() function. It holds the weight matrix as a parameter. It also builds a sampler internally given a list of word counts.

Parameters

- **in_size** (int) – Dimension of input vectors.
- **counts** (int list) – Number of each identifiers.
- **sample_size** (int) – Number of negative samples.
- **power** (float) – Power factor $\alpha$.

See also:

negative_sampling() for more detail.

Variables

- **W** (Variable) – Weight parameter matrix.

Methods

__call__(x, t, reduce='sum')

Computes the loss value for given input and ground truth labels.

Parameters

- **x** (Variable) – Input of the weight matrix multiplication.
- **t** (Variable) – Batch of ground truth labels.
- **reduce** (str) – Reduction option. Its value must be either 'sum' or 'no'. Otherwise, ValueError is raised.

Returns

Loss value.

Return type

Variable

add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- **name** (str) – Name of the parameter. This name is also used as the attribute name.
- **shape** (int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
• `dtype` – Data type of the parameter array.

• `initializer` – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

`add_persistent(name, value)`

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

• `name (str)` – Name of the persistent value. This name is also used for the attribute name.

• `value` – Value to be registered.

`addgrads(link)`

Accumulates gradient values from given link.

This method adds each gradient array of the given link to corresponding gradient array of this link. The accumulation is even done across host and different devices.

Parameters `link (Link)` – Source link object.

`children()`

Returns a generator of all child links.

Returns A generator object that generates all child links.

`cleargrads()`

Cleans all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

`copy()`

Copies the link hierarchy to new one.

The whole hierarchy rooted by this link is copied. The copy is basically shallow, except that the parameter variables are also shallowly copied. It means that the parameter variables of copied one are different from ones of original link, while they share the data and gradient arrays.

The name of the link is reset on the copy, since the copied instance does not belong to the original parent chain (even if exists).

Returns Copied link object.

Return type `Link`

`copyparams(link)`

Copies all parameters from given link.

This method copies data arrays of all parameters in the hierarchy. The copy is even done across the host and devices. Note that this method does not copy the gradient arrays.

Parameters `link (Link)` – Source link object.

`disable_update()`

Disables update rules of all parameters under the link hierarchy.

This method sets the `attr:~chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.
enable_update()
   Enables update rules of all parameters under the link hierarchy.

   This method sets the enabled flag of the update rule of each parameter variable to True.

init_scope(*args, **kwds)
   Creates an initialization scope.

   This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example
In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links(skipself=False)
   Returns a generator of all links under the hierarchy.

   Parameters skipself (bool) – If True, then the generator skips this link and starts with the first child link.

   Returns A generator object that generates all links.

namedlinks(skipself=False)
   Returns a generator of all (path, link) pairs under the hierarchy.

   Parameters skipself (bool) – If True, then the generator skips this link and starts with the first child link.

   Returns A generator object that generates all (path, link) pairs.

namedparams(include_uninit=True)
   Returns a generator of all (path, param) pairs under the hierarchy.

   Parameters include_uninit (bool) – If True, it also generates uninitialized parameters.

   Returns A generator object that generates all (path, parameter) pairs. The paths are relative from this link.

params(include_uninit=True)
   Returns a generator of all parameters under the link hierarchy.

   Parameters include_uninit (bool) – If True, it also generates uninitialized parameters.

   Returns A generator object that generates all parameters.

register_persistent(name)
   Registers an attribute of a given name as a persistent value.

   This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.
Parameters

**name** (*str*) – Name of the attribute to be registered.

**serialize** (*serializer*)

Serializes the link object.

Parameters **serializer** (*AbstractSerializer*) – Serializer object.

**to_cpu** ()

**to_gpu** (*device=None*)

**zerograds** ()

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads()` instead.

Attributes

**update_enabled**

True if at least one parameter has an update rule enabled.

**within_init_scope**

True if the current code is inside of an initialization scope.

See `init_scope()` for the details of the initialization scope.

**xp**

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

**Machine learning models**

<table>
<thead>
<tr>
<th>chainer.links.Classifier</th>
<th>A simple classifier model.</th>
</tr>
</thead>
</table>

**chainer.links.Classifier**

class chainer.links.Classifier (*predictor*, *lossfun=<function softmax_cross_entropy>, *accfun=<function accuracy>*)

A simple classifier model.

This is an example of chain that wraps another chain. It computes the loss and accuracy based on a given input/label pair.

Parameters

- **predictor** (*Link*) – Predictor network.
- **lossfun** (*function*) – Loss function.
- **accfun** (*function*) – Function that computes accuracy.

Variables

- **predictor** (*Link*) – Predictor network.
- **lossfun** (*function*) – Loss function.
• **accfun** (*function*) – Function that computes accuracy.
• **y** (*Variable*) – Prediction for the last minibatch.
• **loss** (*Variable*) – Loss value for the last minibatch.
• **accuracy** (*Variable*) – Accuracy for the last minibatch.
• **compute_accuracy** (*bool*) – If True, compute accuracy on the forward computation.
  The default value is True.

**Methods**

**__call__** (*args*)
Computes the loss value for an input and label pair.
It also computes accuracy and stores it to the attribute.

- **Parameters**
  *args* (*list of ~chainer.Variable*) – Input minibatch.

  The all elements of *args* but last one are features and the last element corresponds to ground truth labels.
  It feeds features to the predictor and compare the result with ground truth labels.

- **Returns**
  Loss value.

- **Return type**
  *Variable*

**__getitem__** (*name*)
Equivalent to getattr.

**add_link** (*name, link*)
Registers a child link to this chain.

- Deprecated since version v2.0.0: Assign the child link directly to an attribute within *an initialization scope*, instead. For example, the following code

  ```python
  chain.add_link('l1', L.Linear(3, 5))
  ```

  can be replaced by the following line.

  ```python
  with self.init_scope():
      chain.l1 = L.Linear(3, 5)
  ```

  The latter one is easier for IDEs to keep track of the attribute’s type.

- **Parameters**
  * name* (*str*) – Name of the child link. This name is also used as the attribute name.
  * link* (*Link*) – The link object to be registered.

**add_param** (*name, shape=None, dtype=<type 'numpy.float32'>, initializer=None*)
Registers a parameter to the link.

- Deprecated since version v2.0.0: Assign a *Parameter* object directly to an attribute within *an initialization scope* instead. For example, the following code

  ```python
  link.add_param('W', shape=(5, 3))
  ```

  can be replaced by the following assignment.

  ```python
  with self.init_scope():
      link.W = chainer.Parameter(None, (5, 3))
  ```
The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** *(str)* – Name of the parameter. This name is also used as the attribute name.
- **shape** *(int or tuple of ints)* – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not `None`, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, `dtype` argument is ignored.

**add_persistent** *(name, value)*

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name** *(str)* – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads** *(link)*

**children**

**cleargrads**

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**copy**

**copyparams** *(link)*

**disable_update**

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to `False`.

**enable_update**

Enables update rules of all parameters under the link hierarchy.

This method sets the `enabled` flag of the update rule of each parameter variable to `True`.

**init_scope** *(*args, **kwds)*

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))

links (skipself=False)
namedlinks (skipself=False)
namedparams (include_uninit=True)
params (include_uninit=True)
register_persistent (name)
    Registers an attribute of a given name as a persistent value.
    This is a convenient method to register an existing attribute as a persistent value. If name has been already
    registered as a parameter, this method removes it from the list of parameter names and re-registers it as a
    persistent value.

    Parameters name (str) – Name of the attribute to be registered.
serialize (serializer)
to_cpu ()
to_gpu (device=None)
zerograds ()
    Initializes all gradient arrays by zero.
    This method can be used for the same purpose of cleargrads, but less efficient. This method is left for
    backward compatibility.
    Deprecated since version v1.15: Use cleargrads() instead.

Attributes

compute_accuracy = True
update_enabled
    True if at least one parameter has an update rule enabled.
within_init_scope
    True if the current code is inside of an initialization scope.
    See init_scope() for the details of the initialization scope.
xp
    Array module for this link.
    Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

Pre-trained models

Pre-trained models are mainly used to achieve a good performance with a small dataset, or extract a semantic feature vector. Although CaffeFunction automatically loads a pre-trained model released as a caffemodel, the following
link models provide an interface for automatically converting caffemodels, and easily extracting semantic feature vectors.

For example, to extract the feature vectors with VGG16Layers, which is a common pre-trained model in the field of image recognition, users need to write the following few lines:

```python
from chainer.links import VGG16Layers
from PIL import Image

model = VGG16Layers()
img = Image.open("path/to/image.jpg")
feature = model.extract([img], layers=['fc7'])['fc7']
```

where fc7 denotes a layer before the last fully-connected layer. Unlike the usual links, these classes automatically load all the parameters from the pre-trained models during initialization.

### VGG16Layers

<table>
<thead>
<tr>
<th><strong>Method</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.links.VGG16Layers</td>
<td>A pre-trained CNN model with 16 layers provided by VGG team.</td>
</tr>
<tr>
<td>chainer.links.model.vision.vgg.prepare</td>
<td>Converts the given image to the numpy array for VGG models.</td>
</tr>
</tbody>
</table>

#### chainer.links.VGG16Layers

**class** chainer.links.VGG16Layers (pretrained_model='auto')

A pre-trained CNN model with 16 layers provided by VGG team.

During initialization, this chain model automatically downloads the pre-trained caffemodel, convert to another chainer model, stores it on your local directory, and initializes all the parameters with it. This model would be useful when you want to extract a semantic feature vector from a given image, or fine-tune the model on a different dataset. Note that this pre-trained model is released under Creative Commons Attribution License.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the constructor, please use convert_caffemodel_to_npz classmethod instead.

See: K. Simonyan and A. Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*

**Parameters**

- **pretrained_model** *(str)* - the destination of the pre-trained chainer model serialized as a .npz file. If this argument is specified as auto, it automatically downloads the caffemodel from the internet. Note that in this case the converted chainer model is stored on $CHAINER_DATASET_ROOT/pfnet/chainer/models directory, where $CHAINER_DATASET_ROOT is set as $HOME/.chainer/dataset unless you specify another value as a environment variable. The converted chainer model is automatically used from the second time. If the argument is specified as None, all the parameters are not initialized by the pre-trained model, but the default initializer used in the original paper, i.e., chainer.initializers.Normal(scale=0.01).

**Variables**

- **available_layers** *(list of str)* - The list of available layer names used by __call__ and extract methods.
Methods

__call__(self, x, layers=['prob'])
Computes all the feature maps specified by layers.

Parameters

- x (Variable) – Input variable.
- layers (list of str) – The list of layer names you want to extract.

Returns
A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type
Dictionary of ~chainer.Variable

__getitem__(name)
Equivalent to getattr.

add_link(name, link)
Registers a child link to this chain.

Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('ll', L.Linear(3, 5))
```

can be replaced by the following line.

```python
with self.init_scope():
    chain.ll = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- name (str) – Name of the child link. This name is also used as the attribute name.
- link (Link) – The link object to be registered.

add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope, instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters
• **name** (*str*) – Name of the parameter. This name is also used as the attribute name.

• **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• **dtype** – Data type of the parameter array.

• **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, **dtype** argument is ignored.

**add_persistent** (*name, value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

• **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.

• **value** – Value to be registered.

**addgrads** (*link*)

**children** ()

**cleargrads** ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**classmethod convert_caffemodel_to_npz** (*path_caffemodel, path_npz*)

Converts a pre-trained caffemodel to a chainer model.

**Parameters**

• **path_caffemodel** (*str*) – Path of the pre-trained caffemodel.

• **path_npz** (*str*) – Path of the converted chainer model.

**copy** ()

**copyparams** (*link*)

**disable_update** ()

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update** ()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

**extract** (*self, images, layers=['fc7'], size=(224, 224))

Extracts all the feature maps of given images.

The difference of directly executing **call** is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls **prepare** and **call** functions.
**Warning:** test and volatile arguments are not supported anymore since v2. Instead, use `chainer.using_config('train', train)` and `chainer.using_config('enable_backprop', not volatile)` respectively. See `chainer.using_config()`.

**Parameters**

- **images**: (*iterable of PIL.Image or numpy.ndarray*) — Input images.
- **layers**: (*list of str*) — The list of layer names you want to extract.
- **size**: (*pair of ints*) — The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is `None`, but the resolutions of all the images should be the same.

**Returns**

A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

**Return type**

Dictionary of `~chainer.Variable`

**init_scope** (*args, **kwd*)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** (*skipself=False*)

**namedlinks** (*skipself=False*)

**namedparams** (*include_uninit=True*)

**params** (*include_uninit=True*)

**predict** (*images, oversample=True*)

Computes all the probabilities of given images.

**Parameters**

- **images**: (*iterable of PIL.Image or numpy.ndarray*) — Input images.
- **oversample**: (*bool*) — If `True`, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

**Returns**

Output that contains the class probabilities of given images.

**Return type**

`Variable`
**register_persistent** *(name)*
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If *name* has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**
- **name** *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*

**to_cpu** *

**to_gpu** *(device=None)*

**zerograds** *
 Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use **cleargrads()** instead.

**Attributes**

- **available_layers**
- **functions**
- **update_enabled**
  - True if at least one parameter has an update rule enabled.
- **within_init_scope**
  - True if the current code is inside of an initialization scope.
  - See **init_scope()** for the details of the initialization scope.
- **xp**
  - Array module for this link.
  - Depending on which of CPU/GPU this link is on, this property returns **numpy** or **cupy**.

**chainer.links.model.vision.vgg.prepare**

chainer.links.model.vision.vgg.**prepare** *(image, size=(224, 224))*
Converts the given image to the numpy array for VGG models.

Note that you have to call this method before **__call__** because the pre-trained vgg model requires to resize the given image, covert the RGB to the BGR, subtract the mean, and permute the dimensions before calling.

**Parameters**
- **image** *(PIL.Image or numpy.ndarray)* – Input image. If an input is **numpy.ndarray**, its shape must be *(height, width), (height, width, channels), or (channels, height, width), and the order of the channels must be RGB.*
- **size** *(pair of ints)* – Size of converted images. If None, the given image is not resized.

**Returns**
- The converted output array.

**Return type** **numpy.ndarray**
GoogLeNet

<table>
<thead>
<tr>
<th>chainer.links.GoogLeNet</th>
<th>A pre-trained GoogLeNet model provided by BVLC.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.links.model.vision.googlenet.prepare</td>
<td>Converts the given image to the numpy array for GoogLeNet.</td>
</tr>
</tbody>
</table>

chainer.links.GoogLeNet

class chainer.links.GoogLeNet (pretrained_model='auto')
A pre-trained GoogLeNet model provided by BVLC.

When you specify the path of the pre-trained chainer model serialized as a .npz file in the constructor, this chain model automatically initializes all the parameters with it. This model would be useful when you want to extract a semantic feature vector per image, or fine-tune the model on a different dataset.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the constructor, please use convert_caffemodel_to_npz classmethod instead.

GoogLeNet, which is also called Inception-v1, is an architecture of convolutional neural network proposed in 2014. This model is relatively lightweight and requires small memory footprint during training compared with modern architectures such as ResNet. Therefore, if you fine-tune your network based on a model pre-trained by Imagenet and need to train it with large batch size, GoogLeNet may be useful. On the other hand, if you just want an off-the-shelf classifier, we recommend you to use ResNet50 or other models since they are more accurate than GoogLeNet.

The original model is provided here: https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet

Parameters
pretrained_model (str) – the destination of the pre-trained chainer model serialized as a .npz file. If this argument is specified as auto, it automatically downloads the caffemodel from the internet. Note that in this case the converted chainer model is stored on $CHAINER_DATASET_ROOT/pfnet/chainer/models directory, where $CHAINER_DATASET_ROOT is set as $HOME/.chainer/dataset unless you specify another value as a environment variable. The converted chainer model is automatically used from the second time. If the argument is specified as None, all the parameters are not initialized by the pre-trained model, but the default initializer used in BVLC, i.e., chainer. initializers.LeCunUniform(scale=1.0). Note that, in Caffe, when weight_filler is specified as “xavier” type without variance_norm parameter, the weights are initialized by Uniform(-s, s), where $s = \sqrt{\frac{3}{fan_{in}}}$ and fan_{in} is the number of input units. This corresponds to LeCunUniform in Chainer but not GlorotUniform.

Variables
available_layers (list of str) – The list of available layer names used by __call__ and extract methods.

Methods

__call__ (self, x, layers=['prob'])
Computes all the feature maps specified by layers.

Warning: train argument is not supported anymore since v2. Instead, use chainer. using_config('train', train). See chainer.using_config().

Parameters
• \texttt{x} (Variable) – Input variable. It should be prepared by \texttt{prepare} function.

• \texttt{layers} (list of str) – The list of layer names you want to extract.

\textbf{Returns} A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

\textbf{Return type} Dictionary of \texttt{chainer.Variable}

\begin{verbatim}
__getitem__(name)
Equivalent to getattr.
\end{verbatim}

\begin{verbatim}
add_link(name, link)
Registers a child link to this chain.

Depreciated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

\begin{verbatim}
chain.add_link('l1', L.Linear(3, 5))
\end{verbatim}

can be replaced by the following line.

\begin{verbatim}
with self.init_scope():
  chain.l1 = L.Linear(3, 5)
\end{verbatim}

The latter one is easier for IDEs to keep track of the attribute’s type.

\textbf{Parameters}

• \texttt{name} (str) – Name of the child link. This name is also used as the attribute name.

• \texttt{link} (Link) – The link object to be registered.

\begin{verbatim}
add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)
Registers a parameter to the link.

Depreciated since version v2.0.0: Assign a \texttt{Parameter} object directly to an attribute within an initialization scope, instead. For example, the following code

\begin{verbatim}
link.add_param('W', shape=(5, 3))
\end{verbatim}

can be replaced by the following assignment.

\begin{verbatim}
with self.init_scope():
  link.W = chainer.Parameter(None, (5, 3))
\end{verbatim}

The latter one is easier for IDEs to keep track of the attribute’s type.

\textbf{Parameters}

• \texttt{name} (str) – Name of the parameter. This name is also used as the attribute name.

• \texttt{shape} (int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.

• \texttt{dtype} – Data type of the parameter array.

• \texttt{initializer} – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, \texttt{dtype} argument is ignored.
add_persistent(name, value)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

Parameters

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

addgrads(link)

children()

cleargrads()

Cleans all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

classmethod convert_caffemodel_to_npz(path_caffemodel, path_npz)

Converts a pre-trained caffemodel to a chainer model.

Parameters

- **path_caffemodel** (*str*) – Path of the pre-trained caffemodel.
- **path_npz** (*str*) – Path of the converted chainer model.

copy()

copyparams(link)

disable_update()

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

enable_update()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

extract(self, images, layers=[`pool5`], size=(224, 224))

Extracts all the feature maps of given images.

The difference of directly executing `__call__` is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls prepare and `__call__` functions.

> **Warning:** `train` and `volatile` arguments are not supported anymore since v2. Instead, use `chainer.using_config('train', train)` and `chainer.using_config('enable_backprop', not volatile)` respectively. See `chainer.using_config()`.

Parameters

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images.
- **layers** (*list of str*) – The list of layer names you want to extract.
• **size** (*pair of ints*) – The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is `None`, but the resolutions of all the images should be the same.

**Returns** A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

**Return type** Dictionary of `~chainer.Variable`

**init_scope** (*args, **kwds*)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for `Chain`) by an assignment. A `Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**predict** (*images, oversample=True*)

Computes all the probabilities of given images.

**Parameters**

- **images** (*iterable of PIL.Image or numpy.ndarray*) – Input images.
- **oversample** (*bool*) – If True, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

**Returns** Output that contains the class probabilities of given images.

**Return type** `Variable`

**register_persistent** (*name*)

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**

- **name** (*str*) – Name of the attribute to be registered.

**serialize** (*serializer*)

**to_cpu** ()
**to_gpu** (*device=None*)

**zerograds** ()

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads()` instead.

**Attributes**

- **available_layers**
- **functions**
- **update_enabled**
  - True if at least one parameter has an update rule enabled.
- **within_init_scope**
  - True if the current code is inside of an initialization scope.
  - See `init_scope()` for the details of the initialization scope.
- **xp**
  - Array module for this link.
  - Depending on which of CPU/GPU this link is on, this property returns `numpy` or `cupy`.

**chainer.links.model.vision.googlenet.prepare**

**chainer.links.model.vision.googlenet.prepare**(*image, size=(224, 224))

Converts the given image to the numpy array for GoogLeNet.

Note that you have to call this method before `__call__` because the pre-trained GoogLeNet model requires to resize the given image, covert the RGB to the BGR, subtract the mean, and permute the dimensions before calling.

**Parameters**

- **image** (*PIL.Image or numpy.ndarray*) – Input image. If an input is numpy.ndarray, its shape must be (height, width), (height, width, channels), or (channels, height, width), and the order of the channels must be RGB.

- **size** (*pair of ints*) – Size of converted images. If None, the given image is not resized.

**Returns**

The converted output array.

**Return type** `numpy.ndarray`

**Residual Networks**

**chainer.links.model.vision.resnet.ResNetLayers**

A pre-trained CNN model provided by MSRA.

**chainer.links.ResNet50Layers**

A pre-trained CNN model with 50 layers provided by MSRA.
Table 3.45 – continued from previous page

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.links.ResNet101Layers</td>
<td>A pre-trained CNN model with 101 layers provided by MSRA.</td>
</tr>
<tr>
<td>chainer.links.ResNet152Layers</td>
<td>A pre-trained CNN model with 152 layers provided by MSRA.</td>
</tr>
<tr>
<td>chainer.links.model.vision.resnet.prepare</td>
<td>Converts the given image to the numpy array for ResNets.</td>
</tr>
</tbody>
</table>

chainer.links.model.vision.resnet.ResNetLayers

class chainer.links.model.vision.resnet.ResNetLayers(pretrained_model, n_layers)

A pre-trained CNN model provided by MSRA.

When you specify the path of the pre-trained chainer model serialized as a .npz file in the constructor, this chain model automatically initializes all the parameters with it. This model would be useful when you want to extract a semantic feature vector per image, or fine-tune the model on a different dataset. Note that unlike VGG16Layers, it does not automatically download a pre-trained caffemodel. This caffemodel can be downloaded at GitHub.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the constructor, please use convert_caffemodel_to_npz classmethod instead.

See: K. He et. al., Deep Residual Learning for Image Recognition

Parameters

- pretrained_model (str) – the destination of the pre-trained chainer model serialized as a .npz file. If this argument is specified as auto, it automatically loads and converts the caffemodel from $CHAINER_DATASET_ROOT/pfnet/chainer/models/ResNet-{n-layers}-model.caffemodel, where $CHAINER_DATASET_ROOT is set as $HOME/.chainer/dataset unless you specify another value by modifying the environment variable and {n_layers} is replaced with the specified number of layers given as the first argument to this constructor. Note that in this case the converted chainer model is stored on the same directory and automatically used from the next time. If this argument is specified as None, all the parameters are not initialized by the pre-trained model, but the default initializer used in the original paper, i.e., chainer.initializers.HeNormal(scale=1.0).

- n_layers (int) – The number of layers of this model. It should be either 50, 101, or 152.

Variables

- available_layers (list of str) – The list of available layer names used by __call__ and extract methods.

Methods

__call__(self, x, layers=['prob'])

Computes all the feature maps specified by layers.

Warning: test argument is not supported anymore since v2. Instead, use chainer.using_config('train', train). See chainer.using_config().

Parameters

- x (Variable) – Input variable.
layers (list of str) – The list of layer names you want to extract.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

__getitem__ (name)
Equivalent to getattr.

add_link (name, link)
Registers a child link to this chain.

Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('l1', L.Linear(3, 5))
```

can be replaced by the following line.

```python
with self.init_scope():
    chain.l1 = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- name (str) – Name of the child link. This name is also used as the attribute name.
- link (Link) – The link object to be registered.

add_param (name, shape=None, dtype=<type ‘numpy.float32’>, initializer=None)
Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

- name (str) – Name of the parameter. This name is also used as the attribute name.
- shape (int or tuple of ints) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- dtype – Data type of the parameter array.
- initializer – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

add_persistent (name, value)
Registers a persistent value to the link.
The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name** *(str)* – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads** *(link)*

**children** ()

**cleargrads** ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**classmethod convertcaffemodeltonpz** *(pathcaffemodel, pathnpz, n_layers=50)*

Converts a pre-trained caffemodel to a chainer model.

**Parameters**

- **pathcaffemodel** *(str)* – Path of the pre-trained caffemodel.
- **pathnpz** *(str)* – Path of the converted chainer model.

**copy** ()

**copyparams** *(link)*

**disable_update** ()

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update** ()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

**extract** *(self, images, layers=['pool5'], size=(224, 224))*

Extracts all the feature maps of given images.

The difference of directly executing **__call__** is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls **prepare** and **__call__** functions.

**Warning:** test and volatile arguments are not supported anymore since v2. Instead, use `chainer.using_config('train', train)` and `chainer.using_config('enable_backprop', not volatile)` respectively. See `chainer.using_config()`.

**Parameters**

- **images** *(iterable of PIL.Image or numpy.ndarray)* – Input images.
- **layers** *(list of str)* – The list of layer names you want to extract.
- **size** *(pair of ints)* – The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is None, but the resolutions of all the images should be the same.
Returns  A directory in which the key contains the layer name and the value contains the cor-
responding feature map variable.

Return type  Dictionary of ~chainer.Variable

init_scope(*args, **kwds)
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example
In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

links (skipself=False)
namedlinks (skipself=False)
namedparams (include_uninit=True)
params (include_uninit=True)
predict (images, oversample=True)
Computes all the probabilities of given images.

Parameters

- **images** (iterable of PIL.Image or numpy.ndarray) – Input images.
- **oversample** (bool) – If True, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

Returns  Output that contains the class probabilities of given images.

Return type  Variable

register_persistent (name)
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters  name (str) – Name of the attribute to be registered.

serialize (serializer)
to_cpu()
to_gpu (device=None)
zerograd()  
Initializes all gradient arrays by zero.
This method can be used for the same purpose of cleargrads, but less efficient. This method is left for  
backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.

Attributes

available_layers

functions

update_enabled

True if at least one parameter has an update rule enabled.

within_init_scope

True if the current code is inside of an initialization scope.

See init_scope() for the details of the initialization scope.

xp

Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.ResNet50Layers

class chainer.links.ResNet50Layers(pretrained_model='auto')  
A pre-trained CNN model with 50 layers provided by MSRA.

When you specify the path of the pre-trained chainer model serialized as a .npz file in the constructor, this  
chain model automatically initializes all the parameters with it. This model would be useful when you want  
to extract a semantic feature vector per image, or fine-tune the model on a different dataset. Note that unlike  
VGG16Layers, it does not automatically download a pre-trained caffemodel. This caffemodel can be down- 
loaded at GitHub.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the  
constructor, please use convert_caffemodel_to_npz classmethod instead.

ResNet50 has 25,557,096 trainable parameters, and it’s 58% and 43% fewer than ResNet101 and ResNet152,  
respectively. On the other hand, the top-5 classification accuracy on ImageNet dataset drops only 0.7% and  
1.1% from ResNet101 and ResNet152, respectively. Therefore, ResNet50 may have the best balance between  
the accuracy and the model size. It would be basically just enough for many cases, but some advanced models  
for object detection or semantic segmentation use deeper ones as their building blocks, so these deeper ResNets  
are here for making reproduction work easier.

See: K. He et. al., Deep Residual Learning for Image Recognition

Parameters pretrained_model (str) – the destination of the pre-trained chainer model  
serialized as a .npz file. If this argument is specified as auto, it automatically loads and converts the caffemodel from  
$CHAINER_DATASET_ROOT/pfnet/chainer/models/ResNet-50-model.caffemodel, where $CHAINER_DATASET_ROOT is set  
as $HOME/.chainer/dataset unless you specify another value by modifying the environment variable. Note that in this case the converted chainer model is stored on the same directory and automatically used from the next time. If this argument is specified as None, all the parameters are not initialized by the pre-trained model, but the default initializer used in the original  
paper, i.e., chainer.initializers.HeNormal(scale=1.0).
Variables **available_layers** *(list of str)* – The list of available layer names used by __call__ and extract methods.

**Methods**

__call__(self, x, layers=['prob'])

Computes all the feature maps specified by layers.

**Parameters**

- **x** *(Variable)* – Input variable.
- **layers** *(list of str)* – The list of layer names you want to extract.

**Returns** A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

**Return type** Dictionary of ~chainer.Variable

__getitem__(name)

Equivalent to getattr.

add_link(name, link)

Registers a child link to this chain.

Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('l1', L.Linear(3, 5))
```

can be replaced by the following line.

```python
with self.init_scope():
    chain.l1 = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** *(str)* – Name of the child link. This name is also used as the attribute name.
- **link** *(Link)* – The link object to be registered.

add_param(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```
The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** (*str*) – Name of the parameter. This name is also used as the attribute name.
- **shape** (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not *None*, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, *dtype* argument is ignored.

**add_persistent** (*name, value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name** (*str*) – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads** (*link*)

**children** ()

**cleargrads** ()

Clears all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**convert_caffemodel_to_npz** (*path_caffemodel, path_npz*, *n_layers=50*)

Converts a pre-trained caffemodel to a chainer model.

**Parameters**

- **path_caffemodel** (*str*) – Path of the pre-trained caffemodel.
- **path_npz** (*str*) – Path of the converted chainer model.

**copy** ()

**copyparams** (*link*)

**disable_update** ()

Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to *False*.

**enable_update** ()

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to *True*.

**extract** (*self, images, layers=['pool5'], size=(224, 224))

Extracts all the feature maps of given images.

The difference of directly executing `__call__` is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls `prepare` and `__call__` functions.

3.5. Standard Link implementations
Warning: test and volatile arguments are not supported anymore since v2. Instead, use chainer.using_config('train', train) and chainer.using_config('enable_backprop', not volatile) respectively. See chainer.using_config().

Parameters

- **images** *(iterable of PIL.Image or numpy.ndarray)* – Input images.
- **layers** *(list of str)* – The list of layer names you want to extract.
- **size** *(pair of ints)* – The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is None, but the resolutions of all the images should be the same.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

*init_scope* (*args, **kwd*)

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

Example

In most cases, the parameter registration is done in the initializer method. Using the *init_scope* method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

*links* *(skipself=False)*

*namedlinks* *(skipself=False)*

*namedparams* *(include_uninit=True)*

*params* *(include_uninit=True)*

*predict* *(images, oversample=True)*

Computes all the probabilities of given images.

Parameters

- **images** *(iterable of PIL.Image or numpy.ndarray)* – Input images.
- **oversample** *(bool)* – If True, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

Returns Output that contains the class probabilities of given images.

Return type Variable
register_persistent (name)
    Registers an attribute of a given name as a persistent value.

    This is a convenient method to register an existing attribute as a persistent value. If name has been already
registered as a parameter, this method removes it from the list of parameter names and re-registers it as a
persistent value.

    Parameters
    name (str) – Name of the attribute to be registered.

serialize (serializer)

to_cpu()

to_gpu (device=None)

zerograds()
    Initializes all gradient arrays by zero.

    This method can be used for the same purpose of cleargrads, but less efficient. This method is left for
backward compatibility.

    Deprecated since version v1.15: Use cleargrads() instead.

Attributes

available_layers

functions

update_enabled
    True if at least one parameter has an update rule enabled.

within_init_scope
    True if the current code is inside of an initialization scope.

    See init_scope() for the details of the initialization scope.

xp
    Array module for this link.

    Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.ResNet101Layers

class chainer.links.ResNet101Layers (pretrained_model='auto')

    A pre-trained CNN model with 101 layers provided by MSRA.

    When you specify the path of the pre-trained chainer model serialized as a .npz file in the constructor, this
chain model automatically initializes all the parameters with it. This model would be useful when you want
to extract a semantic feature vector per image, or fine-tune the model on a different dataset. Note that unlike
VGG16Layers, it does not automatically download a pre-trained caffemodel. This caffemodel can be down-
loaded at GitHub.

    If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the
constructor, please use convert_caffemodel_to_npz classmethod instead.

    ResNet101 has 44,549,224 trainable parameters, and it’s 43% fewer than ResNet152 model, while the top-5
classification accuracy on ImageNet dataset drops 1.1% from ResNet152. For many cases, ResNet50 may have
the best balance between the accuracy and the model size.

    See: K. He et. al., Deep Residual Learning for Image Recognition
Parameters **pretrained_model** (*str*) – the destination of the pre-trained chainer model serialized as a .npz file. If this argument is specified as auto, it automatically loads and converts the caffemodel from $\text{SCHAINER\_DATASET\_ROOT}/\text{pfnet}/\text{chainer}/models/\text{ResNet-101-model.caffemodel}$, where $\text{SCHAINER\_DATASET\_ROOT}$ is set as $\text{HOME}/.chainer/dataset$ unless you specify another value by modifying the environment variable. Note that in this case the converted chainer model is stored on the same directory and automatically used from the next time. If this argument is specified as None, all the parameters are not initialized by the pre-trained model, but the default initializer used in the original paper, i.e., chainer.initializers.HeNormal(scale=1.0).

Variables **available_layers** (*list of str*) – The list of available layer names used by **__call__** and **extract** methods.

Methods

**__call__**(*self, x, layers=['prob'])**
Computes all the feature maps specified by layers.

---

**Warning**: test argument is not supported anymore since v2. Instead, use chainer.using_config('train', True). See chainer.using_config().

Parameters

• **x** (*Variable*) – Input variable.

• **layers** (*list of str*) – The list of layer names you want to extract.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

**__getitem__**(*name*)
Equivalent to getattr.

**add_link**(*name, link*)
Registers a child link to this chain.

Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('ll', L.Linear(3, 5))
```

can be replaced by the following line.

```python
with self.init_scope():
    chain.ll = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

Parameters

• **name** (*str*) – Name of the child link. This name is also used as the attribute name.

• **link** (*Link*) – The link object to be registered.


**add_param** *(name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)*

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name** *(str)* – Name of the parameter. This name is also used as the attribute name.
- **shape** *(int or tuple of ints)* – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

**add_persistent** *(name, value)*

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name** *(str)* – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads** *(link)*

**children()**

**cleargrads()**

Cleans all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

**convert_caffemodel_to_npz** *(path_caffemodel, path_npz, n_layers=50)*

Converts a pre-trained caffemodel to a chainer model.

**Parameters**

- **path_caffemodel** *(str)* – Path of the pre-trained caffemodel.
- **path_npz** *(str)* – Path of the converted chainer model.

**copy()**

**copyparams** *(link)*

**disable_update()**

Disables update rules of all parameters under the link hierarchy.
This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

**enable_update()**

Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

**extract(self, images, layers=['pool5'], size=(224, 224))**

Extracts all the feature maps of given images.

The difference of directly executing __call__ is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls prepare and __call__ functions.

**Warning:** test and volatile arguments are not supported anymore since v2. Instead, use chainer.using_config('train', train) and chainer.using_config('enable_backprop', not volatile) respectively. See chainer.using_config().

**Parameters**

- **images** *(iterable of PIL.Image or numpy.ndarray)* – Input images.
- **layers** *(list of str)* – The list of layer names you want to extract.
- **size** *(pair of ints)* – The resolution of resized images used as an input of CNN. All the given images are not resized if this argument is None, but the resolutions of all the images should be the same.

**Returns** A dictionary in which the key contains the layer name and the value contains the corresponding feature map variable.

**Return type** Dictionary of ~chainer.Variable

**init_scope(**args, **kwargs)**

Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.

**Example**

In most cases, the parameter registration is done in the initializer method. Using the init_scope method, we can simply assign a Parameter object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

**links** *(skipself=False)*

**namedlinks** *(skipself=False)*
**namedparams** *(include_uninit=True)*
**params** *(include_uninit=True)*

**predict** *(images, oversample=True)*
Computes all the probabilities of given images.

**Parameters**

- **images** *(iterable of PIL.Image or numpy.ndarray)* – Input images.
- **oversample** *(bool)* – If True, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

**Returns** Output that contains the class probabilities of given images.

**Return type** Variable

**register_persistent** *(name)*
Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If name has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters**
name *(str)* – Name of the attribute to be registered.

**serialize** *(serializer)*

**to_cpu** *

**to_gpu** *(device=None)*

**zerograds** *
Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.

**Attributes**

**available_layers**

**functions**

**update_enabled**
True if at least one parameter has an update rule enabled.

**within_init_scope**
True if the current code is inside of an initialization scope.

See init_scope() for the details of the initialization scope.

**xp**
Array module for this link.

Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.
chainer.links.ResNet152Layers

class chainer.links.ResNet152Layers(pretrained_model='auto')

A pre-trained CNN model with 152 layers provided by MSRA.

When you specify the path of the pre-trained chainer model serialized as a .npz file in the constructor, this chain model automatically initializes all the parameters with it. This model would be useful when you want to extract a semantic feature vector per image, or fine-tune the model on a different dataset. Note that unlike VGG16Layers, it does not automatically download a pre-trained caffemodel. This caffemodel can be downloaded at GitHub.

If you want to manually convert the pre-trained caffemodel to a chainer model that can be specified in the constructor, please use convert_caffemodel_to_npz classmethod instead.

ResNet152 has 60,192,872 trainable parameters, and it’s the deepest ResNet model and it achieves the best result on ImageNet classification task in ILSVRC 2015.

See: K. He et. al., Deep Residual Learning for Image Recognition

Parameters

**pretrained_model** *(str)* – the destination of the pre-trained chainer model serialized as a .npz file. If this argument is specified as auto, it automatically loads and converts the caffemodel from $CHAINER_DATASET_ROOT/pfnet/chainer/models/ResNet-152-model.caffemodel, where $CHAINER_DATASET_ROOT is set as $HOME/.chainer/dataset unless you specify another value by modifying the environment variable. Note that in this case the converted chainer model is stored on the same directory and automatically used from the next time. If this argument is specified as None, all the parameters are not initialized by the pre-trained model, but the default initializer used in the original paper, i.e., chainer.initializers.HeNormal(scale=1.0).

Variables

**available_layers** *(list of str)* – The list of available layer names used by __call__ and extract methods.

Methods

**__call__**(self, x, layers=['prob'])

Computes all the feature maps specified by layers.

Warning: test argument is not supported anymore since v2. Instead, use chainer.using_config('train', train). See chainer.using_config().

Parameters

- **x** *(Variable)* – Input variable.
- **layers** *(list of str)* – The list of layer names you want to extract.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

**__getitem__**(name)

Equivalent to getattr.

**add_link**(name, link)

Registers a child link to this chain.
Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('l1', L.Linear(3, 5))
```
can be replaced by the following line.

```python
with self.init_scope():
    chain.l1 = L.Linear(3, 5)
```
The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name (str)** – Name of the child link. This name is also used as the attribute name.
- **link (Link)** – The link object to be registered.

**add_param** (name, shape=None, dtype=<type 'numpy.float32'>, initializer=None)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```
can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```
The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- **name (str)** – Name of the parameter. This name is also used as the attribute name.
- **shape (int or tuple of ints)** – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- **dtype** – Data type of the parameter array.
- **initializer** – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

**add_persistent** (name, value)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- **name (str)** – Name of the persistent value. This name is also used for the attribute name.
- **value** – Value to be registered.

**addgrads** (link)

**children** ()
cleargrads()
Cleans all gradient arrays.

This method should be called before the backward computation at every iteration of the optimization.

convert_caffemodel_to_npz(path_caffemodel, path_npz, n_layers=50)
Converts a pre-trained caffemodel to a chainer model.

Parameters
- path_caffemodel (str) – Path of the pre-trained caffemodel.
- path_npz (str) – Path of the converted chainer model.

copy()
copyparams(link)
disable_update()
Disables update rules of all parameters under the link hierarchy.

This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.

enable_update()
Enables update rules of all parameters under the link hierarchy.

This method sets the enabled flag of the update rule of each parameter variable to True.

extract(self, images, layers=['pool5'], size=(224, 224))
Extracts all the feature maps of given images.

The difference of directly executing __call__ is that it directly accepts images as an input and automatically transforms them to a proper variable. That is, it is also interpreted as a shortcut method that implicitly calls prepare and __call__ functions.

Warning: test and volatile arguments are not supported anymore since v2. Instead, use chainer.using_config('train', train) and chainer.using_config('enable_backprop', not volatile) respectively. See chainer.using_config().

Parameters
- images (iterable of PIL.Image or numpy.ndarray) – Input images.
- layers (list of str) – The list of layer names you want to extract.
- size (pair of ints) – The resolution of resized images used as an input of CNN.

All the given images are not resized if this argument is None, but the resolutions of all the images should be the same.

Returns A directory in which the key contains the layer name and the value contains the corresponding feature map variable.

Return type Dictionary of ~chainer.Variable

init_scope(*args, **kwds)
Creates an initialization scope.

This method returns a context manager object that enables registration of parameters (and links for Chain) by an assignment. A Parameter object can be automatically registered by assigning it to an attribute under this context manager.
Example

In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a `Parameter` object to register it to the link.

```python
class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))
```

`links` *(skipself=False)*

`namedlinks` *(skipself=False)*

`namedparams` *(include_uninit=True)*

`params` *(include_uninit=True)*

`predict` *(images, oversample=True)*

Computes all the probabilities of given images.

**Parameters**

- **images** *(iterable of PIL.Image or numpy.ndarray)* — Input images.
- **oversample** *(bool)* — If True, it averages results across center, corners, and mirrors. Otherwise, it uses only the center.

**Returns** Output that contains the class probabilities of given images.

**Return type** `Variable`

`register_persistent` *(name)*

Registers an attribute of a given name as a persistent value.

This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

**Parameters** **name** *(str)* — Name of the attribute to be registered.

`serialize` *(serializer)*

`to_cpu` *

`to_gpu` *(device=None)*

`zerograds` *

Initializes all gradient arrays by zero.

This method can be used for the same purpose of `cleargrads`, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use `cleargrads()` instead.

**Attributes**

- **available_layers**
- **functions**

3.5. Standard Link implementations
update_enabled
True if at least one parameter has an update rule enabled.

within_init_scope
True if the current code is inside of an initialization scope.
See init_scope() for the details of the initialization scope.

xp
Array module for this link.
Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

chainer.links.model.vision.resnet.prepare
chainer.links.model.vision.resnet.prepare(image, size=(224, 224))
Converts the given image to the numpy array for ResNets.
Note that you have to call this method before __call__ because the pre-trained resnet model requires to resize
the given image, covert the RGB to the BGR, subtract the mean, and permute the dimensions before calling.

Parameters
• image (PIL.Image or numpy.ndarray) – Input image. If an input is numpy.
darray, its shape must be (height, width),(height, width, channels),
or (channels, height, width), and the order of the channels must be RGB.
• size (pair of ints) – Size of converted images. If None, the given image is not
resized.

Returns
The converted output array.

Return type
numpy.ndarray

Optimizers

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.optimizers.AdaDelta</td>
<td>Zeiler’s ADADELTA.</td>
</tr>
<tr>
<td>chainer.optimizers.AdamGrad</td>
<td>AdaGrad optimizer.</td>
</tr>
<tr>
<td>chainer.optimizers.Adam</td>
<td>Adam optimizer.</td>
</tr>
<tr>
<td>chainer.optimizers.MomentumSGD</td>
<td>Momentum SGD optimizer.</td>
</tr>
<tr>
<td>chainer.optimizers.NesterovAG</td>
<td>Nesterov’s Accelerated Gradient.</td>
</tr>
<tr>
<td>chainer.optimizers.RMSprop</td>
<td>RMSprop optimizer.</td>
</tr>
<tr>
<td>chainer.optimizers.RMSpropGraves</td>
<td>Alex Graves’s RMSprop.</td>
</tr>
<tr>
<td>chainer.optimizers.SGD</td>
<td>Vanilla Stochastic Gradient Descent.</td>
</tr>
<tr>
<td>chainer.optimizers.SMORMS3</td>
<td>Simon Funk’s SMORMS3.</td>
</tr>
</tbody>
</table>

chainer.optimizers.AdaDelta

class chainer.optimizers.AdaDelta(rho=0.95, eps=1e-06)
Zeiler’s ADADELTA.


Parameters
• rho (float) – Exponential decay rate of the first and second order moments.
• **eps** (*float*) – Small value for the numerical stability.

## Methods

### add_hook (hook, name=None)

Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method.

**Parameters**

- **hook** (*function*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.

### call_hooks ()

Invokes hook functions in registration order.

### create_update_rule ()

### new_epoch ()

Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

### reallocate_cleared_grads ()

Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

### remove_hook (name)

Removes a hook function.

**Parameters**

- **name** (*str*) – Registered name of the hook function to remove.

### serialize (serializer)

Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (`t` and `epoch`)

**It does not saves nor loads the parameters of the target link.** They should be separately saved or loaded.

**Parameters**

- **serializer** (*AbstractSerializer*) – Serializer or deserializer object.

### setup (link)

### update (lossfun=None, *args, **kwds)

Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- **If lossfun** is given, then it is used as a loss function to compute gradients.
- **Otherwise**, this method assumes that the gradients are already computed.
In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

```python
use_cleargrads (use=True)
```

Enables or disables use of `cleargrads()` in `update`.

**Parameters**

- `use` (bool) – If True, this function enables use of `cleargrads`. If False, disables use of `cleargrads` (`zerograd` is used).

Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograd()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

### Attributes

- `eps`  
  Alias to `self.hyperparam.eps`
- `rho`  
  Alias to `self.hyperparam.rho`

#### chainer.optimizers.AdaGrad

```python
class chainer.optimizers.AdaGrad (lr=0.001, eps=1e-08)
```

AdaGrad optimizer.

See: http://jmlr.org/papers/v12/duchi11a.html

**Parameters**

- `lr` (float) – Learning rate.
- `eps` (float) – Small value for the numerical stability.

### Methods

- `add_hook (hook, name=None)`  
  Registers a hook function.  
  Hook function is typically called right after the gradient computation, though the timing depends on the optimization method.

  **Parameters**

  - `hook` (function) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
  - `name` (str) – Name of the registration. If omitted, `hook.name` is used by default.

- `call_hooks ()`  
  Invokes hook functions in registration order.

- `create_update_rule ()`

- `new_epoch ()`  
  Starts a new epoch.
This method increments the epoch count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

**reallocate_cleared_grads()**
Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have None. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

**remove_hook**(name)
Removes a hook function.

- **Parameters**
  - name (str) – Registered name of the hook function to remove.

**serialize**(serializer)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (t and epoch)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

- **Parameters**
  - serializer (AbstractSerializer) – Serializer or deserializer object.

**setup**(link)

**update**(lossfun=None, *args, **kwds)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If lossfun is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

**use_cleargrads**(use=True)
Enables or disables use of `cleargrads()` in `update`.

- **Parameters**
  - use (bool) – If True, this function enables use of `cleargrads`. If False, disables use of `cleargrads` (zerograds is used).

Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

**Attributes**

- **eps**
  - Alias to `self.hyperparam.eps`

- **lr**
  - Alias to `self.hyperparam.lr`
chainer.optimizers.Adam

class chainer.optimizers.Adam(alpha=0.001, beta1=0.9, beta2=0.999, eps=1e-08)
Adam optimizer.
See: http://arxiv.org/abs/1412.6980v8

Parameters

• alpha (float) – Step size.
• beta1 (float) – Exponential decay rate of the first order moment.
• beta2 (float) – Exponential decay rate of the second order moment.
• eps (float) – Small value for the numerical stability.

Methods

add_hook (hook, name=None)
Registers a hook function.
Hook function is typically called right after the gradient computation, though the timing depends on the
optimization method.

Parameters

• hook (function) – Hook function. If hook.call_for_each_param is true, this
hook function is called for each parameter by passing the update rule and the parameter.
Otherwise, this hook function is called only once each iteration by passing the optimizer.
• name (str) – Name of the registration. If omitted, hook.name is used by default.

call_hooks ()
Invokes hook functions in registration order.

create_update_rule ()

new_epoch ()
Starts a new epoch.
This method increments the epoch count. Note that if the optimizer depends on the epoch count, then
user should call this method appropriately at the beginning of each epoch.

reallocate_cleared_grads ()
Reallocate gradients cleared by cleargrad().
This method allocates arrays for all gradients which have None. This method is called before and after
every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override
this method with a blank function.

remove_hook (name)
Removes a hook function.

Parameters name (str) – Registered name of the hook function to remove.

serialize (serializer)
Serializes or deserializes the optimizer.
It only saves or loads the following things:
• Optimizer states
• Global states (t and epoch)
It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

**Parameters**

- **serializer** (*AbstractSerializer*) – Serializer or deserializer object.

**setup**(link)

**update**(lossfun=None, *args, **kwds)

Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

**use_cleargrads**(use=True)

Enables or disables use of `cleargrads()` in `update`.

**Parameters**

- **use** (*bool*) – If True, this function enables use of `cleargrads`. If False, disables use of `cleargrads` (zerograds is used).

Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

**Attributes**

- **alpha**
  
  Alias to `self.hyperparam.alpha`

- **beta1**
  
  Alias to `self.hyperparam.beta1`

- **beta2**
  
  Alias to `self.hyperparam.beta2`

- **eps**
  
  Alias to `self.hyperparam.eps`

- **lr**

**chainer.optimizers.MomentumSGD**

**class** `chainer.optimizers.MomentumSGD`(*lr=0.01, momentum=0.9)

Momentum SGD optimizer.

**Parameters**

- **lr** (*float*) – Learning rate.
- **momentum** (*float*) – Exponential decay rate of the first order moment.

**Methods**

- **add_hook**(hook, name=None)
  
  Registers a hook function.
Hook function is typically called right after the gradient computation, though the timing depends on the optimization method.

**Parameters**

- **hook** (*function*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.

- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.

**call_hooks** ()

Invokes hook functions in registration order.

**create_update_rule** ()

**new_epoch** ()

Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

**reallocate_cleared_grads** ()

Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

**remove_hook** (*name*)

Removes a hook function.

**serialize** (*serializer*)

Serializes or deserializes the optimizer.

It only saves or loads the following things:

- Optimizer states
- Global states (`t` and `epoch`)

**It does not save or load the parameters of the target link.** They should be separately saved or loaded.

**Parameters**

- **serializer** (*AbstractSerializer*) – Serializer or deserializer object.

**setup** (*link*)

**update** (*lossfun=None, *args, **kwds*)

Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- If `lossfun` is given, then it is used as a loss function to compute gradients.
- Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

**use_cleargrads** (*use=True*)

Enables or disables use of `cleargrads()` in update.

**Parameters**

- **use** (*bool*) – If True, this function enables use of `cleargrads`. If False, disables use of `cleargrads` (zerograds is used).
Deprecated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

### Attributes

- `lr`
  - Alias to `self.hyperparam.lr`

- `momentum`
  - Alias to `self.hyperparam.momentum`

### `chainer.optimizers.NesterovAG`

class `chainer.optimizers.NesterovAG(lr=0.01, momentum=0.9)`
Nesterov’s Accelerated Gradient.


#### Parameters

- `lr` (`float`) – Learning rate.
- `momentum` (`float`) – Exponential decay rate of the first order moment.

#### Methods

- `add_hook(hook, name=None)`
  - Registers a hook function.
    
    Hook function is typically called right after the gradient computation, though the timing depends on the optimization method.

    **Parameters**
    
    - `hook` (`function`) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
    - `name` (`str`) – Name of the registration. If omitted, `hook.name` is used by default.

- `call_hooks()`
  - Invokes hook functions in registration order.

- `create_update_rule()`

- `new_epoch()`
  - Starts a new epoch.
    
    This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

- `reallocate_cleared_grads()`
  - Reallocate gradients cleared by `cleargrad()`.
    
    This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.
remove_hook(name)
Removes a hook function.

Parameters name (str) – Registered name of the hook function to remove.

serialize(serializer)
Serializes or deserializes the optimizer.

It only saves or loads the following things:
• Optimizer states
• Global states (t and epoch)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters serializer (AbstractSerializer) – Serializer or deserializer object.

setup(link)

update(lossfun=None, *args, **kwds)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.
• If lossfun is given, then it is used as a loss function to compute gradients.
• Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

use_cleargrads(use=True)
Enables or disables use of cleargrads() in update.

Parameters use (bool) – If True, this function enables use of cleargrads. If False, disables use of cleargrads (zerograds is used).

Deprecated since version v2.0: Note that update() calls cleargrads() by default. cleargrads() is more efficient than zerograds(), so one does not have to call use_cleargrads(). This method remains for backward compatibility.

Attributes

lr
Alias to self.hyperparam.lr

momentum
Alias to self.hyperparam.momentum

chainer.optimizers.RMSprop

class chainer.optimizers.RMSprop(lr=0.01, alpha=0.99, eps=1e-08)
RMSProp optimizer.


Parameters

• lr (float) – Learning rate.
• alpha (float) – Exponential decay rate of the second order moment.
• **eps** *(float)* – Small value for the numerical stability.

**Methods**

```python
add_hook(hook, name=None)
```

Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method.

**Parameters**

- **hook** *(function)* – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.
- **name** *(str)* – Name of the registration. If omitted, `hook.name` is used by default.

```python
call_hooks()
```

Invokes hook functions in registration order.

```python
create_update_rule()
```

```python
new_epoch()
```

Starts a new epoch.

This method increments the `epoch` count. Note that if the optimizer depends on the epoch count, then user should call this method appropriately at the beginning of each epoch.

```python
reallocate_cleared_grads()
```

Reallocate gradients cleared by `cleargrad()`.

This method allocates arrays for all gradients which have `None`. This method is called before and after every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override this method with a blank function.

```python
remove_hook(name)
```

Removes a hook function.

**Parameters**

- **name** *(str)* – Registered name of the hook function to remove.

```python
serialize(serializer)
```

Serializes or deserializes the optimizer.

It only saves or loads the following things:

- **Optimizer states**
- **Global states** *(t and epoch)*

**It does not saves nor loads the parameters of the target link.** They should be separately saved or loaded.

**Parameters**

- **serializer** *(AbstractSerializer)* – Serializer or deserializer object.

```python
setup(link)
```

```python
update(lossfun=None, *args, **kwds)
```

Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

- **If lossfun is given, then it is used as a loss function to compute gradients.**
- **Otherwise, this method assumes that the gradients are already computed.**
In both cases, the computed gradients are used to update parameters. The actual update routines are defined by the update rule of each parameter.

**use_cleargrads** (*use=True*)  
Enables or disables use of `cleargrads()` in `update`.  

**Parameters**  
- **use** (*bool*) – If `True`, this function enables use of `cleargrads`. If `False`, disables use of `cleargrads` (zerograds is used).  

Depreciated since version v2.0: Note that `update()` calls `cleargrads()` by default. `cleargrads()` is more efficient than `zerograds()`, so one does not have to call `use_cleargrads()`. This method remains for backward compatibility.

**Attributes**

- **alpha**  
  Alias to `self.hyperparam.alpha`

- **eps**  
  Alias to `self.hyperparam.eps`

- **lr**  
  Alias to `self.hyperparam.lr`

**chainer.optimizers.RMSpropGraves**

```python
class chainer.optimizers.RMSpropGraves(lr=0.0001, alpha=0.95, momentum=0.9, eps=0.0001)
```

Alex Graves’s RMSprop.  


**Parameters**

- **lr** (*float*) – Learning rate.

- **alpha** (*float*) – Exponential decay rate of the first and second order moments of the raw gradient.

- **momentum** (*float*) – Exponential decay rate of the first order moment of the adjusted gradient.

- **eps** (*float*) – Small value for the numerical stability.

**Methods**

**add_hook** (*hook, name=None*)  
Registers a hook function.  

Hook function is typically called right after the gradient computation, though the timing depends on the optimization method.

**Parameters**

- **hook** (*function*) – Hook function. If `hook.call_for_each_param` is true, this hook function is called for each parameter by passing the update rule and the parameter. Otherwise, this hook function is called only once each iteration by passing the optimizer.

- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.
call_hooks()
   Invokes hook functions in registration order.

create_update_rule()

new_epoch()
   Starts a new epoch.

   This method increments the epoch count. Note that if the optimizer depends on the epoch count, then
   user should call this method appropriately at the beginning of each epoch.

reallocate_cleared_grads()
   Reallocate gradients cleared by cleargrad().

   This method allocates arrays for all gradients which have None. This method is called before and after
   every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override
   this method with a blank function.

remove_hook(name)
   Removes a hook function.

   Parameters name (str) – Registered name of the hook function to remove.

serialize(serializer)
   Serializes or deserializes the optimizer.

   It only saves or loads the following things:
   • Optimizer states
   • Global states (t and epoch)

   It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

   Parameters serializer (AbstractSerializer) – Serializer or deserializer object.

setup(link)

update(lossfun=None, *args, **kwds)
   Updates parameters based on a loss function or computed gradients.

   This method runs in two ways.
   • If lossfun is given, then it is used as a loss function to compute gradients.
   • Otherwise, this method assumes that the gradients are already computed.

   In both cases, the computed gradients are used to update parameters. The actual update routines are defined
   by the update rule of each parameter.

use_cleargrads(use=True)
   Enables or disables use of cleargrads() in update.

   Parameters use (bool) – If True, this function enables use of cleargrads. If False, disables
   use of cleargrads (zero_grads is used).

   Deprecated since version v2.0: Note that update() calls cleargrads() by default.
   cleargrads() is more efficient than zero_grads(), so one does not have to call
   use_cleargrads(). This method remains for backward compatibility.

Attributes

alpha
   Alias to self.hyperparam.alpha
eps
    Alias to self.hyperparam.eps
lr
    Alias to self.hyperparam.lr
momentum
    Alias to self.hyperparam.momentum

chainer.optimizers.SGD

class chainer.optimizers.SGD(lr=0.01)
Vanilla Stochastic Gradient Descent.

Parameters  lr (float) – Learning rate.

Methods

add_hook (hook, name=None)
    Registers a hook function.
    Hook function is typically called right after the gradient computation, though the timing depends on the
    optimization method.

    Parameters

    • hook (function) – Hook function. If hook.call_for_each_param is true, this
      hook function is called for each parameter by passing the update rule and the parameter.
      Otherwise, this hook function is called only once each iteration by passing the optimizer.

    • name (str) – Name of the registration. If omitted, hook.name is used by default.

call_hooks ()
    Invokes hook functions in registration order.

create_update_rule ()

new_epoch ()
    Starts a new epoch.
    This method increments the epoch count. Note that if the optimizer depends on the epoch count, then
    user should call this method appropriately at the beginning of each epoch.

reallocate_cleared_grads ()
    Reallocate gradients cleared by cleargrad().
    This method allocates arrays for all gradients which have None. This method is called before and after
    every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override
    this method with a blank function.

remove_hook (name)
    Removes a hook function.

    Parameters name (str) – Registered name of the hook function to remove.

serialize (serializer)
    Serializes or deserializes the optimizer.
    It only saves or loads the following things:

    • Optimizer states
Global states (t and epoch)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters serializer (AbstractSerializer) – Serializer or deserializer object.

setup (link)

update (lossfun=None, *args, **kwds)

Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

• If lossfun is given, then it is used as a loss function to compute gradients.
• Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined
by the update rule of each parameter.

use_cleargrads (use=True)

Enables or disables use of cleargrads () in update.

Parameters use (bool) – If True, this function enables use of cleargrads. If False, disables
use of cleargrads (zerograds is used).

Deprecated since version v2.0: Note that update() calls cleargrads() by default. cleargrads() is more efficient than zerograds(), so one does not have to call
use_cleargrads (). This method remains for backward compatibility.

Attributes

lr

Alias to self.hyperparam.lr

chainer.optimizers.SMORMS3

class chainer.optimizers.SMORMS3 (lr=0.001, eps=1e-16)

Simon Funk’s SMORMS3.


Parameters

• lr (float) – Learning rate.
• eps (float) – Small value for the numerical stability.

Methods

add_hook (hook, name=None)

Registers a hook function.

Hook function is typically called right after the gradient computation, though the timing depends on the
optimization method.

Parameters
• hook (function) – Hook function. If hook.call_for_each_param is true, this
hook function is called for each parameter by passing the update rule and the parameter.
Otherwise, this hook function is called only once each iteration by passing the optimizer.

• name (str) – Name of the registration. If omitted, hook.name is used by default.

call_hooks()
Invokes hook functions in registration order.

create_update_rule()

new_epoch()
Starts a new epoch.

This method increments the epoch count. Note that if the optimizer depends on the epoch count, then
user should call this method appropriately at the beginning of each epoch.

reallocate_cleared_grads()
Reallocate gradients cleared by cleargrad().

This method allocates arrays for all gradients which have None. This method is called before and after
every optimizer hook. If an inheriting optimizer does not require this allocation, the optimizer can override
this method with a blank function.

remove_hook (name)
Removes a hook function.

Parameters name (str) – Registered name of the hook function to remove.

serialize (serializer)
Serializes or deserializes the optimizer.

It only saves or loads the following things:

• Optimizer states
• Global states (t and epoch)

It does not saves nor loads the parameters of the target link. They should be separately saved or loaded.

Parameters serializer (AbstractSerializer) – Serializer or deserializer object.

setup (link)

update (lossfun=None, *args, **kwds)
Updates parameters based on a loss function or computed gradients.

This method runs in two ways.

• If lossfun is given, then it is used as a loss function to compute gradients.
• Otherwise, this method assumes that the gradients are already computed.

In both cases, the computed gradients are used to update parameters. The actual update routines are defined
by the update rule of each parameter.

use_cleargrads (use=True)
Enables or disables use of cleargrads() in update.

Parameters use (bool) – If True, this function enables use of cleargrads. If False, disables
use of cleargrads (zerograd is used).

Deprecated since version v2.0: Note that update() calls cleargrads() by default.
cleargrads() is more efficient than zerograd(), so one does not have to call
use_cleargrads(). This method remains for backward compatibility.
Attributes

eps
   Alias to `self.hyperparam.eps`

lr
   Alias to `self.hyperparam.lr`

Serializers

Serialization in NumPy NPZ format

NumPy serializers can be used in arbitrary environments that Chainer runs with. It consists of asymmetric serializer/deserializer due to the fact that `numpy.savez()` does not support online serialization. Therefore, serialization requires two-step manipulation: first packing the objects into a flat dictionary, and then serializing it into npz format.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.serializers.NpzDeserializer</code></td>
<td>Deserializer for NPZ format.</td>
</tr>
<tr>
<td><code>chainer.serializers.save_npz</code></td>
<td>Saves an object to the file in NPZ format.</td>
</tr>
<tr>
<td><code>chainer.serializers.load_npz</code></td>
<td>Loads an object from the file in NPZ format.</td>
</tr>
</tbody>
</table>

`chainer.serializers.DictionarySerializer`

class `chainer.serializers.DictionarySerializer`(target=None, path='')

Serializer for dictionary.

This is the standard serializer in Chainer. The hierarchy of objects are simply mapped to a flat dictionary with keys representing the paths to objects in the hierarchy.

Note: Despite of its name, this serializer DOES NOT serialize the object into external files. It just build a flat dictionary of arrays that can be fed into `numpy.savez()` and `numpy.savez_compressed()`. If you want to use this serializer directly, you have to manually send a resulting dictionary to one of these functions.

Parameters

- **target (dict)** – The dictionary that this serializer saves the objects to. If target is None, then a new dictionary is created.
- **path (str)** – The base path in the hierarchy that this serializer indicates.

Variables target (dict) – The target dictionary. Once the serialization completes, this dictionary can be fed into `numpy.savez()` or `numpy.savez_compressed()` to serialize it in the NPZ format.

Methods

- `__call__(key, value)`
- `__getitem__(key)`

3.7. Serializers
save(obj)
Saves an object by this serializer.

This is equivalent to obj.serialize(self).

Parameters

obj – Target object to be serialized.

chainer.serializers.NpzDeserializer

class chainer.serializers.NpzDeserializer(npz, path='', strict=True)
Deserializer for NPZ format.
This is the standard deserializer in Chainer. This deserializer can be used to read an object serialized by save_npz().

Parameters

• npz – npz file object.
• path – The base path that the deserialization starts from.
• strict (bool) – If True, the deserializer raises an error when an expected value is not found in the given NPZ file. Otherwise, it ignores the value and skip deserialization.

Methods

__call__(key, value)
__getitem__(key)
load(obj)
Loads an object from this deserializer.

This is equivalent to obj.serialize(self).

Parameters

obj – Target object to be serialized.

chainer.serializers.save_npz

chainer.serializers.save_npz(filename, obj, compression=True)
Saves an object to the file in NPZ format.

This is a short-cut function to save only one object into an NPZ file.

Parameters

• filename (str) – Target file name.
• obj – Object to be serialized. It must support serialization protocol.
• compression (bool) – If True, compression in the resulting zip file is enabled.

chainer.serializers.load_npz

chainer.serializers.load_npz(filename, obj)
Loads an object from the file in NPZ format.

This is a short-cut function to load from an .npz file that contains only one object.

Parameters
• **filename** (*str*) – Name of the file to be loaded.
• **obj** – Object to be deserialized. It must support serialization protocol.

## Serialization in HDF5 format

<table>
<thead>
<tr>
<th>Chainer.serializers.HDF5Serializer</th>
<th>Serializer for HDF5 format.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chainer.serializers.HDF5Deserializer</td>
<td>Deserializer for HDF5 format.</td>
</tr>
<tr>
<td>Chainer.serializers.save_hdf5</td>
<td>Saves an object to the file in HDF5 format.</td>
</tr>
<tr>
<td>Chainer.serializers.load_hdf5</td>
<td>Loads an object from the file in HDF5 format.</td>
</tr>
</tbody>
</table>

### Chainer.serializers.HDF5Serializer

```python
class chainer.serializers.HDF5Serializer (group, compression=4)
```

This is the standard serializer in Chainer. The chain hierarchy is simply mapped to HDF5 hierarchical groups.

**Parameters**

• **group** (*h5py.Group*) – The group that this serializer represents.
• **compression** (*int*) – Gzip compression level.

**Methods**

- `__call__(key, value)`
- `__getitem__(key)`
- `save(obj)`
  
  Saves an object by this serializer.
  
  This is equivalent to `obj.serialize(self)`.

  **Parameters** **obj** – Target object to be serialized.

### Chainer.serializers.HDF5Deserializer

```python
class chainer.serializers.HDF5Deserializer (group, strict=True)
```

This is the standard deserializer in Chainer. This deserializer can be used to read an object serialized by `HDF5Serializer`.

**Parameters**

• **group** (*h5py.Group*) – The group that the deserialization starts from.
• **strict** (*bool*) – If `True`, the deserializer raises an error when an expected value is not found in the given HDF5 file. Otherwise, it ignores the value and skip deserialization.
Methods

__call__(key, value)
__getitem__(key)
load(obj)

Loads an object from this deserializer.

Parameters

obj – Target object to be serialized.

chainer.serializers.save_hdf5

chainer.serializers.save_hdf5(filename, obj, compression=4)

Saves an object to the file in HDF5 format.

Parameters

filename (str) – Target file name.
obj – Object to be serialized. It must support serialization protocol.
compression (int) – Gzip compression level.

chainer.serializers.load_hdf5

chainer.serializers.load_hdf5(filename, obj)

Loads an object from the file in HDF5 format.

Parameters

filename (str) – Name of the file to be loaded.
obj – Object to be deserialized. It must support serialization protocol.

Function hooks

Chainer provides a function-hook mechanism that enriches the behavior of forward and backward propagation of Function.

Base class

chainer.function.FunctionHook

Base class of hooks for Functions.
**chainer.function.FunctionHook**

**class chainer.function.FunctionHook**
Base class of hooks for Functions.

`FunctionHook` is an callback object that is registered to `Function`. Registered function hooks are invoked before and after forward and backward operations of each function.

Function hooks that derive `FunctionHook` are required to implement four methods: `forward_preprocess()`, `forward_postprocess()`, `backward_preprocess()`, and `backward_postprocess()`. By default, these methods do nothing.

Specifically, when `__call__()` method of some function is invoked, `forward_preprocess()` (resp. `forward_postprocess()`) of all function hooks registered to this function are called before (resp. after) forward propagation.

Likewise, when `backward()` of some `Variable` is invoked, `backward_preprocess()` (resp. `backward_postprocess()`) of all function hooks registered to the function which holds this variable as a gradient are called before (resp. after) backward propagation.

There are two ways to register `FunctionHook` objects to `Function` objects.

First one is to use `with` statement. Function hooks hooked in this way are registered to all functions within `with` statement and are unregistered at the end of `with` statement.

**Example**

The following code is a simple example in which we measure the elapsed time of a part of forward propagation procedure with `TimerHook`, which is a subclass of `FunctionHook`.

```python
>>> from chainer import function_hooks
>>> class Model(chainer.Chain):
...     def __call__(self, x1):
...         return F.exp(self.l(x1))
...     return F.exp(self.l(x1))
>>> model1 = Model(l=L.Linear(10, 10))
>>> model2 = Model(l=L.Linear(10, 10))
>>> x = chainer.Variable(np.zeros((1, 10), 'f'))
>>> with chainer.function_hooks.TimerHook() as m:
...     _ = model1(x)
...     y = model2(x)
...     print("Total time : " + str(m.total_time()))
...     model3 = Model(l=L.Linear(10, 10))
...     z = model3(y)
Total time : ...
```

In this example, we measure the elapsed times for each forward propagation of all functions in `model1` and `model2` (specifically, `LinearFunction` and `Exp` of `model1` and `model2`). Note that `model3` is not a target of measurement as `TimerHook` is unregistered before forward propagation of `model3`.

**Note:** Chainer stores the dictionary of registered function hooks as a thread local object. So, function hooks registered are different depending on threads.

The other one is to register directly to `Function` object with `add_hook()` method. Function hooks registered in this way can be removed by `delete_hook()` method. Contrary to former registration method, function hooks are registered only to the function which `add_hook()` is called.

**Parameters**

- `name (str)` – Name of this function hook.
Methods

__enter__()
__exit__("_

backward_postprocess (function, in_data, out_grad)
Callback function invoked after backward propagation.

Parameters

- **function** (Function) – Function object to which the function hook is registered.
- **in_data** (tuple of numpy.ndarray or tuple of cupy.ndarray) – Input of forward propagation.
- **out_grad** (tuple of numpy.ndarray or tuple of cupy.ndarray) – Gradient data of backward propagation.

backward_preprocess (function, in_data, out_grad)
Callback function invoked before backward propagation.

Parameters

- **function** (Function) – Function object to which the function hook is registered.
- **in_data** (tuple of numpy.ndarray or tuple of cupy.ndarray) – Input data of forward propagation.
- **out_grad** (tuple of numpy.ndarray or tuple of cupy.ndarray) – Gradient data of backward propagation.

forward_postprocess (function, in_data)
Callback function invoked after forward propagation.

Parameters

- **function** (Function) – Function object to which the function hook is registered.
- **in_data** (tuple of numpy.ndarray or tuple of cupy.ndarray) – Input data of forward propagation.

forward_preprocess (function, in_data)
Callback function invoked before forward propagation.

Parameters

- **function** (Function) – Function object to which the function hook is registered.
- **in_data** (tuple of numpy.ndarray or tuple of cupy.ndarray) – Input data of forward propagation.

Attributes

name = ‘FunctionHook’

Concrete function hooks

<table>
<thead>
<tr>
<th>chainer.function_hooks.PrintHook</th>
<th>Function hook that prints debug information.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.function_hooks.TimerHook</td>
<td>Function hook for measuring elapsed time of functions.</td>
</tr>
</tbody>
</table>
chainer.function_hooks.PrintHook

class chainer.function_hooks.PrintHook(sep='', end='n', file=<open file 'stdout'>, mode='w', flush=True)

Function hook that prints debug information.

This function hook outputs the debug information of input arguments of forward and backward methods involved in the hooked functions at preprocessing time (that is, just before each method is called).

Unlike simple “debug print” technique, where users insert print functions at every function to be inspected, we can show the information of all functions involved with single with statement.

Further, this hook enables us to show the information of backward methods without inserting print functions into Chainer’s library code.

Variables

- **sep** – Separator of print function.
- **end** – Character to be added at the end of print function.
- **file** – Output file_like object that that redirect to.
- **flush** – If True, this hook forcibly flushes the text stream at the end of preprocessing.

Example

The basic usage is to use it with with statement.

```python
>>> from chainer import function_hooks
>>> l = L.Linear(10, 10)
>>> x = chainer.Variable(np.zeros((1, 10), 'f'))
>>> with chainer.function_hooks.PrintHook():
... y = l(x)
... z = F.sum(y)
... z.backward()
```

In this example, PrintHook shows the debug information of forward propagation of LinearFunction (which is implicitly called by l) and Sum (called by F.sum) and backward propagation of z and y.

Methods

- **__enter__()**
- **__exit__(exc_type)***

backward_postprocess(function, in_data, out_grad)

Callback function invoked after backward propagation.

Parameters

- **function** (Function) – Function object to which the function hook is registered.
- **in_data** (tuple of numpy.ndarray or tuple of cupy.ndarray) – Input of forward propagation.
- **out_grad** (tuple of numpy.ndarray or tuple of cupy.ndarray) – Gradient data of backward propagation.

backward_preprocess(function, in_data, out_grad)
forward_postprocess (function, in_data)
Callback function invoked after forward propagation.

Parameters
- **function** (Function) – Function object to which the function hook is registered.
- **in_data** (tuple of numpy.ndarray or tuple of cupy.ndarray) – Input data of forward propagation.

forward_preprocess (function, in_data)

Attributes

name = ‘PrintHook’

chainer.function_hooks.TimerHook

class chainer.function_hooks.TimerHook
Function hook for measuring elapsed time of functions.

Variables call_history – List of measurement results. It consists of pairs of the function that calls this hook and the elapsed time the function consumes.

Methods

__enter__()
__exit__(*_)

backward_postprocess (function, in_data, out_grad)
backward_preprocess (function, in_data, out_grad)
forward_postprocess (function, in_data)
forward_preprocess (function, in_data)

total_time ()
Returns total elapsed time in seconds.

Attributes

name = ‘TimerHook’

Weight Initializers

Weight initializer is an instance of *Initializer* that destructively edits the contents of *numpy.ndarray* or *cupy.ndarray*. Typically, weight initializers are passed to __init__ of *Link* and initializes its the weights and biases.

Base class
**chainer.Initializer**

**class** `chainer.Initializer(dtype=None)`  
Initializes array.

It initializes the given array.

**Variables**  
- **dtype** – Data type specifier. It is for type check in `__call__` function.

**Methods**

- `__call__(array)`  
  Initializes given array.

  This method destructively changes the value of array. The derived class is required to implement this method. The algorithms used to make the new values depend on the concrete derived classes.

  **Parameters**  
  - **array** (*numpy.ndarray or cupy.ndarray*) – An array to be initialized by this initializer.

**Concrete initializers**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>chainer.initializers.Identity(scale=1.0, dtype=None)</code></td>
<td>Initializes array with the identity matrix.</td>
</tr>
<tr>
<td><code>chainer.initializers.Constant</code></td>
<td>Initializes array with constant value.</td>
</tr>
<tr>
<td><code>chainer.initializers.Zero</code></td>
<td>Returns initializer that initializes array with the all-zero array.</td>
</tr>
<tr>
<td><code>chainer.initializers.One</code></td>
<td>Returns initializer that initializes array with the all-one array.</td>
</tr>
<tr>
<td><code>chainer.initializers.Normal</code></td>
<td>Initializes array with a normal distribution.</td>
</tr>
<tr>
<td><code>chainer.initializers.GlorotNormal</code></td>
<td>Initializes array with scaled Gaussian distribution.</td>
</tr>
<tr>
<td><code>chainer.initializers.HeNormal</code></td>
<td>Initializes array with scaled Gaussian distribution.</td>
</tr>
<tr>
<td><code>chainer.initializers.Orthogonal</code></td>
<td>Initializes array with an orthogonal system.</td>
</tr>
<tr>
<td><code>chainer.initializers.Uniform</code></td>
<td>Initializes array with a scaled uniform distribution.</td>
</tr>
<tr>
<td><code>chainer.initializers.LeCunUniform</code></td>
<td>Initializes array with a scaled uniform distribution.</td>
</tr>
<tr>
<td><code>chainer.initializers.GlorotUniform</code></td>
<td>Initializes array with a scaled uniform distribution.</td>
</tr>
<tr>
<td><code>chainer.initializers.HeUniform</code></td>
<td>Initializes array with a scaled uniform distribution.</td>
</tr>
</tbody>
</table>

**chainer.initializers.Identity**

**class** `chainer.initializers.Identity(scale=1.0, dtype=None)`  
Initializes array with the identity matrix.

It initializes the given array with the constant multiple of the identity matrix. Note that arrays to be passed must be 2D squared matrices.

**Variables**  
- **scale** (scalar) – A constant to be multiplied to identity matrices.
Methods

```
__call__(array)
```

**chainer.initializers.Constant**

class chainer.initializers.Constant (fill_value, dtype=None)

Initializes array with constant value.

Variables

- **fill_value** *(scalar or numpy.ndarray or cupy.ndarray)* – A constant to be assigned to the initialized array. Broadcast is allowed on this assignment.
- **dtype** – Data type specifier.

Methods

```
__call__(array)
```

**chainer.initializers.Zero**

chainer.initializers.Zero (dtype=None)

Returns initializer that initializes array with the all-zero array.

Parameters **dtype** – Data type specifier.

Returns An initialized array.

Return type numpy.ndarray or cupy.ndarray

**chainer.initializers.One**

chainer.initializers.One (dtype=None)

Returns initializer that initializes array with the all-one array.

Parameters **dtype** – Data type specifier.

Returns An initialized array.

Return type numpy.ndarray or cupy.ndarray

**chainer.initializers.Normal**

class chainer.initializers.Normal (scale=0.05, dtype=None)

Initializes array with a normal distribution.

Each element of the array is initialized by the value drawn independently from Gaussian distribution whose mean is 0, and standard deviation is `scale`.

Parameters

- **scale** *(float)* – Standard deviation of Gaussian distribution.
- **dtype** – Data type specifier.
Methods

__call__(array)

chainer.initializers.GlorotNormal

class chainer.initializers.GlorotNormal (scale=1.0, dtype=None)

Initializes array with scaled Gaussian distribution.

Each element of the array is initialized by the value drawn independently from Gaussian distribution whose mean is 0, and standard deviation is $scale \times \sqrt{\frac{2}{fan_{in}+fan_{out}}}$, where $fan_{in}$ and $fan_{out}$ are the number of input and output units, respectively.

Reference: Glorot & Bengio, AISTATS 2010

Parameters

- scale (float) – A constant that determines the scale of the standard deviation.
- dtype – Data type specifier.

Methods

__call__(array)

chainer.initializers.HeNormal

class chainer.initializers.HeNormal (scale=1.0, dtype=None)

Initializes array with scaled Gaussian distribution.

Each element of the array is initialized by the value drawn independently from Gaussian distribution whose mean is 0, and standard deviation is $scale \times \sqrt{\frac{2}{fan_{in}}}$, where $fan_{in}$ is the number of input units.


Parameters

- scale (float) – A constant that determines the scale of the standard deviation.
- dtype – Data type specifier.

Methods

__call__(array)

chainer.initializers.Orthogonal

class chainer.initializers.Orthogonal (scale=1.1, dtype=None)

Initializes array with an orthogonal system.

This initializer first makes a matrix of the same shape as the array to be initialized whose elements are drawn independently from standard Gaussian distribution. Next, it applies Singular Value Decomposition (SVD) to the matrix. Then, it initializes the array with either side of resultant orthogonal matrices, depending on the shape of the input array. Finally, the array is multiplied by the constant $scale$. 

3.9. Weight Initializers
If the \texttt{ndim} of the input array is more than 2, we consider the array to be a matrix by concatenating all axes except the first one.

The number of vectors consisting of the orthogonal system (i.e. first element of the shape of the array) must be equal to or smaller than the dimension of each vector (i.e. second element of the shape of the array).

**Variables**

- \texttt{\textit{scale}} (float) – A constant to be multiplied by.
- \texttt{\textit{dtype}} – Data type specifier.

**Reference:** Saxe et al., https://arxiv.org/abs/1312.6120

**Methods**

\begin{verbatim}
__call__(array)
\end{verbatim}

\texttt{chainer.initializers.Uniform}

\texttt{class chainer.initializers.Uniform}(\texttt{scale=0.05, dtype=None})

Initializes array with a scaled uniform distribution.

Each element of the array is initialized by the value drawn independently from uniform distribution $[-\texttt{scale}, \texttt{scale}]$.

**Variables**

- \texttt{\textit{scale}} (float) – A constant that determines the scale of the uniform distribution.
- \texttt{\textit{dtype}} – Data type specifier.

**Methods**

\begin{verbatim}
__call__(array)
\end{verbatim}

\texttt{chainer.initializers.LeCunUniform}

\texttt{class chainer.initializers.LeCunUniform}(\texttt{scale=1.0, dtype=None})

Initializes array with a scaled uniform distribution.

Each element of the array is initialized by the value drawn independently from uniform distribution $[-s, s]$ where $s = \texttt{scale} \times \sqrt{\frac{3}{\text{fan}_{in}}}$. Here \textit{fan}$_{in}$ is the number of input units.


**Variables**

- \texttt{\textit{scale}} (float) – A constant that determines the scale of the uniform distribution.
- \texttt{\textit{dtype}} – Data type specifier.

**Methods**

\begin{verbatim}
__call__(array)
\end{verbatim}
chainer.initializers.GlorotUniform

**class chainer.initializers.GlorotUniform(scale=1.0, dtype=None)**

Initializes array with a scaled uniform distribution.

Each element of the array is initialized by the value drawn independently from uniform distribution \([-s, s]\) where 
\[s = \text{scale} \times \sqrt{\frac{6}{\text{fan}_{\text{in}} + \text{fan}_{\text{out}}}}.\]

Here, \(\text{fan}_{\text{in}}\) and \(\text{fan}_{\text{out}}\) are the number of input and output units, respectively.

**Variables**

- `scale (float)` – A constant that determines the scale of the uniform distribution.
- `dtype` – Data type specifier.

**Methods**

```python
__call__(array)
```

chainer.initializers.HeUniform

**class chainer.initializers.HeUniform(scale=1.0, dtype=None)**

Initializes array with scaled uniform distribution.

Each element of the array is initialized by the value drawn independently from uniform distribution \([-s, s]\) where 
\[s = \text{scale} \times \sqrt{\frac{6}{\text{fan}_{\text{in}}}}.\]

Here, \(\text{fan}_{\text{in}}\) is the number of input units.

**Variables**

- `scale (float)` – A constant that determines the scale of the uniform distribution.
- `dtype` – Data type specifier.

**Methods**

```python
__call__(array)
```

**Helper function**

```python
chainer.initializers.generate_array
```

Return initialized array.

chainer.initializers.generate_array

**chainer.initializers.generate_array(initializer, shape, xp)**

Return initialized array.

The algorithms used to make the new values depend on the concrete derived classes. The dtype of a generated
array depends on `initializer.dtype`.

**Parameters**

- `initializer` – A callable object that takes `numpy.ndarray` or `cupy.ndarray` and
  edits its value.
Dataset examples

The most basic dataset implementation is an array. Both NumPy and CuPy arrays can be used directly as datasets. In many cases, though, the simple arrays are not enough to write the training procedure. In order to cover most of such cases, Chainer provides many built-in implementations of datasets.

These built-in datasets are divided into two groups. One is a group of general datasets. Most of them are wrapper of other datasets to introduce some structures (e.g., tuple or dict) to each data point. The other one is a group of concrete, popular datasets. These concrete examples use the downloading utilities in the `chainer.dataset` module to cache downloaded and converted datasets.

General datasets

General datasets are further divided into three types.

The first one is `DictDataset` and `TupleDataset`, both of which combine other datasets and introduce some structures on them.

The second one is `SubDataset`, which represents a subset of an existing dataset. It can be used to separate a dataset for hold-out validation or cross validation. Convenient functions to make random splits are also provided.

The third one is `TransformDataset`, which wraps around a dataset by applying a function to data indexed from the underlying dataset. It can be used to modify behavior of a dataset that is already prepared.

The last one is a group of domain-specific datasets. Currently, `ImageData` and `LabeledImageData` are provided for datasets of images.

DictDataset

`chainer.datasets.DictDataset`

Dataset of a dictionary of datasets.

```python
chainer.datasets.DictDataset(**datasets)
```

Dataset of a dictionary of datasets.

It combines multiple datasets into one dataset. Each example is represented by a dictionary mapping a key to an example of the corresponding dataset.

**Parameters**

- `datasets` – Underlying datasets. The keys are used as the keys of each example. All datasets must have the same length.

**Methods**

- `__getitem__(index)`
TupleDataset

chainer.datasets.TupleDataset

Dataset of a tuple of datasets.

Parameters
datasets – Underlying datasets. The i-th one is used for the i-th item of each example. All datasets must have the same length.

Methods

__getitem__(index)

__len__()
There are two ways to make training-validation splits. One is a single split, where the dataset is split just into two subsets. It can be done by `split_dataset()` or `split_dataset_random()`. The other one is a $k$-fold cross validation, in which the dataset is divided into $k$ subsets, and $k$ different splits are generated using each of the $k$ subsets as a validation set and the rest as a training set. It can be done by `get_cross_validation_datasets()`.

Parameters

- `dataset` – Base dataset.
- `start` (*int*) – The first index in the interval.
- `finish` (*int*) – The next-to-the-last index in the interval.
- `order` (*sequence of ints*) – Permutation of indexes in the base dataset. If this is `None`, then the ascending order of indexes is used.

Methods

`__getitem__(index)`

Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the `get_example()` method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters `index` (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns If index is int, returns an example created by `get_example`. If index is either slice or one-dimensional list or numpy.ndarry, returns a list of examples created by `get_example`.

Example

```python
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]  # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

`__len__()`

`get_example(i)`
chainer.datasets.split_dataset

chainer.datasets.split_dataset(dataset, split_at, order=None)
Splits a dataset into two subsets.

This function creates two instances of SubDataset. These instances do not share any examples, and they
together cover all examples of the original dataset.

Parameters

- **dataset** – Dataset to split.
- **split_at** ([int]) – Position at which the base dataset is split.
- **order** ([sequence of int]) – Permutation of indexes in the base dataset. See the
document of SubDataset for details.

Returns

Two SubDataset objects. The first subset represents the indexes
order[:split_at] while the second subset represents the examples of indexes
order[split_at:].

Return type: tuple

chainer.datasets.split_dataset_random

chainer.datasets.split_dataset_random(dataset, first_size, seed=None)
Splits a dataset into two subsets randomly.

This function creates two instances of SubDataset. These instances do not share any examples, and they
together cover all examples of the original dataset. The split is automatically done randomly.

Parameters

- **dataset** – Dataset to split.
- **first_size** ([int]) – Size of the first subset.
- **seed** ([int]) – Seed the generator used for the permutation of indexes. If an integer being
convertible to 32 bit unsigned integers is specified, it is guaranteed that each sample in the
given dataset always belongs to a specific subset. If None, the permutation is changed
randomly.

Returns

Two SubDataset objects. The first subset contains first_size examples randomly
chosen from the dataset without replacement, and the second subset contains the rest of the
dataset.

Return type: tuple

chainer.datasets.get_cross_validation_datasets

chainer.datasets.get_cross_validation_datasets(dataset, n_fold, order=None)
Creates a set of training/test splits for cross validation.

This function generates n_fold splits of the given dataset. The first part of each split corresponds to the
training dataset, while the second part to the test dataset. No pairs of test datasets share any examples, and all
test datasets together cover the whole base dataset. Each test dataset contains almost same number of examples (the numbers may differ up to 1).

Parameters

- **dataset** – Dataset to split.
- **n_fold** *(int)* – Number of splits for cross validation.
- **order** *(sequence of ints)* – Order of indexes with which each split is determined. If it is None, then no permutation is used.

Returns List of dataset splits.

Return type list of tuples

**chainer.datasets.get_cross_validation_datasets_random**

**chainer.datasets.get_cross_validation_datasets_random**(dataset, n_fold, seed=None)

Creates a set of training/test splits for cross validation randomly.

This function acts almost same as `get_cross_validation_dataset()`, except automatically generating random permutation.

Parameters

- **dataset** – Dataset to split.
- **n_fold** *(int)* – Number of splits for cross validation.
- **seed** *(int)* – Seed the generator used for the permutation of indexes. If an integer beging convertible to 32 bit unsigned integers is specified, it is guaranteed that each sample in the given dataset always belongs to a specific subset. If None, the permutation is changed randomly.

Returns List of dataset splits.

Return type list of tuples

**TransformDataset**

**chainer.datasets.TransformDataset**

**chainer.datasets.TransformDataset** *(dataset, transform)*

Dataset that indexes the base dataset and transforms the data.

This dataset wraps the base dataset by modifying the behavior of the base dataset’s `__getitem__()`. Arrays returned by `__getitem__()` of the base dataset with integer as an argument are transformed by the given function `transform`. Also, `__len__()` returns the integer returned by the base dataset’s `__len__()`. The function `transform` takes, as an argument, `in_data`, which is the output of the base dataset’s `__getitem__()`, and returns the transformed arrays as output. Please see the following example.

```python
>>> from chainer.datasets import get_mnist

>>> from chainer.datasets import TransformDataset
```
>>> dataset, _ = get_mnist()
>>> def transform(in_data):
...     img, label = in_data
...     img -= 0.5  # scale to [-0.5, -0.5]
...     return img, label
>>> dataset = TransformDataset(dataset, transform)

Parameters

- **dataset** – The underlying dataset. The index of this dataset corresponds to the index of the base dataset. This object needs to support functions `__getitem__()` and `__len__()` as described above.

- **transform** (*callable*) – A function that is called to transform values returned by the underlying dataset’s `__getitem__()`.

Methods

`__getitem__(index)`

Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the `get_example()` method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

Parameters

- **index** (*int, slice, list or numpy.ndarray*) – An index of an example or indexes of examples.

Returns

- If index is int, returns an example created by `get_example`. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by `get_example`.

Example

```python
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         self.values = values
...     def __len__(self):
...         return len(self.values)
...     def get_example(self, i):
...         return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]  # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

`__len__()`
get_example(i)

ImageDataset

chainer.datasets.ImageDataset

**Dataset of images built from a list of paths to image files.**

```python
class chainer.datasets.ImageDataset (paths, root='.', dtype=<type 'numpy.float32'>):
```

This dataset reads an external image file on every call of the `__getitem__()` operator. The paths to the image to retrieve is given as either a list of strings or a text file that contains paths in distinct lines.

Each image is automatically converted to arrays of shape `channels, height, width`, where `channels` represents the number of channels in each pixel (e.g., 1 for grey-scale images, and 3 for RGB-color images).

**Note:** This dataset requires the Pillow package being installed. In order to use this dataset, install Pillow (e.g. by using the command `pip install Pillow`). Be careful to prepare appropriate libraries for image formats you want to use (e.g. libpng for PNG images, and libjpeg for JPG images).

**Parameters**

- `paths (str or list of strs)`: If it is a string, it is a path to a text file that contains paths to images in distinct lines. If it is a list of paths, the i-th element represents the path to the i-th image. In both cases, each path is a relative one from the root path given by another argument.
- `root (str)`: Root directory to retrieve images from.
- `dtype`: Data type of resulting image arrays.

**Methods**

`__getitem__(index)`

Returns an example or a sequence of examples.

It implements the standard Python indexing and one-dimensional integer array indexing. It uses the `get_example()` method by default, but it may be overridden by the implementation to, for example, improve the slicing performance.

**Parameters**

- `index (int, slice, list or numpy.ndarray)`: An index of an example or indexes of examples.

**Returns**

If index is int, returns an example created by `get_example`. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by `get_example`.

**Example**

```python
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
...     def __init__(self, values):
...         ...
```
...   self.values = values
... def __len__(self):
...     return len(self.values)
... def get_example(self, i):
...     return self.values[i]
... 
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]  # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]

__len__()

get_example(i)

LabeledImageDataset

counter = 0
for x, y in zip(xs, ys):
    if y:
        counter += 1

counter

counter

Dataset of image and label pairs built from a list of paths and labels.

Dataset of image and label pairs built from a list of paths and labels.

This dataset reads an external image file like ImageDataset. The difference from ImageDataset is that this dataset also returns a label integer. The paths and labels are given as either a list of pairs or a text file contains paths/labels pairs in distinct lines. In the latter case, each path and corresponding label are separated by white spaces. This format is same as one used in Caffe.

Note: This dataset requires the Pillow package being installed. In order to use this dataset, install Pillow (e.g. by using the command pip install Pillow). Be careful to prepare appropriate libraries for image formats you want to use (e.g. libpng for PNG images, and libjpeg for JPG images).

Parameters

- **pairs** (str or list of tuples) – If it is a string, it is a path to a text file that contains paths to images in distinct lines. If it is a list of pairs, the i-th element represents a pair of the path to the i-th image and the corresponding label. In both cases, each path is a relative one from the root path given by another argument.
- **root** (str) – Root directory to retrieve images from.
- **dtype** – Data type of resulting image arrays.
- **label_dtype** – Data type of the labels.
Methods

__getitem__(index)

Parameters

- **index** (int, slice, list or numpy.ndarray) – An index of an example or indexes of examples.

Returns

If index is int, returns an example created by get_example. If index is either slice or one-dimensional list or numpy.ndarray, returns a list of examples created by get_example.

Example

```python
>>> import numpy
>>> from chainer import dataset
>>> class SimpleDataset(dataset.DatasetMixin):
... def __init__(self, values):
...     self.values = values
... def __len__(self):
...     return len(self.values)
... def get_example(self, i):
...     return self.values[i]
...
>>> ds = SimpleDataset([0, 1, 2, 3, 4, 5])
>>> ds[1]  # Access by int
1
>>> ds[1:3]  # Access by slice
[1, 2]
>>> ds[[4, 0]]  # Access by one-dimensional integer list
[4, 0]
>>> index = numpy.arange(3)
>>> ds[index]  # Access by one-dimensional integer numpy.ndarray
[0, 1, 2]
```

__len__()

get_example(i)

Concrete datasets

<table>
<thead>
<tr>
<th>chainer.datasets.get_mnist</th>
<th>Gets the MNIST dataset.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.datasets.get_cifar10</td>
<td>Gets the CIFAR-10 dataset.</td>
</tr>
<tr>
<td>chainer.datasets.get_cifar100</td>
<td>Gets the CIFAR-100 dataset.</td>
</tr>
<tr>
<td>chainer.datasets.get_ptb_words</td>
<td>Gets the Penn Tree Bank dataset as long word sequences.</td>
</tr>
<tr>
<td>chainer.datasets.get_ptb_words_vocabulary</td>
<td>Gets the Penn Tree Bank word vocabulary.</td>
</tr>
</tbody>
</table>
chainer.datasets.get_mnist

chainer.datasets.get_mnist(withlabel=True, ndim=1, scale=1.0, dtype=<type 'numpy.float32'>, label_dtype=<type 'numpy.int32'>)

Gets the MNIST dataset.

MNIST is a set of hand-written digits represented by grey-scale 28x28 images. In the original images, each pixel is represented by a one-byte unsigned integer. This function scales the pixels to floating point values in the interval [0, scale].

This function returns the training set and the test set of the official MNIST dataset. If withlabel is True, each dataset consists of tuples of images and labels, otherwise it only consists of images.

Parameters

• withlabel (bool) – If True, it returns datasets with labels. In this case, each example is a tuple of an image and a label. Otherwise, the datasets only contain images.

• ndim (int) – Number of dimensions of each image. The shape of each image is determined depending on ndim as follows:
  – ndim == 1: the shape is (784,)
  – ndim == 2: the shape is (28, 28)
  – ndim == 3: the shape is (1, 28, 28)

• scale (float) – Pixel value scale. If it is 1 (default), pixels are scaled to the interval [0, 1].

• dtype – Data type of resulting image arrays.

• label_dtype – Data type of the labels.

Returns A tuple of two datasets. If withlabel is True, both datasets are TupleDataset instances. Otherwise, both datasets are arrays of images.

chainer.datasets.get_cifar10

chainer.datasets.get_cifar10(withlabel=True, ndim=3, scale=1.0)

Gets the CIFAR-10 dataset.

CIFAR-10 is a set of small natural images. Each example is an RGB color image of size 32x32, classified into 10 groups. In the original images, each component of pixels is represented by a one-byte unsigned integer. This function scales the components to floating point values in the interval [0, scale].

This function returns the training set and the test set of the official CIFAR-10 dataset. If withlabel is True, each dataset consists of tuples of images and labels, otherwise it only consists of images.

Parameters

• withlabel (bool) – If True, it returns datasets with labels. In this case, each example is a tuple of an image and a label. Otherwise, the datasets only contain images.

• ndim (int) – Number of dimensions of each image. The shape of each image is determined depending on ndim as follows:
  – ndim == 1: the shape is (3072,)
  – ndim == 3: the shape is (3, 32, 32)

• scale (float) – Pixel value scale. If it is 1 (default), pixels are scaled to the interval [0, 1].
Returns A tuple of two datasets. If withlabel is True, both datasets are `TupleDataset` instances. Otherwise, both datasets are arrays of images.

`chainer.datasets.get_cifar100`

`chainer.datasets.get_cifar100(withlabel=True, ndim=3, scale=1.0)`

Gets the CIFAR-100 dataset.

CIFAR-100 is a set of small natural images. Each example is an RGB color image of size 32x32, classified into 100 groups. In the original images, each component pixels is represented by one-byte unsigned integer. This function scales the components to floating point values in the interval \([0, \text{scale}]\).

This function returns the training set and the test set of the official CIFAR-100 dataset. If withlabel is True, each dataset consists of tuples of images and labels, otherwise it only consists of images.

Parameters

- `withlabel (bool)` – If True, it returns datasets with labels. In this case, each example is a tuple of an image and a label. Otherwise, the datasets only contain images.
- `ndim (int)` – Number of dimensions of each image. The shape of each image is determined depending on ndim as follows:
  - ndim == 1: the shape is (3072,)
  - ndim == 3: the shape is (3, 32, 32)
- `scale (float)` – Pixel value scale. If it is 1 (default), pixels are scaled to the interval \([0, 1]\).

Returns A tuple of two datasets. If withlabel is True, both are `TupleDataset` instances. Otherwise, both datasets are arrays of images.

`chainer.datasets.get_ptb_words`

`chainer.datasets.get_ptb_words()`

Gets the Penn Tree Bank dataset as long word sequences.

Penn Tree Bank is originally a corpus of English sentences with linguistic structure annotations. This function uses a variant distributed at https://github.com/tomsercu/lstm, which omits the annotation and splits the dataset into three parts: training, validation, and test.

This function returns the training, validation, and test sets, each of which is represented as a long array of word IDs. All sentences in the dataset are concatenated by End-of-Sentence mark ‘<eos>’, which is treated as one of the vocabulary.

Returns Int32 vectors of word IDs.

Return type tuple of `numpy.ndarray`

See also:

Use `get_ptb_words_vocabulary()` to get the mapping between the words and word IDs.

`chainer.datasets.get_ptb_words_vocabulary`

`chainer.datasets.get_ptb_words_vocabulary()`

Gets the Penn Tree Bank word vocabulary.

Returns
Dictionary that maps words to corresponding word IDs. The IDs are used in the Penn Tree Bank long sequence datasets.

Return type: dict

See also:
See `get_ptb_words()` for the actual datasets.

### Iterator examples

Chainer provides some iterators that implement typical strategies to create mini-batches by iterating over datasets. `SerialIterator` is the simplest one, which extract mini batches in the main thread. `MultiprocessIterator` is a parallelized version of `SerialIterator`. It maintains worker subprocesses to load the next mini-batch in parallel.

<table>
<thead>
<tr>
<th>chainer.iterators.SerialIterator</th>
<th>Dataset iterator that serially reads the examples.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.iterators.MultiprocessIterator</td>
<td>Dataset iterator that loads examples in parallel.</td>
</tr>
</tbody>
</table>

#### chainer.iterators.SerialIterator

```python
class chainer.iterators.SerialIterator(dataset, batch_size, repeat=True, shuffle=True)
```

Dataset iterator that serially reads the examples.

This is a simple implementation of `Iterator` that just visits each example in either the order of indexes or a shuffled order.

To avoid unintentional performance degradation, the `shuffle` option is set to `True` by default. For validation, it is better to set it to `False` when the underlying dataset supports fast slicing. If the order of examples has an important meaning and the updater depends on the original order, this option should be set to `False`.

This iterator saves `-1` instead of `None` in snapshots since some serializers do not support `None`.

**Parameters**

- `dataset` – Dataset to iterate.
- `batch_size` (*int*) – Number of examples within each batch.
- `repeat` (*bool*) – If `True`, it infinitely loops over the dataset. Otherwise, it stops iteration at the end of the first epoch.
- `shuffle` (*bool*) – If `True`, the order of examples is shuffled at the beginning of each epoch. Otherwise, examples are extracted in the order of indexes.

**Methods**

- `__next__()`:
- `__iter__()`:
  Returns self.
- `finalize()`:
  Finalizes the iterator and possibly releases the resources.

  This method does nothing by default. Implementation may override it to better handle the internal resources.
next()
reset()
serialize(serializer)

Attributes

epoch_detail
previous_epoch_detail

chainer.iterators.MultiprocessIterator

class chainer.iterators.MultiprocessIterator(dataset, batch_size, repeat=True, shuffle=True, n_processes=None, n_prefetch=1, shared_mem=None)

Dataset iterator that loads examples in parallel.

This is an implementation of Iterator that loads examples with worker processes. It uses the standard multiprocessing module to parallelize the loading. The dataset is sent to the worker processes in the standard way using pickle.

Note that this iterator effectively prefetches the examples for the next batch asynchronously after the current batch is returned.

This iterator saves -1 instead of None in snapshots since some serializers do not support None.

Parameters

• dataset (Dataset) – Dataset to iterate.
• batch_size (int) – Number of examples within each batch.
• repeat (bool) – If True, it infinitely loops over the dataset. Otherwise, it stops iteration at the end of the first epoch.
• shuffle (bool) – If True, the order of examples is shuffled at the beginning of each epoch. Otherwise, examples are extracted in the order of indexes.
• n_processes (int) – Number of worker processes. The number of CPUs is used by default.
• n_prefetch (int) – Number of prefetch batches.
• shared_mem (int) – The size of using shared memory per data. If None, size is adjusted automatically.

Methods

__next__()
__iter__()
Returns self.
finalize()
next()
reset()
serialize(serializer)

**Attributes**

epoch_detail

previous_epoch_detail

## Trainer extensions

<table>
<thead>
<tr>
<th>chainer.training.extensions.dump_graph</th>
<th>Returns a trainer extension to dump a computational graph.</th>
</tr>
</thead>
<tbody>
<tr>
<td>chainer.training.extensions.Evaluator</td>
<td>Trainer extension to evaluate models on a validation set.</td>
</tr>
<tr>
<td>chainer.training.extensions.ExponentialShift</td>
<td>Trainer extension to exponentially shift an optimizer attribute.</td>
</tr>
<tr>
<td>chainer.training.extensions.LinearShift</td>
<td>Trainer extension to change an optimizer attribute linearly.</td>
</tr>
<tr>
<td>chainer.training.extensions.LogReport</td>
<td>Trainer extension to output the accumulated results to a log file.</td>
</tr>
<tr>
<td>chainer.training.extensions.observe_lr</td>
<td>Returns a trainer extension to record the learning rate.</td>
</tr>
<tr>
<td>chainer.training.extensions.observe_value</td>
<td>Returns a trainer extension to continuously record a value.</td>
</tr>
<tr>
<td>chainer.training.extensions.snapshot</td>
<td>Returns a trainer extension to take snapshots of the trainer.</td>
</tr>
<tr>
<td>chainer.training.extensions.snapshot_object</td>
<td>Returns a trainer extension to take snapshots of a given object.</td>
</tr>
<tr>
<td>chainer.training.extensions.PlotReport</td>
<td>Trainer extension to output plots.</td>
</tr>
<tr>
<td>chainer.training.extensions.PrintReport</td>
<td>Trainer extension to print the accumulated results.</td>
</tr>
<tr>
<td>chainer.training.extensions.ProgressBar</td>
<td>Trainer extension to print a progress bar and recent training status.</td>
</tr>
</tbody>
</table>

### chainer.training.extensions.dump_graph

chainer.training.extensions.dump_graph(root_name, out_name='cg.dot', variable_style=None, function_style=None)

Returns a trainer extension to dump a computational graph.

This extension dumps a computational graph. The graph is output in DOT language.

It only dumps a graph at the first invocation.

**Note:** As of v2.0.0, the computational graph is not kept by default. This extension changes this behavior until the first invocation. **It is strongly recommended to use it with the default trigger setting.**

The detailed behavior of this extension since v2.0.0 is as follows.

1. In its initializer, it turns on the chainer.config.keep_graph_on_report flag.
2. At the first iteration, it dumps the graph using the graph held by the reported variable.
3. After dumping the graph, it turns off the flag (if it was originally turned off) so that any variable reported afterward does not hold a computational graph.

When the keep_graph_on_report flag is turned on, the computational graph created by the updater is kept during the invocation of extensions. It will cause an unnecessarily large memory consumption when an extension also uses a large amount of memory, e.g. Evaluator.
With the default setting, the dump_graph extension is called at the first iteration. Since Evaluator is not called at the first iteration in most cases, it does not cause any memory problem.

Parameters

- **root_name** *(str)* – Name of the root of the computational graph. The root variable is retrieved by this name from the observation dictionary of the trainer.
- **out_name** *(str)* – Output file name.
- **variable_style** *(dict)* – Dot node style for variables. Each variable is rendered by an octagon by default.
- **function_style** *(dict)* – Dot node style for functions. Each function is rendered by a rectangular by default.

See also:

See `build_computational_graph()` for the variable_style and function_style arguments.

chainer.training.extensions.Evaluator

class chainer.training.extensions.Evaluator(iterator, target, converter=<function concat_examples>, device=None, eval_hook=None, eval_func=None)

Trainer extension to evaluate models on a validation set.

This extension evaluates the current models by a given evaluation function. It creates a Reporter object to store values observed in the evaluation function on each iteration. The report for all iterations are aggregated to DictSummary. The collected mean values are further reported to the reporter object of the trainer, where the name of each observation is prefixed by the evaluator name. See Reporter for details in naming rules of the reports.

Evaluator has a structure to customize similar to that of StandardUpdater. The main differences are:

- There are no optimizers in an evaluator. Instead, it holds links to evaluate.
- An evaluation loop function is used instead of an update function.
- Preparation routine can be customized, which is called before each evaluation. It can be used, e.g., to initialize the state of stateful recurrent networks.

There are two ways to modify the evaluation behavior besides setting a custom evaluation function. One is by setting a custom evaluation loop via the eval_func argument. The other is by inheriting this class and overriding the evaluate() method. In latter case, users have to create and handle a reporter object manually. Users also have to copy the iterators before using them, in order to reuse them at the next time of evaluation. In both cases, the functions are called in testing mode (i.e., chainer.config.train is set to False).

This extension is called at the end of each epoch by default.

Parameters

- **iterator** – Dataset iterator for the validation dataset. It can also be a dictionary of iterators. If this is just an iterator, the iterator is registered by the name 'main'.
- **target** – Link object or a dictionary of links to evaluate. If this is just a link object, the link is registered by the name 'main'.
- **converter** – Converter function to build input arrays. concat_examples() is used by default.
• **device** – Device to which the training data is sent. Negative value indicates the host memory (CPU).

• **eval_hook** – Function to prepare for each evaluation process. It is called at the beginning of the evaluation. The evaluator extension object is passed at each call.

• **eval_func** – Evaluation function called at each iteration. The target link to evaluate as a callable is used by default.

**Variables**

• **converter** – Converter function.

• **device** – Device to which the training data is sent.

• **eval_hook** – Function to prepare for each evaluation process.

• **eval_func** – Evaluation function called at each iteration.

**Methods**

__call__(trainer=None)

Executes the evaluator extension.

Unlike usual extensions, this extension can be executed without passing a trainer object. This extension reports the performance on validation dataset using the report() function. Thus, users can use this extension independently from any trainer by manually configuring a Reporter object.

Parameters

- **trainer** (Trainer) – Trainer object that invokes this extension. It can be omitted in case of calling this extension manually.

Returns

- **Result dictionary that contains mean statistics of values** reported by the evaluation function.

  **Return type** dict

**evaluate()**

Evaluates the model and returns a result dictionary.

This method runs the evaluation loop over the validation dataset. It accumulates the reported values to DictSummary and returns a dictionary whose values are means computed by the summary.

Users can override this method to customize the evaluation routine.

Returns

- **Result dictionary. This dictionary is further reported via report() without specifying any observer.**

  **Return type** dict

**finalize()**

Finalizes the extension.

This method is called at the end of the training loop.

**get_all_iterators()**

Returns a dictionary of all iterators.

**get_all_targets()**

Returns a dictionary of all target links.
get_iterator (name)
Returns the iterator of the given name.

get_target (name)
Returns the target link of the given name.

initialize (trainer)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of Trainer can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, ExponentialShift extension changes the optimizer’s hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The ExponentialShift extension recovers it in its initialize method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters

trainer (Trainer) – Trainer object that runs the training loop.

serialize (serializer)
Serializes the extension state.

It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

Attributes

default_name = ‘validation’
priority = 300
trigger = (1, ‘epoch’)

chainer.training.extensions.ExponentialShift

class chainer.training.extensions.ExponentialShift (attr, rate, init=None, target=None, optimizer=None)

Trainer extension to exponentially shift an optimizer attribute.

This extension exponentially increases or decreases the specified attribute of the optimizer. The typical use case is an exponential decay of the learning rate.

This extension is also called before the training loop starts by default.

Parameters

- attr (str) – Name of the attribute to shift.
- rate (float) – Rate of the exponential shift. This value is multiplied to the attribute at each call.
- init (float) – Initial value of the attribute. If it is None, the extension extracts the attribute at the first call and uses it as the initial value.
- target (float) – Target value of the attribute. If the attribute reaches this value, the shift stops.
- optimizer (Optimizer) – Target optimizer to adjust the attribute. If it is None, the main optimizer of the updater is used.
Methods

__call__(trainer)

finalize()
Finalizes the extension.

This method is called at the end of the training loop.

initialize(trainer)

serialize(serializer)

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

priority = 100

trigger = (1, ‘iteration’)

chainer.training.extensions.LinearShift
class chainer.training.extensions.LinearShift(attr, value_range, time_range, optimizer=None)

Trainer extension to change an optimizer attribute linearly.

This extension changes an optimizer attribute from the first value to the last value linearly within a specified duration. The typical use case is warming up of the momentum coefficient.

For example, suppose that this extension is called at every iteration, and value_range == (x, y) and time_range == (i, j). Then, this extension keeps the attribute to be x up to the i-th iteration, linearly shifts the value to y by the j-th iteration, and then keeps the value to be y after the j-th iteration.

This extension is also called before the training loop starts by default.

Parameters

• attr (str) – Name of the optimizer attribute to adjust.
• value_range (tuple of float) – The first and the last values of the attribute.
• time_range (tuple of ints) – The first and last counts of calls in which the attribute is adjusted.
• optimizer (Optimizer) – Target optimizer object. If it is None, the main optimizer of the trainer is used.

Methods

__call__(trainer)

finalize()
Finalizes the extension.

This method is called at the end of the training loop.
initialize (trainer)
serialize (serializer)

Attributes

default_name
Default name of the extension.
   It is the name of the class by default. Implementation can override this property, or provide a class attribute
to hide it.
priority = 100
trigger = (1, 'iteration')

chainer.training.extensions.LogReport

class chainer.training.extensions.LogReport (keys=None, trigger=(1, 'epoch'), postprocess=None, log_name='log')

Trainer extension to output the accumulated results to a log file.

This extension accumulates the observations of the trainer to DictSummary at a regular interval specified by
a supplied trigger, and writes them into a log file in JSON format.

There are two triggers to handle this extension. One is the trigger to invoke this extension, which is used
to handle the timing of accumulating the results. It is set to 1, 'iteration' by default. The other is the
trigger to determine when to emit the result. When this trigger returns True, this extension appends the summary
of accumulated values to the list of past summaries, and writes the list to the log file. Then, this extension makes
a new fresh summary object which is used until the next time that the trigger fires.

It also adds some entries to each result dictionary.
   • 'epoch' and 'iteration' are the epoch and iteration counts at the output, respectively.
   • 'elapsed_time' is the elapsed time in seconds since the training begins. The value is taken from
     Trainer.elapsed_time.

Parameters
   • keys (iterable of strs) – Keys of values to accumulate. If this is None, all the
     values are accumulated and output to the log file.
   • trigger – Trigger that decides when to aggregate the result and output the values. This
     is distinct from the trigger of this extension itself. If it is a tuple in the form <int>,
     'epoch' or <int>, 'iteration', it is passed to IntervalTrigger.
   • postprocess – Callback to postprocess the result dictionaries. Each result dictionary is
     passed to this callback on the output. This callback can modify the result dictionaries, which
     are used to output to the log file.
   • log_name (str) – Name of the log file under the output directory. It can be a format
     string: the last result dictionary is passed for the formatting. For example, users can use
     '{iteration}' to separate the log files for different iterations. If the log name is None, it does
     not output the log to any file.
Methods

__call__ (trainer)

finalize ()
Finalizes the extension.

This method is called at the end of the training loop.

initialize (trainer)
Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of Trainer can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, ExponentialShift extension changes the optimizer’s hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The ExponentialShift extension recovers it in its initialize method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters:

trainer (Trainer) – Trainer object that runs the training loop.

serialize (serializer)

Attributes

default_name
Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

log
The current list of observation dictionaries.

priority = 100

trigger = (1, ‘iteration’)

chainer.training.extensions.observe_lr

chainer.training.extensions.observe_lr (optimizer_name=’main’, observation_key=’lr’)
Returns a trainer extension to record the learning rate.

Parameters:

• optimizer_name (str) – Name of optimizer whose learning rate is recorded.

• observation_key (str) – Key of observation to record.

Returns The extension function.

chainer.training.extensions.observe_value

chainer.training.extensions.observe_value (observation_key, target_func)
Returns a trainer extension to continuously record a value.
Parameters

- **observation_key** *(str)* – Key of observation to record.
- **target_func** *(function)* – Function that returns the value to record. It must take one argument: :class:`chainer.training.Trainer` object.

**Returns** The extension function.

```
chainer.training.extensions.snapshot
```

chainer.training.extensions.snapshot *(savefun=<function save_npz>, filename='snapshot_iter_{.updater.iteration}')*

Returns a trainer extension to take snapshots of the trainer.

This extension serializes the trainer object and saves it to the output directory. It is used to support resuming the training loop from the saved state.

This extension is called once per epoch by default. To take a snapshot at a different interval, a trigger object specifying the required interval can be passed along with this extension to the `extend()` method of the trainer.

The default priority is -100, which is lower than that of most built-in extensions.

**Note:** This extension first writes the serialized object to a temporary file and then rename it to the target file name. Thus, if the program stops right before the renaming, the temporary file might be left in the output directory.

```
chainer.training.extensions.snapshot_object
```

chainer.training.extensions.snapshot_object *(target, filename, savefun=<function save_npz>)*

Returns a trainer extension to take snapshots of a given object.

This extension serializes the given object and saves it to the output directory.

This extension is called once per epoch by default. To take a snapshot at a different interval, a trigger object specifying the required interval can be passed along with this extension to the `extend()` method of the trainer.

The default priority is -100, which is lower than that of most built-in extensions.

**Parameters**

- **target** – Object to serialize.
- **filename** *(str)* – Name of the file into which the object is serialized. It can be a format string, where the trainer object is passed to the `str.format()` method. For example, `'snapshot_{.updater.iteration}'` is converted to `'snapshot_10000'` at the 10,000th iteration.
- **savefun** – Function to save the object. It takes two arguments: the output file path and the object to serialize.
Returns  An extension function.

```
chainer.training.extensions.PlotReport
```

class chainer.training.extensions.PlotReport(y_keys, x_key='iteration', trigger=(1, 'epoch'), postprocess=None, file_name='plot.png', marker='x', grid=True)

Trainer extension to output plots.

This extension accumulates the observations of the trainer to `DictSummary` at a regular interval specified by a supplied trigger, and plot a graph with using them.

There are two triggers to handle this extension. One is the trigger to invoke this extension, which is used to handle the timing of accumulating the results. It is set to 1, 'iteration' by default. The other is the trigger to determine when to emit the result. When this trigger returns True, this extension appends the summary of accumulated values to the list of past summaries, and writes the list to the log file. Then, this extension makes a new fresh summary object which is used until the next time that the trigger fires.

It also adds 'epoch' and 'iteration' entries to each result dictionary, which are the epoch and iteration counts at the output.

```
Warning:  If your environment needs to specify a backend of matplotlib explicitly, please call matplotlib.use before importing Chainer. For example:

import matplotlib
matplotlib.use('Agg')
import chainer

Then, once chainer.training.extensions is imported, matplotlib.use will have no effect.
```

For the details, please see here: http://matplotlib.org/faq/usage_faq.html#what-is-a-backend

Parameters

- **y_keys** *(iterable of strs)* – Keys of values regarded as y. If this is None, nothing is output to the graph.
- **x_key** *(str)* – Keys of values regarded as x. The default value is 'iteration'.
- **trigger** – Trigger that decides when to aggregate the result and output the values. This is distinct from the trigger of this extension itself. If it is a tuple in the form `<int>, 'epoch'>`, '<epoch> or `<int>, 'iteration'>`, it is passed to `IntervalTrigger`.
- **postprocess** – Callback to postprocess the result dictionaries. Figure object, Axes object, and all plot data are passed to this callback in this order. This callback can modify the figure.
- **file_name** *(str)* – Name of the figure file under the output directory. It can be a format string.
- **marker** *(str)* – The marker used to plot the graph. Default is 'x'. If `None` is given, it draws with no markers.
- **grid** *(bool)* – Set the axis grid on if True. Default is True.
Methods

__call__(trainer)

static available()

finalize()

This method is called at the end of the training loop.

initialize(trainer)

Initializes up the trainer state.

This method is called before entering the training loop. An extension that modifies the state of Trainer can override this method to initialize it.

When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.

For example, ExponentialShift extension changes the optimizer’s hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The ExponentialShift extension recovers it in its initialize method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters

trainer (Trainer) – Trainer object that runs the training loop.

serialize (serializer)

Attributes

default_name

Default name of the extension.

It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

priority = 100

trigger = (1, ‘iteration’)

chainer.training.extensions.PrintReport


Trainer extension to print the accumulated results.

This extension uses the log accumulated by a LogReport extension to print specified entries of the log in a human-readable format.

Parameters

• entries (list of str) – List of keys of observations to print.

• log_report (str or LogReport) – Log report to accumulate the observations. This is either the name of a LogReport extensions registered to the trainer, or a LogReport instance to use internally.

• out – Stream to print the bar. Standard output is used by default.
Methods

__call__ (trainer)

finalize ()
Finalizes the extension.
This method is called at the end of the training loop.

initialize (trainer)
Initializes up the trainer state.
This method is called before entering the training loop. An extension that modifies the state of Trainer can override this method to initialize it.
When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.
For example, ExponentialShift extension changes the optimizer’s hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The ExponentialShift extension recovers it in its initialize method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters
trainer (Trainer) – Trainer object that runs the training loop.

serialize (serializer)

Attributes

default_name
Default name of the extension.
It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.

priority = 100

trigger = (1, ‘iteration’)

chainer.training.extensions.ProgressBar

class chainer.training.extensions.ProgressBar (training_length=None, update_interval=100, bar_length=50, out=<open file ‘<stdout>’, mode ‘w’>)

Trainer extension to print a progress bar and recent training status.
This extension prints a progress bar at every call. It watches the current iteration and epoch to print the bar.

Parameters

• training_length (tuple) – Length of whole training. It consists of an integer and either 'epoch' or 'iteration'. If this value is omitted and the stop trigger of the trainer is IntervalTrigger, this extension uses its attributes to determine the length of the training.
• update_interval (int) – Number of iterations to skip printing the progress bar.
• bar_length (int) – Length of the progress bar in characters.
• out – Stream to print the bar. Standard output is used by default.
Methods

__call__(trainer)
finalize()
initialize(trainer)

Initializes the trainer state.
This method is called before entering the training loop. An extension that modifies the state of Trainer can override this method to initialize it.
When the trainer has been restored from a snapshot, this method has to recover an appropriate part of the state of the trainer.
For example, ExponentialShift extension changes the optimizer’s hyperparameter at each invocation. Note that the hyperparameter is not saved to the snapshot; it is the responsibility of the extension to recover the hyperparameter. The ExponentialShift extension recovers it in its initialize method if it has been loaded from a snapshot, or just setting the initial value otherwise.

Parameters
trainer (Trainer) – Trainer object that runs the training loop.

serialize(serializer)
Serializes the extension state.
It is called when a trainer that owns this extension is serialized. It serializes nothing by default.

Attributes

default_name
Default name of the extension.
It is the name of the class by default. Implementation can override this property, or provide a class attribute to hide it.
priority = 100
trigger = (1, ‘iteration’)
chainer.training.triggers.IntervalTrigger

class chainer.training.triggers.IntervalTrigger(period, unit)
    Trigger based on a fixed interval.
    This trigger accepts iterations divided by a given interval. There are two ways to specify the interval: per
    iterations and epochs. Iteration means the number of updates, while epoch means the number of sweeps over
    the training dataset. Fractional values are allowed if the interval is a number of epochs; the trigger uses the
    iteration and epoch_detail attributes defined by the updater.
    For the description of triggers, see get_trigger().

Parameters

• period (int or float) – Length of the interval. Must be an integer if unit is
  'iteration'.
• unit (str) – Unit of the length specified by period. It must be either 'iteration'
  or 'epoch'.

Methods

__call__(trainer)
    Decides whether the extension should be called on this iteration.

Parameters trainer (Trainer) – Trainer object that this trigger is associated with. The
updater associated with this trainer is used to determine if the trigger should fire.

Returns

True if the corresponding extension should be invoked in this iteration.

Return type bool

chainer.training.triggers.ManualScheduleTrigger

class chainer.training.triggers.ManualScheduleTrigger(points, unit)
    Trigger invoked at specified point(s) of iterations or epochs.
    This trigger accepts iterations or epochs indicated by given point(s). There are two ways to specify the point(s):
    iteration and epoch. Iteration means the number of updates, while epoch means the number of sweeps over
    the training dataset. Fractional values are allowed if the point is a number of epochs; the trigger uses the
    iteration and epoch_detail attributes defined by the updater.

Parameters

• points (int, float, or list of int or float) – time of the trigger. Must
  be an integer or list of integer if unit is 'iteration'.
• unit (str) – Unit of the time specified by points. It must be either 'iteration'
  or 'epoch'.

Methods

__call__(trainer)
    Decides whether the extension should be called on this iteration.
Parameters `trainer` *(Trainer)* – Trainer object that this trigger is associated with. The updater associated with this trainer is used to determine if the trigger should fire.

Returns

- **True if the corresponding extension should be invoked in this** iteration.

Return type `bool`

### chainer.training.triggers.MaxValueTrigger

**class** `chainer.training.triggers.MaxValueTrigger(key, trigger=(1, 'epoch'))`

Trigger invoked when specific value becomes maximum.

For example you can use this trigger to take snapshot on the epoch the validation accuracy is maximum.

**Parameters**

- `key` *(str)* – Key of value. The trigger fires when the value associated with this key becomes maximum.

- `trigger` – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `IntervalTrigger`.

**Methods**

`__call__(trainer)`

Decides whether the extension should be called on this iteration.

**Parameters**

- `trainer` *(Trainer)* – Trainer object that this trigger is associated with. The observation of this trainer is used to determine if the trigger should fire.

**Returns**

- **True if the corresponding extension should be invoked in** this iteration.

**Return type** `bool`

### chainer.training.triggers.MinValueTrigger

**class** `chainer.training.triggers.MinValueTrigger(key, trigger=(1, 'epoch'))`

Trigger invoked when specific value becomes minimum.

For example you can use this trigger to take snapshot on the epoch the validation loss is minimum.

**Parameters**

- `key` *(str)* – Key of value. The trigger fires when the value associated with this key becomes minimum.

- `trigger` – Trigger that decides the comparison interval between current best value and new value. This must be a tuple in the form of `<int>, 'epoch'` or `<int>, 'iteration'` which is passed to `IntervalTrigger`. 
Methods

___call___(trainer)
Decides whether the extension should be called on this iteration.

Parameters
trainer (Trainer) – Trainer object that this trigger is associated with. The
observation of this trainer is used to determine if the trigger should fire.

Returns
True if the corresponding extension should be invoked in this iteration.

Return type
bool

Caffe Reference Model Support

Caffe is a popular framework maintained by BVLC at UC Berkeley. It is widely used by computer vision communities,
and aims at fast computation and easy usage without any programming. The BVLC team provides trained reference
models in their Model Zoo, one of the reason why this framework gets popular.

Chainer can import the reference models and emulate the network by Link implementations. This functionality is
provided by the chainer.links.caffe.CaffeFunction class.

chainer.links.caffe.CaffeFunction
Caffe emulator based on the model file of Caffe.

chainer.links.caffe.CaffeFunction
Caffe emulator based on the model file of Caffe.

class chainer.links.caffe.CaffeFunction(model_path)
Caffe emulator based on the model file of Caffe.

Given a protocol buffers file of a Caffe model, this class loads and emulates it on Variable objects. It supports
the official reference models provided by BVLC.

Note: protobuf>=3.0.0 is required if you use Python 3 because protobuf 2 is not supported on Python 3.

Note: CaffeFunction ignores the following layers:

• Layers that CaffeFunction does not support (including data layers)
• Layers that have no top blobs
• Layers whose bottom blobs are incomplete (i.e., some or all of them are not given nor computed)

Warning: It does not support full compatibility against Caffe. Some layers and configurations are not
implemented in Chainer yet, though the reference models provided by the BVLC team are supported except
data layers.

Example

Consider we want to extract the (unnormalized) log class probability of given images using BVLC reference CaffeNet. The model can be downloaded from:

http://dl.caffe.berkeleyvision.org/bvlc_reference_caffenet.caffemodel

We want to compute the fc8 blob from the data blob. It is simply written as follows:

```python
# Load the model
func = CaffeFunction('path/to/bvlc_reference_caffenet.caffemodel')

# Minibatch of size 10
x_data = numpy.ndarray((10, 3, 227, 227), dtype=numpy.float32)
... # (Fill the minibatch here)

# Forward the pre-trained net
x = Variable(x_data)
y, = func(inputs={'data': x}, outputs=['fc8'])
```

The result y contains the Variable corresponding to the fc8 blob. The computational graph is memorized as a usual forward computation in Chainer, so we can run backprop through this pre-trained net.

### Parameters

- **model_path** (str) – Path to the binary-proto model file of Caffe.

### Variables

- **forwards** (dict) – A mapping from layer names to corresponding functions.

### Methods

- **__call__**(self, inputs, outputs, disable=())

  Executes a sub-network of the network.

  This function acts as an interpreter of the network definition for Caffe. On execution, it interprets each layer one by one, and if the bottom blobs are already computed, then emulates the layer and stores output blobs as Variable objects.

  **Warning:** train argument is not supported anymore since v2. Instead, use chainer.using_config('train', train). See chainer.using_config().

### Parameters

- **inputs** (dict) – A dictionary whose key-value pairs indicate initial correspondences between blob names and Variable objects.

- **outputs** (Iterable) – A list of blob names whose corresponding Variable objects are returned.

- **disable** (Iterable) – A list of layer names that will be ignored during the forward computation.

### Returns

- A tuple of output Variable objects corresponding to elements of the outputs argument.

### Return type
tuple

- **__getitem__**(name)

  Equivalent to getattr.
**add_link** (*name*, *link*)

Registers a child link to this chain.

Deprecated since version v2.0.0: Assign the child link directly to an attribute within an initialization scope, instead. For example, the following code

```python
chain.add_link('ll', L.Linear(3, 5))
```

can be replaced by the following line.

```python
with self.init_scope():
    chain.ll = L.Linear(3, 5)
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- *name* (*str*) – Name of the child link. This name is also used as the attribute name.
- *link* (*Link*) – The link object to be registered.

**add_param** (*name*, *shape=None*, *dtype=<type 'numpy.float32'>*, *initializer=None*)

Registers a parameter to the link.

Deprecated since version v2.0.0: Assign a Parameter object directly to an attribute within an initialization scope instead. For example, the following code

```python
link.add_param('W', shape=(5, 3))
```

can be replaced by the following assignment.

```python
with self.init_scope():
    link.W = chainer.Parameter(None, (5, 3))
```

The latter one is easier for IDEs to keep track of the attribute’s type.

**Parameters**

- *name* (*str*) – Name of the parameter. This name is also used as the attribute name.
- *shape* (*int or tuple of ints*) – Shape of the parameter array. If it is omitted, the parameter variable is left uninitialized.
- *dtype* – Data type of the parameter array.
- *initializer* – If it is not None, the data is initialized with the given initializer. If it is an array, the data is directly initialized by it. If it is callable, it is used as a weight initializer. Note that in these cases, dtype argument is ignored.

**add_persistent** (*name*, *value*)

Registers a persistent value to the link.

The registered value is saved and loaded on serialization and deserialization. The value is set to an attribute of the link.

**Parameters**

- *name* (*str*) – Name of the persistent value. This name is also used for the attribute name.
- *value* – Value to be registered.

**addgrads** (*link*)

**children** ()
cleargrads()
Clears all gradient arrays.
This method should be called before the backward computation at every iteration of the optimization.

copy()
copyparams(link)
disable_update()
Disables update rules of all parameters under the link hierarchy.
This method sets the :attr:`chainer.UpdateRule.enabled` flag of the update rule of each parameter variable to False.
enable_update()
Enables update rules of all parameters under the link hierarchy.
This method sets the `enabled` flag of the update rule of each parameter variable to True.

init_scope(*args, **kwds)
Creates an initialization scope.
This method returns a context manager object that enables registration of parameters (and links for :class:`Chain`) by an assignment. A :class:`Parameter` object can be automatically registered by assigning it to an attribute under this context manager.

Example
In most cases, the parameter registration is done in the initializer method. Using the `init_scope` method, we can simply assign a :class:`Parameter` object to register it to the link.

class MyLink(chainer.Link):
    def __init__(self):
        super().__init__()
        with self.init_scope():
            self.W = chainer.Parameter(0, (10, 5))
            self.b = chainer.Parameter(0, (5,))

links(skipself=False)
namedlinks(skipself=False)
namedparams(include_uninit=True)
params(include_uninit=True)

register_persistent(name)
 Registers an attribute of a given name as a persistent value.
This is a convenient method to register an existing attribute as a persistent value. If `name` has been already registered as a parameter, this method removes it from the list of parameter names and re-registers it as a persistent value.

Parameters name (str) – Name of the attribute to be registered.

serialize(name (str) – Name of the attribute to be registered.

to_cpu()
to_gpu(device=None)
**zerograds()**

Initializes all gradient arrays by zero.

This method can be used for the same purpose of cleargrads, but less efficient. This method is left for backward compatibility.

Deprecated since version v1.15: Use cleargrads() instead.

**Attributes**

- **update_enabled**
  
  True if at least one parameter has an update rule enabled.

- **within_init_scope**
  
  True if the current code is inside of an initialization scope.

  See init_scope() for the details of the initialization scope.

- **xp**
  
  Array module for this link.

  Depending on which of CPU/GPU this link is on, this property returns numpy or cupy.

**Visualization of Computational Graph**

As neural networks get larger and complicated, it gets much harder to confirm if their architectures are constructed properly. Chainer supports visualization of computational graphs. Users can generate computational graphs by invoking build_computational_graph(). Generated computational graphs are dumped to specified format (Currently Dot Language is supported).

Basic usage is as follows:

```python
import chainer.computational_graph as c
...
g = c.build_computational_graph(vs)
with open('path/to/output/file', 'w') as o:
    o.write(g.dump())
```

where `vs` is list of Variable instances and `g` is an instance of ComputationalGraph. This code generates the computational graph that are backward-reachable (i.e. reachable by repetition of steps backward) from at least one of `vs`.

Here is an example of (a part of) the generated graph (inception(3a) in GoogLeNet). This example is from example/imagenet.

```
<table>
<thead>
<tr>
<th>chainer.computational_graph.</th>
<th>Builds a graph of functions and variables backward-reachable from outputs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>build_computational_graph</td>
<td></td>
</tr>
<tr>
<td>chainer.computational_graph.</td>
<td>Class that represents computational graph.</td>
</tr>
<tr>
<td>ComputationalGraph</td>
<td></td>
</tr>
</tbody>
</table>
```

3.15. Visualization of Computational Graph 437
chainer.computational_graph.build_computational_graph

chainer.computational_graph.build_computational_graph(outputs, remove_split=True, variable_style={'shape': 'octagon', 'style': 'filled', 'fillcolor': '#E0E0E0'}, function_style={'shape': 'record', 'style': 'filled', 'fillcolor': '#6495ED'}, rankdir='TB', remove_variable=False, show_name=True)

Builds a graph of functions and variables backward-reachable from outputs.

Parameters

- **outputs (list)** – nodes from which the graph is constructed. Each element of outputs must be either Variable object, VariableNode object, or :class:`Function` object.
- **remove_split (bool)** – It must be True. This argument is left for backward compatibility.
- **variable_style (dict)** – Dot node style for variable. Possible keys are ‘shape’, ‘color’, ‘fillcolor’, ‘style’, and etc.
- **function_style (dict)** – Dot node style for function.
- **rankdir (str)** – Direction of the graph that must be TB (top to bottom), BT (bottom to top), LR (left to right) or RL (right to left).
- **remove_variable (bool)** – If True, Variable s are removed from the resulting computational graph. Only Functions are shown in the output.
- **show_name (bool)** – If True, the name attribute of each node is added to the label of the node. Default is True.

Returns

A graph consisting of nodes and edges that are backward-reachable from at least one of outputs.
If `unchain_backward` was called in some variable in the computational graph before this function, backward step is stopped at this variable.

For example, suppose that computational graph is as follows:

```
|--> f ---> y
\n|--> g ---> z
```

Let `outputs = [y, z]`. Then the full graph is emitted.

Next, let `outputs = [y]`. Note that `z` and `g` are not backward-reachable from `y`. The resulting graph would be following:

```
x ---> f ---> y
```

See `TestGraphBuilder` for details.

**Return type** `ComputationalGraph`

---

**Note:** The default behavior of `ComputationalGraph` has been changed from v1.23.0, so that it outputs the richest representation of a graph as default, namely, styles are set and names of functions and variables are shown. To reproduce the same result as previous versions (<= v1.22.0), please specify `variable_style=None`, `function_style=None`, and `show_name=False` explicitly.

---

### chainer.computational_graph.ComputationalGraph

```python
class chainer.computational_graph.ComputationalGraph
```

Class that represents computational graph.

**Note:** We assume that the computational graph is directed and acyclic.

**Parameters**

- **nodes (list)** – List of nodes. Each node is either `VariableNode` object or `Function` object.
- **edges (list)** – List of edges. Each edge consists of pair of nodes.
- **variable_style (dict)** – Dot node style for variable.
- **function_style (dict)** – Dot node style for function.
- **rankdir (str)** – Direction of the graph that must be TB (top to bottom), BT (bottom to top), LR (left to right) or RL (right to left).
- **remove_variable (bool)** – If `True`, `Variable`s are removed from the resulting computational graph. Only `Function`s are shown in the output.
• `show_name` (bool) – If True, the name attribute of each node is added to the label of the node. Default is True.

**Note:** The default behavior of `ComputationalGraph` has been changed from v1.23.0, so that it outputs the richest representation of a graph as default, namely, styles are set and names of functions and variables are shown. To reproduce the same result as previous versions (<= v1.22.0), please specify `variable_style=None`, `function_style=None`, and `show_name=False` explicitly.

### Methods

**dump** *(format='dot')*

Dumps graph as a text.

**Parameters**

- **format** (str) – The graph language name of the output.

**Returns** The graph in specified format.

**Return type** str

### Environment variables

Here are the environment variables Chainer uses.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAINER_CUDNN</td>
<td>Set 0 to disable cuDNN in Chainer. Otherwise cuDNN is enabled automatically.</td>
</tr>
<tr>
<td>CHAINER_SEED</td>
<td>Default seed value of random number generators for CUDA. If it is not set, the seed value is generated from Python random module. Set an integer value in decimal format.</td>
</tr>
<tr>
<td>CHAINER_TYPE_CHECK</td>
<td>Set 0 to disable type checking. Otherwise type checking is enabled automatically. See <code>Function</code> for details.</td>
</tr>
</tbody>
</table>
This document provides detailed information of differences between Chainer v1 and v2. You will know by reading it which part of your code is required (or recommended) to be fixed when you upgrade Chainer from v1 to v2.

- **CuPy**
  - CuPy has been separated from Chainer into a separate package
- **Global configurations**
  - Training mode is configured by a thread-local flag
  - Configurations are added and replace some of existing global flags
- **Variable**
  - Volatile flag is removed
  - Variable is not a part of a computational graph anymore
  - Parameter has to be an instance of Parameter class
  - Small changes to Variable
- **Function**
  - The force_tuple option of split_axis is True by default
  - Type check APIs are updated to enable lazy building of the error messages
  - Methods to release unneeded arrays are added
- **Link/Chain/ChainList**
  - wscale option is removed from links
  - bias option is removed from links
  - The bias vector is enabled by default in N-dimensional convolution links
- `init_weight` function is removed
- The order of arguments of GRU is changed
- The default value of the forget bias for LSTM and StatelessLSTM is changed to 1
- The interfaces of GRU and LSTM are aligned
- Aliases of links in `chainer.functions` are removed
- Parameter link is removed
- New-style parameter registration APIs are added to `Link`
- New-style child link registration APIs are added to `Chain`
- The input-size placeholder of links are made optional

**Optimizer**
- Deprecated methods of `Optimizer` are removed
- `GradientMethod` uses `Link.cleargrads` instead of `Link.zerograds` by default
- `GradientMethod` is redesigned to allow parameter-specific update rules

**Serializer**
- None is serializable

**Trainer and Extension**
- Updater and Evaluator pass raw data arrays to the loss function
- `trigger` option is removed from snapshot and `snapshot_object`
- `Extension.invoke_before_training` is removed
- The `dump_graph` extension dumps the valid graph only at its first invocation

**Reporter**
- When a variable is reported, the variable is copied with the graph purged

**Other utilities**
- Some obsolete classes and functions are removed

---

**CuPy**

**CuPy has been separated from Chainer into a separate package**

CuPy, which was originally a part of Chainer, has been separated into a different Python package since Chainer v2. It changes the way to set up Chainer with CUDA support. In particular, you have to separately install CuPy package to enable CUDA support. See Installation Guide for the recommended installation steps.

Fortunately, there is no need of updating your source code to catch up with this change.
Global configurations

Training mode is configured by a thread-local flag

In Chainer v2, the concept of training mode is added. It is represented by a thread-local flag `chainer.config.train`, which is a part of the unified configuration. When `chainer.config.train` is True, functions of Chainer run in the training mode, and otherwise they run in the test mode. For example, `BatchNormalization` and `dropout()` behave differently in each mode.

In Chainer v1, such a behavior was configured by the `train` or `test` argument of each function. This `train/test argument has been removed in Chainer v2`. If your code is using the `train` or `test` argument, you have to update it. In most cases, what you have to do is just removing the `train/test` argument from any function calls.

Example

Consider the following model definition and the code to call it in test mode written for Chainer v1.

```python
# Chainer v1
import chainer.functions as F

class MyModel(chainer.Link):
    ...

    def __call__(self, x, train=True):
        return f(F.dropout(x, train=train))

m = MyModel(...)
y = m(x, train=False)
```

In Chainer v2, it should be updated into the following code:

```python
# Chainer v2
import chainer.functions as F

class MyModel(chainer.Link):
    ...

    def __call__(self, x):
        return f(F.dropout(x))

m = MyModel(...)
with chainer.using_config('train', False):
    y = m(x)
```

Configurations are added and replace some of existing global flags

There are many global settings moved to the unified configuration other than the training mode. Following is the complete list of the configuration entries that have corresponding features in Chainer v1.

- `chainer.config.cudnn_deterministic` It is corresponding to the `deterministic` argument of some convolution functions in Chainer v1. This argument has been removed since Chainer v2. If you are using this argument, you have to use the `chainer.config.cudnn_deterministic` flag to change the behavior of the convolution functions.
chainer.config.debug It is corresponding to the debug mode in Chainer v1, which was configured by set_debug() and extracted by is_debug(). These functions are also available in Chainer v2, so you basically do not need to update the code related to the debug mode.

chainer.config.enable_backprop It is corresponding to the backprop mode in Chainer v1. The functions no_backprop_mode() and force_backprop_mode() are still available in Chainer v2, which automatically turns on/off the enable_backprop flag. One important difference from Chainer v1 is that the volatile flag is removed from Variable. Therefore, there are more situations that you need to modify the enable_backprop flag.

chainer.config.keep_graph_on_report This flag configures whether or not to keep the computational graph alive for a reported variable. In Chainer v2, when a Variable object is reported by report(), a copy of the variable isolated from the computational graph is created and stored by default. Setting True to this flag, you can change this behavior and then the original Variable object is stored as is. See When a variable is reported, the variable is copied with the graph purged for the details.

chainer.config.train It is corresponding to the train or test argument of some functions in Chainer v1. This argument has been removed since Chainer v2. If you are using this argument, you have to use the chainer.config.train flag instead. See Training mode is configured by a thread-local flag for more details.

chainer.config.type_check It is corresponding to the Function.type_check_enable flag. If your code touches this flag, you have to use chainer.config.type_check instead. Note that the environment variable CHAINER_TYPE_CHECK is still available in Chainer v2, so if you are only using the environment variable, there is no need of updating your code.

chainer.config.use_cudnn It is corresponding to the use_cudnn argument of many functions that have cuDNN implementations. This argument has been removed since Chainer v2. If you are using this argument, you have to use the chainer.config.use_cudnn flag instead. Note that this flag is ternary, not binary. See Configuring Chainer for more details.

These configurations can be modified in two ways.

- Simply substituting a new value to an entry, like chainer.config.train = False.
- Using the chainer.using_config context manager. It can be used with the with statement of Python as follows:

```python
with chainer.using_config('train', False):
    do something  # this code runs with chainer.config.train == False
```

It recovers the original configuration after quitting the with block.

The chainer.config manages the thread-local configuration. You can also set the global configuration by modifying chainer.global_config. Note that the global configuration is used only if the entry of the thread-local configuration is not explicitly set up.

**Variable**

**Volatile flag is removed**

The Variable.volatile flag has been removed since Chainer v2.

Instead, the configuration chainer.config.enable_backprop can be used to enable/disable the automatic differentiation feature. If it is True, Chainer always creates a computational graph on the forward propagation, which corresponds to passing non-volatile variables in Chainer v1. Otherwise, Chainer does not create a graph, which corresponds to passing volatile variables in Chainer v1. The biggest difference is that enable_backprop is a
thread-local flag, whereas volatile was a flag local to each Variable object. Note that enable_backprop flag has already existed in Chainer v1, which took effect only if all the inputs to the function have volatile == 'auto'.

The chainer.config.enable_backprop flag can be modified directly or by using using_config(). See Configuring Chainer for details. There is also a convenience function, no_backprop_mode(), to turn off the flag. If you are using the Variable.volatile flag, you have to stop setting this flag (it will not take effect), and set the enable_backprop flag instead.

Example

Let model be your model, and consider the following code that calls it in volatile mode.

```python
# Chainer v1
x_data = ...  # ndarray
x = chainer.Variable(x_data, volatile=True)
y = model(x)
```

In Chainer v2, it should be updated as follows.

```python
# Chainer v2
x_data = ...  # ndarray
x = chainer.Variable(x_data)
with chainer.no_backprop_mode():
    y = model(x)
```

Variable is not a part of a computational graph anymore

The Variable class has been separated into two distinct classes, the Variable class and the VariableNode class, since Chainer v2. Every class:Variable object owns its own VariableNode object. A computational graph consists of Function objects and VariableNode objects. When one applies a Function to a Variable, the VariableNode object of the variable is extracted and set to one of the inputs of the function.

Note that the underlying data array of the variable is till held by the Variable object. It allows each Function implementation to release unneeded arrays from the computational graph, resulting in greatly reduced memory consumption.

This change does not affect most users’ code. If you are directly traversing the computational graph by yourself or modifying the graph ad-hoc, you may have to update your code. In most cases, it is enough to just change Variable into VariableNode in the code traversing the computational graph.

Parameter has to be an instance of Parameter class

Chainer v2 has a subclass of Variable called Parameter. This class has an interface convenient on setting up a parameter variable registered to Link.

You basically do not need to update your code because Link.add_param() creates a Parameter object in Chainer v2. There is a new recommended way of registering parameters to a link in Chainer v2, though. See here for the recommended way of parameter registration.

Small changes to Variable

There are some changes on the interface and specification of methods.
• `len(variable)` returns the length of the first axis of the underlying array in Chainer v2. This is equivalent to `len(variable.data)`. It is different from the behavior of Chainer v1, in which `len` returned the total number of elements in the underlying array.
• `repr(variable)` returns a NumPy-like text representation of the underlying array in Chainer v2. In Chainer v1, it just returns a string that shows the name of the variable.

**Function**

**The force_tuple option of split_axis is True by default**

In Chainer v2, the `force_tuple` argument of `functions.split_axis()` is set to `True` by default. Therefore, it always returns a tuple regardless of the number of sections made after the split. It was `False` by default in Chainer v1.

**Type check APIs are updated to enable lazy building of the error messages**

In Chainer v2, the type check APIs are updated so that the overhead of checking types is greatly reduced. In order to achieve the overhead reduction, some APIs are changed.

*If you have custom Function implementations that do type checking, you have to update your code.* The following list shows which part has to be updated.

• Use `utils.type_check.eval()` instead of `Expr.eval`.
• Use `utils.type_check.make_variable()` to create a `utils.type_check.Variable` object instead of directly constructing it by yourself.
• Stop using `.name` attribute of any expression.

*Background of this change:* In Chainer v1, the type checking APIs build an abstract syntax tree (AST) based on each expression that tests some condition. The AST is used to emit a kind error message. However, building an AST requires constructions of many Python objects, which adds large Python overheads. In Chainer v2, the `Function.type_check_forward()` method is called once or twice. At the first call, the type checking APIs run in *light-weight mode*, where it does not build an AST and just checks the condition. The second call is made only if there is a test that fails, where it builds an AST. This change makes the ordinary path of running the type checking much faster, while keeping the kindful error messages.

**Methods to release unneeded arrays are added**

*As is written above,* Chainer v2 introduced a new mechanism to reduce the memory consumption of each `Function` implementation. In many cases, a `Function` implementation does not need some input arrays in its backward computation. A new method called `Function.retain_inputs()` can be used to specify which input arrays are actually needed. This method must not be called from the outside of `Function.forward()`.

**Example**

For example, consider the following simple addition function.

```python
class AddFunction(chainer.Function):
    def forward(self, inputs):
        return inputs[0] + inputs[1],
```
def backward(self, inputs, grad_outputs):
    return grad_outputs[0], grad_outputs[0]

It can be seen that the backward computation of this function does not use any of the inputs. Then, specifying an empty tuple of indexes to `retain_inputs()` will reduce the memory overhead.

class AddFunction(chainer.Function):
    def forward(self, inputs):
        self.retain_inputs(()).# does not retain both inputs
        return inputs[0] + inputs[1],
    
    def backward(self, inputs, grad_outputs):
        return grad_outputs[0], grad_outputs[0]

In some cases, the function can (or have to) use the output arrays instead of the inputs in its backward computation. In Chainer v1, we have written code that store the output arrays to attributes of the Function object and reuse them in the `backward()` method. In Chainer v2, it is recommended to use `Function.retain_outputs()` to declare which outputs are required in the backward computation. The retained output arrays can be accessed via `Function.output_data`.

**Note:** The existing Function implementations that store the output arrays to its attributes will run correctly in Chainer v2. There is no any memory overhead right now. It is recommended to use `retain_outputs()`, though, so that we can incorporate more memory optimization in the future.

**Example**

For example, consider the following simple implementation of the tanh function.

class TanhFunction(chainer.Function):
    def forward(self, inputs):
        xp = chainer.cuda.get_array_module(inputs[0])
        self.y = xp.tanh(inputs[0])
        return self.y,
    
    def backward(self, inputs, grad_outputs):
        one = self.y.dtype.type(1)  # avoid type promotion
        return grad_outputs[0] * (one - self.y * self.y),

We can use `retain_outputs()` instead of preserving the output array by ourselves as follows.

class TanhFunction(chainer.Function):
    def forward(self, inputs):
        self.retain_outputs((0,))
        xp = chainer.cuda.get_array_module(inputs[0])
        return xp.tanh(inputs[0]),
    
    def backward(self, inputs, grad_outputs):
        y = self.output_data[0]
        one = y.dtype.type(1)  # avoid type promotion
        return grad_outputs[0] * (one - y * y)
Link/Chain/ChainList

wscale option is removed from links

The wscale option has been removed from links since Chainer v2. **If you are using wscale option, you have to update your code.** The recommended way is to explicitly set the initializer.

**Example**

Consider the case of adding a Linear link with the weight initialized by 0.5x of the default initialization.

```python
# Chainer v1
linear = chainer.links.Linear(10, 5, wscale=0.5)
```

Note that the default initializer of the weight matrix of Linear is a normal distribution of the standard deviation $1/\sqrt{\text{fan in}}$. Therefore, it can be fixed as follows.

```python
# Chainer v2
linear = chainer.links.Linear(10, 5, initialW=chainer.initializers.Normal(0.5 / math.sqrt(10)))
```

Or, by using the fact that initializers.HeNormal provides the initialization with a normal distribution of the standard deviation $scale \times \sqrt{2/fan in}$, the following code is also equivalent to the original.

```python
# Chainer v2, using HeNormal
linear = chainer.links.Linear(10, 5, initialW=chainer.initializers.HeNormal(0.5 / math.sqrt(2)))
```

bias option is removed from links

In Chainer v2, the bias option is removed from the following links: Linear, Convolution2D, Deconvolution2D, and DilatedConvolution2D. The effect of this argument was duplicated with the initial_bias option. Use initial_bias instead.

The bias vector is enabled by default in N-dimensional convolution links

In Chainer v2, the bias parameter is enabled by default in ConvolutionND and DeconvolutionND. It was unintentionally disabled by default in Chainer v1.

**If you are using ConvolutionND or DeconvolutionND without specifying the initial_bias argument, you have to fix your code.** If you want to keep the old behavior (i.e., no bias vector is created by the link), pass nobias=True to the link at the construction. Otherwise it will automatically create a bias vector.

init_weight function is removed

The chainer.initializers.init_weight function that was used on weight initialization has been removed since Chainer v2.

**You have to update your code if you are using init_weight.** In most cases, the update is simple: pass an initializer to Parameter.
Example

Consider the following code that initializes a weight matrix randomly and a bias vector by zero.

```python
# Chainer v1
class MyLink(chainer.Link):
    def __init__(self):
        super(MyLink, self).__init__(
            W=(10, 5),
            b=(5,),
        )
        chainer.initializers.init_weight(self.W, chainer.initializers.Normal(0.05))
        self.b.data.fill(0)

...  
```

This code should be fixed as follows (see the next topic for the use of Parameter).

```python
# Chainer v2
class MyLink(chainer.Link):
    def __init__(self):
        super(MyLink, self).__init__()
        self.W = chainer.Parameter(chainer.initializers.Normal(0.05), (10, 5))
        self.b = chainer.Parameter(0, (5,))

...  
```

The order of arguments of GRU is changed

In Chainer v2, the first two arguments of GRU is the input size and the output size. It was reversed in Chainer v1, causing an inconsistent interface compared to other links including LSTM. If you are using GRU, you have to update your code. The update is done by simply flipping the first two arguments.

Example

Consider the following code that creates a GRU link.

```python
# Chainer v1
gru = chainer.links.GRU(20, 10)
```

It should be fixed into the following code.

```python
# Chainer v2
gru = chainer.links.GRU(10, 20)
```

Note that if you were omitting the output size, the code works as is because GRU supports the omitted input size.

```python
# Chainer v1/v2
gru = chainer.links.GRU(20)
```

The default value of the forget bias for LSTM and StatelessLSTM is changed to 1

In Chainer v2, the default forget bias value of LSTM and StatelessLSTM links is changed to 1. This change is based on the paper reporting that using a large forget bias improves the training performance. The new behavior is
also consistent with the implementation of BasicLSTMCell in TensorFlow.

It will improve the most use cases of LSTMs, although this change would break the reproducibility of the existing experiments. **If you want to keep the same initialization procedure, you have to update your code.** The change is simple: pass `forget_bias_init=0` to `LSTM` and `StatelessLSTM`.

**The interfaces of GRU and LSTM are aligned**

In Chainer v1, **GRU** was stateless, as opposed to the current implementation. To align with the naming convention of LSTM links, we have changed the naming convention from Chainer v2 so that the shorthand name points the stateful links. **If you are using** `StatelessGRU` for stateless version, whose implementation is identical to `chainer.links.GRU` in v1.

**Aliases of links in chainer.functions are removed**

For the compatibility reason, there were some links that have aliases in the `chainer.functions` module. These aliases are removed in Chainer v2. Use `chainer.links` instead.

**Parameter link is removed**

The `chainer.links.Parameter` link is removed in Chainer v2. This link existed in Chainer v1 only for the backward compatibility. Use `chainer.Parameter` instead (for the new `Parameter` class, see `Parameter has to be an instance of Parameter class`).

**New-style parameter registration APIs are added to Link**

In Chainer v2, `Link.init_scope()` method returns a context manager that automatically registers a `Parameter` object to the link at setting it to an attribute. If you are using IDE like PyCharm, it is recommended to use this new-style parameter registration so that IDEs can easily detect the existence of the parameter as an attribute. It is also a good practice to use the new-style API even if you are not using IDEs, if you are planning to make the code public.

**Note:** The existing code that uses the conventional way of registering parameters are still valid.

**Example**

For example, the following link initialization code

```python
# Chainer v1
class MyLink(chainer.Link):
    def __init__(self):
        super(MyLink, self).__init__(
            W=(10, 5),
            b=(5,),
        )
        chainer.initializers.Normal(0.05)(self.W.data)
        self.b.data.fill(0)
```

is recommended to be updated as follows.
# Chainer v2

class MyLink(chainer.Link):
    def __init__(self):
        super(MyLink, self).__init__()
        with self.init_scope():
            self.W = chainer.Parameter(chainer.initializers.Normal(0.05), (10, 5))
            self.b = chainer.Parameter(0, (5,))  # initialize by zero
...

**Note:** To keep a `Parameter` object as an attribute without registration, you can set the attribute without using the with `self.init_scope()` block.

## New-style child link registration APIs are added to Chain

Like `Parameter`, a `Link` object is also automatically registered to a `Chain` object by substitution to an attribute within a `init_scope()` scope. If you are using IDE like PyCharm, it is recommended to use the new-style child link registration so that IDEs can easily detect the existence of the child link as an attribute. It is also a good practice to use the new-style API even if you are not using IDEs, if you are planning to make the code public.

**Note:** The existing code that uses the conventional way of registering child links are still valid.

### Example

For example, the following chain initialization code

```python
# Chainer v1
class MyMLP(chainer.Chain):
    def __init__(self):
        super(MyMLP, self).__init__(
            layer1=L.Linear(None, 20),
            layer2=L.Linear(None, 30),
        )
...
```

is recommended to be updated as follows.

```python
# Chainer v2
class MyMLP(chainer.Chain):
    def __init__(self):
        super(MyMLP, self).__init__()
        with self.init_scope():
            self.layer1 = L.Linear(20)
            self.layer2 = L.Linear(30)
```

Note that this example also demonstrates the use of new APIs with the *omitted input size*, explained below.

**Note:** To keep a `Link` object as an attribute without registration, you can set the attribute without using the with `self.init_scope()` block.

4.5. Link/Chain/ChainList
The input-size placeholder of links are made optional

In Chainer v2, the input size of many links, including Linear and Convolution2D, is made optional. In Chainer v1, we had to use None as the placeholder to specify that the input size should be determined at the first iteration. The placeholder can also be used in Chainer v2, although it is easier to just omit the input size.

See the previous item for the example of omitting the input size of Linear. The following links currently support the omitted input size.

- Convolution2D
- Deconvolution2D
- DilatedConvolution2D
- Linear
- LSTM
- MLPConvolution2D
- StatelessLSTM

Optimizer

Deprecated methods of Optimizer are removed

The following methods are removed from Optimizer. These methods have been already deprecated in the past versions. If you are using these methods, you have to update your code.

- zero_grads: use Link.zerograds() instead.
- compute_grads_norm: you can compute the gradient norm by iterating the list of parameters by Link.params().
- clip_grads: use GradientClipping instead.
- weight_decay: use WeightDecay instead.
- accumulate_grads: use Link.addgrads() instead.

GradientMethod uses Link.cleargrads instead of Link.zerograds by default

In Chainer v2, GradientMethod clears the gradient before running backprop by Link.cleargrads(). It means that the gradient of each parameter is initialized by None instead of a zero array. Note that all the optimizer implementations provided by Chainer are subclasses of GradientMethod, and therefore this change affects all of them.

In most cases, you do not need to update your code. If your code relies on the zeroing initialization, you have to fix your code to explicitly initialize the gradient by zero, or to pass False to GradientMethod.use_cleargrads().

GradientMethod is redesigned to allow parameter-specific update rules

In Chainer v2, the new class UpdateRule is used to define an update rule specific to each Parameter object. The UpdateRule is set to each Parameter object, and is used at each update step. This object implements an update formula using the data and gradient arrays.
Each `UpdateRule` object has an `enabled` flag, which configures if the update rule should be applied to that parameter on update. By setting the flag to `False`, you can freeze the parameter. There is also a convenient method `Link.enable_update()` and `Link.disable_update()`, which configure the flag of each parameter under the link hierarchy. In other frameworks, a similar feature is called layer freezing. In Chainer v2, this is officially supported by these methods.

Each `UpdateRule` object can also hold its own hook functions similar to `Optimizer`. The built-in hook functions except for `GradientClipping` can also be used as a hook function of `UpdateRule`.

**In most cases, you do not have to update your code** because each optimizer automatically sets up an appropriate `UpdaterRule` object to each parameter.

If you are using a custom gradient-based optimizer implementation, you need to update the implementation. The following list shows what you have to do.

- Write a subclass of `UpdateRule` that implements the update rule.
- Rewrite your `GradientMethod` implementation. The new implementation only has to set up the update rule for each parameter in the target link.

You can see live examples in the optimizer implementations provided by Chainer.

## Serializer

### None is serializable

In Chainer v2, all serializers start supporting `None` value to be serialized and deserialized. Users’ code can rely on this feature, i.e., it can serialize and deserialize `None` value with any given serializer. This change only affects your code if it provides its own serializer implementations.

## Trainer and Extension

### Updater and Evaluator pass raw data arrays to the loss function

In Chainer v2, `Updater` and `Evaluator` pass raw data arrays to the loss function without wrapping them with `Variable`. You might need to update your code so that the loss function (in most cases, the model’s `__call__`) accepts raw arrays.

Note that raw arrays can be directly passed to any `Function`; they are automatically wrapped by `Variable`. For example, if the input is directly passed to a `Function` object (or any function under `chainer.functions`), you do not need to update the code.

#### Example

Consider the following code that obtains the shape of the input via `Variable.data`.

```python
# Chainer v1
class MyLink(chainer.Link):
    def __call__(self, x):
        shape = x.data.shape  # valid if x is Variable, invalid if x is ndarray
        ...
```

It should be updated so that the link also accepts a raw array as the input. In this case, we have `Variable.shape` which is equivalent to `data.shape`, so you can simply write as follows.
trigger option is removed from snapshot and snapshot_object

In Chainer v2, the trigger option is removed from the snapshot() and snapshot_object() extensions. The effect of the option was duplicated with the trigger option of Trainer.extend. If you are passing the trigger argument to these extensions, you have to update your code. The update can be done by passing the value to the corresponding Trainer.extend.

Example

Assume that trainer is an instance of Trainer, and consider that you were adding a snapshot() extension as follows.

```python
# Chainer v1
trainer.extend(chainer.training.extensions.snapshot(trigger=(1000, 'iteration')))
```

It should be updated as follows (note that this code also works with Chainer v1).

```python
# Chainer v1/v2
trainer.extend(chainer.training.extensions.snapshot(), trigger=(1000, 'iteration'))
```

Extension.invoke_before_training is removed

In Chainer v2, The attribute invoke_before_training of Extension is removed. Instead, the Extension.initialize method is added. This method is called by Trainer.run before entering the training loop.

In Chainer v1, the extension is just called before entering the training loop when invoke_before_training is True. If you have a custom extension that has invoke_before_training=True**, you have to update the code.** What you have to do is to remove the `invoke_before_training` flag and override initialize() method. If you are using the make_extension() decorator, you can set the initialize function by passing the initializer argument to make_extension().

The dump_graph extension dumps the valid graph only at its first invocation

In Chainer v2, the dump_graph() extension dumps the valid computational graph only at its first invocation. If you want to dump the graph more than once, you have to fix the code. The easiest fix is setting the chainer.config.keep_graph_on_report flag to True. Note that this fix will cancel the improvement on the memory consumption made in Chainer v2. More memory-efficient fix is to dump the graph without using an extension, e.g. by customizing the loss function or the updater.

Here is the background of this change. In Chainer v2, the Reporter copies reported variables with purging the computational graph by default. On the other hand, the dump_graph() extension requires the computational graph reachable from the reported variable. In order to make the graph available, the dump_graph() extension turns on the chainer.config.keep_graph_on_report flag at its initializer (i.e., it turns on the graph before entering
Since we also wanted to achieve the memory efficiency, the `dump_graph()` extension **turns off the flag after dumping the graph at its first invocation** (strictly speaking, it recovers the original value). As a result, the computational graph is not available from the second invocation.

Since the `dump_graph()` recovers the original flag value at its invocation, you can keep the graph dumped more than once by changing the original flag value.

### Reporter

**When a variable is reported, the variable is copied with the graph purged**

In Chainer v2, when a `Variable` object is reported using `report()` function (or directly using `Reporter`), a copy of the variable is made without preserving the computational graph. **If your code depends on the reachability of the computational graph from the reported variable, you have to update your code.** The easiest way to update your code is setting `chainer.config.keep_graph_on_report` to True, then Chainer will keep the computational graph reachable from the reported variable.

The possible examples that are affected by this change are as follows (not exhaustive).

- A custom extension that runs backprop from a reported variable. It is definitely an example of assuming the reachability of the computational graph from the reported variable.
- An extension that visualizes the computational graph from a reported variable. If you are writing such an extension by yourself, you have to turn on the `keep_graph_on_report` flag. The `dump_graph()` extension is another example, for which see the above item for the details.

This change is made for the memory performance reason; with this change, the memory used by the computational graph for training is immediately released before invoking extensions. Therefore, **changing the behavior by overwriting chainer.config.keep_graph_on_report may increase the memory consumption.** It may cause an out-of-memory error if the computational graph of the loss function consumes almost all the memory available in your environment and there is an extension that uses a certain amount of memory (e.g. `Evaluator`).

### Other utilities

**Some obsolete classes and functions are removed**

The following classes and functions are removed in Chainer v2.

- `chainer.Flag`
- `chainer.FunctionSet` (Use `Chain` or `ChainList` instead)
- `chainer.cuda.init` (It did nothing except for calling `check_cuda_available()`)
- `chainer.cuda.empty` (Use `cupy.empty()`)
- `chainer.cuda.empty_like` (Use `cupy.empty_like()`)
- `chainer.cuda.full` (Use `cupy.full()`)
- `chainer.cuda.full_like` (Use `cupy.full_like()`)
- `chainer.cuda.ones` (Use `cupy.ones()`)
- `chainer.cuda.ones_like` (Use `cupy.ones_like()`)
- `chainer.cuda.zeros` (Use `cupy.zeros()`)

---

4.9. Reporter 455
• `chainer.cuda.zeros_like(Use cupy.zeros_like())`
CHAPTER 5

Contribution Guide

This is a guide for all contributions to Chainer. The development of Chainer is running on the official repository at GitHub. Anyone that wants to register an issue or to send a pull request should read through this document.

Note: Many points of this document are updated at v2. We strongly recommend all contributors of v1 to read through the document again.

Classification of Contributions

There are several ways to contribute to Chainer community:

1. Registering an issue
2. Sending a pull request (PR)
3. Sending a question/reply to StackOverflow (with chainer tag) or Chainer User Group
4. Open-sourcing an external example
5. Writing a post about Chainer

This document mainly focuses on 1 and 2, though other contributions are also appreciated.

Development Cycle

This section explains the development process of Chainer. Before contributing to Chainer, it is strongly recommended to understand the development cycle.
Versioning

The versioning of Chainer follows PEP 440 and a part of Semantic versioning. The version number consists of three or four parts: X.Y.Zw where X denotes the major version, Y denotes the minor version, Z denotes the revision number, and the optional w denotes the pre-release suffix. While the major, minor, and revision numbers follow the rule of semantic versioning, the pre-release suffix follows PEP 440 so that the version string is much friendly with Python eco-system.

Note that a major update basically does not contain compatibility-breaking changes from the last release candidate (RC). This is not a strict rule, though; if there is a critical API bug that we have to fix for the major version, we may add breaking changes to the major version up.

As for the backward compatibility, see API Compatibility Policy.

Release Cycle

Starting from v2.0.0, we are developing two tracks of versions at the same time. The first one is the track of stable versions, which is a series of revision updates for the latest major version. The second one is the track of development versions, which is a series of pre-releases for the upcoming major version.

Consider that X.0.0 is the latest major version and Y.0.0, Z.0.0 are the succeeding major versions. Then, the timeline of the updates is depicted by the following table.

<table>
<thead>
<tr>
<th>Date</th>
<th>ver X</th>
<th>ver Y</th>
<th>ver Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 weeks</td>
<td>X.0rc1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4 weeks</td>
<td>X.0</td>
<td>Y.0a1</td>
<td>-</td>
</tr>
<tr>
<td>8 weeks</td>
<td>X.0.1</td>
<td>Y.0b1</td>
<td>-</td>
</tr>
<tr>
<td>12 weeks</td>
<td>X.0.2</td>
<td>Y.0rc1</td>
<td>-</td>
</tr>
<tr>
<td>16 weeks</td>
<td>-</td>
<td>Y.0</td>
<td>Z.0a1</td>
</tr>
</tbody>
</table>

The dates shown in the left-most column are relative to the release of X.0rc1. In particular, each revision release is made four weeks after the previous one of the same major version, and the pre-release of the upcoming major version is made at the same time.

Note that the development of X.0.x stops at X.0.2. During the parallel development of Y.0.0 and Z.0a1, the version Y is treated as an almost-stable version and Z is treated as a development version.

If there is a critical bug found in X.0.2 after stopping the development of version X, we may release a hot-fix for this version at any time.

Note: The release cycle of 2.0.x and 3.0.x are slightly different from this table because we do not have 3.0.0a1 at the timing of the release of 2.0.0. In this case, the releases of 3.0.x are shifted four weeks behind the usual timeline, that is, 3.0.0alpha will be released at the same time with 2.0.1.

As you can see in the above table, we basically do not have any minor releases from v2. All changes that add and/or modify APIs should be made by the pre-release updates.

We create a milestone for each upcoming release at GitHub. The GitHub milestone is basically used for collecting the issues and PRs resolved in the release.

Git Branches

The master branch is used to develop pre-release versions. It means that alpha, beta, and RC updates are developed at the master branch. This branch contains the most up-to-date source tree that includes features newly added after the latest major version.
The stable version is developed at the individual branch named as \texttt{vN} where “N” reflects the version number (we call it a \textit{versioned branch}). For example, v3.0.0, v3.0.1, and v3.0.2 will be developed at the \texttt{v3} branch.

\textbf{Notes for contributors:} When you send a pull request, you basically have to send it to the \texttt{master} branch. If the change can also be applied to the stable version, a core team member will apply the same change to the stable version so that the change is also included in the next revision update.

If the change is only applicable to the stable version and not to the \texttt{master} branch, please send it to the versioned branch. We basically only accept changes to the latest versioned branch (where the stable version is developed) unless the fix is critical.

\textit{Note: a change that can be applied to both branches should be sent to the \texttt{master} branch. Each release of the stable version is also merged to the development version so that the change is also reflected to the next major version.}

### Issues and Pull Requests

In this section, we explain how to file issues and send pull requests (PRs).

#### Issue/PR Labels

Issues and PRs are labeled by the following tags:

- **Bug**: bug reports (issues) and bug fixes (PRs)
- **Enhancement**: implementation improvements without breaking the interface
- **Feature**: feature requests (issues) and their implementations (PRs)
- **NoCompat**: disrupts backward compatibility
- **Test**: test fixes and updates
- **Document**: document fixes and improvements
- **Example**: fixes and improvements on the examples
- **Install**: fixes installation script
- **Contribution-Welcome**: issues that we request for contribution (only issues are categorized to this)
- **Other**: other issues and PRs

Multiple tags might be labeled to one issue/PR. \textbf{Note that revision releases cannot include PRs in Feature and NoCompat categories.}

#### How to File an Issue

On registering an issue, write precise explanations on how you want Chainer to be. Bug reports must include necessary and sufficient conditions to reproduce the bugs. Feature requests must include \textit{what} you want to do (and \textit{why} you want to do, if needed) with Chainer. You can contain your thoughts on \textit{how} to realize it into the feature requests, though \textit{what} part is most important for discussions.

\textbf{Warning}: If you have a question on usages of Chainer, it is highly recommended to send a post to StackOverflow or Chainer User Group instead of the issue tracker. The issue tracker is not a place to share knowledge on practices. We may suggest these places and immediately close how-to question issues.
How to Send a Pull Request

If you can write code to fix an issue, we encourage to send a PR.

First of all, before starting to write any code, do not forget to confirm the following points.

- Read through the Coding Guidelines and Testing Guidelines.
- Check the appropriate branch that you should send the PR following Git Branches. If you do not have any idea about selecting a branch, please choose the master branch.

In particular, check the branch before writing any code. The current source tree of the chosen branch is the starting point of your change.

After writing your code (including unit tests and hopefully documentations!), send a PR on GitHub. You have to write a precise explanation of what and how you fix; it is the first documentation of your code that developers read, which is a very important part of your PR.

Once you send a PR, it is automatically tested on Travis CI for Linux and Mac OS X, and on AppVeyor for Windows. Your PR needs to pass at least the test for Linux on Travis CI. After the automatic test passes, some of the core developers will start reviewing your code. Note that this automatic PR test only includes CPU tests.

Note: We are also running continuous integration with GPU tests for the master branch and the versioned branch of the latest major version. Since this service is currently running on our internal server, we do not use it for automatic PR tests to keep the server secure.

If you are planning to add a new feature or modify existing APIs, it is recommended to open an issue and discuss the design first. The design discussion needs lower cost for the core developers than code review. Following the consequences of the discussions, you can send a PR that is smoothly reviewed in a shorter time.

Even if your code is not complete, you can send a pull request as a work-in-progress PR by putting the [WIP] prefix to the PR title. If you write a precise explanation about the PR, core developers and other contributors can join the discussion about how to proceed the PR. WIP PR is also useful to have discussions based on a concrete code.

Coding Guidelines

We use PEP 8 and a part of OpenStack Style Guidelines related to general coding style as our basic style guidelines.

To check your code, use autopep8 and flake8 command installed by hacking package:

```bash
$ pip install autopep8 hacking
$ autopep8 --global-config .pep8 path/to/your/code.py
$ flake8 path/to/your/code.py
```

The autopep8 supports automatically correct Python code to conform to the PEP 8 style guide:

```bash
$ autopep8 --in-place --global-config .pep8 path/to/your/code.py
```

The flake8 command lets you know the part of your code not obeying our style guidelines. Before sending a pull request, be sure to check that your code passes the flake8 checking.

Note that flake8 command is not perfect. It does not check some of the style guidelines. Here is a (not-complete) list of the rules that flake8 cannot check.

- Relative imports are prohibited. [H304]
- Importing non-module symbols is prohibited.
• Import statements must be organized into three parts: standard libraries, third-party libraries, and internal imports. [H306]

In addition, we restrict the usage of shortcut symbols in our code base. They are symbols imported by packages and sub-packages of chainer. For example, chainer.Variable is a shortcut of chainer.variable.Variable. It is not allowed to use such shortcuts in the “chainer” library implementation. Note that you can still use them in tests and examples directories. Also note that you should use shortcut names of CuPy APIs in Chainer implementation.

Once you send a pull request, your coding style is automatically checked by Travis-CI. The reviewing process starts after the check passes.

### Testing Guidelines

Testing is one of the most important part of your code. You must test your code by unit tests following our testing guidelines. Note that we are using the nose package and the mock package for testing, so install nose and mock before writing your code:

```bash
$ pip install nose mock
```

### How to Run Tests

You can run unit tests simply by running nosetests command at the repository root:

```bash
$ nosetests
```

or specify the test script that you want to run:

```bash
$ nosetests path/to/your/test.py
```

You can also run all unit tests under a specified directory:

```bash
$ nosetests tests/chainer_tests/<directory name>
```

It requires CUDA by default. In order to run unit tests that do not require CUDA, pass --attr='!gpu' option to the nosetests command:

```bash
$ nosetests path/to/your/test.py --attr='!gpu'
```

Some GPU tests involve multiple GPUs. If you want to run GPU tests with insufficient number of GPUs, specify the number of available GPUs by --eval-attr='gpu<N' where N is a concrete integer. For example, if you have only one GPU, launch nosetests by the following command to skip multi-GPU tests:

```bash
$ nosetests path/to/gpu/test.py --eval-attr='gpu<2'
```

Some tests spend too much time. If you want to skip such tests, pass --attr='!slow' option to the nosetests command:

```bash
$ nosetests path/to/your/test.py --attr='!slow'
```

If you modify the code related to existing unit tests, you must run appropriate commands and confirm that the tests pass.
Test File and Directory Naming Conventions

Tests are put into the `tests/chainer_tests` directory. In order to enable test runner to find test scripts correctly, we are using special naming convention for the test subdirectories and the test scripts.

- The name of each subdirectory of `tests` must end with the `_tests` suffix.
- The name of each test script must start with the `test_` prefix.

When we write a test for a module, we use the appropriate path and file name for the test script whose correspondence to the tested module is clear. For example, if you want to write a test for a module `chainer.x.y.z`, the test script must be located at `tests/chainer_tests/x_tests/y_tests/test_z.py`.

How to Write Tests

There are many examples of unit tests under the `tests` directory, so reading some of them is a good and recommended way to learn how to write tests for Chainer. They simply use the `unittest` package of the standard library, while some tests are using utilities from `chainer.testing`.

Even if your patch includes GPU-related code, your tests should not fail without GPU capability. Test functions that require CUDA must be tagged by `chainer.testing.attr.gpu` decorator:

```python
import unittest
from chainer.testing import attr

class TestMyFunc(unittest.TestCase):
...
    @attr.gpu
    def test_my_gpu_func(self):
        ...
```

The functions tagged by the `gpu` decorator are skipped if `--attr='!gpu'` is given. We also have the `chainer.testing.attr.cudnn` decorator to let `nose` know that the test depends on cuDNN.

The test functions decorated by `gpu` must not depend on multiple GPUs. In order to write tests for multiple GPUs, use `chainer.testing.attr.multi_gpu()` decorator instead:

```python
import unittest
from chainer.testing import attr

class TestMyFunc(unittest.TestCase):
...
    @attr.multi_gpu(2)  # specify the number of required GPUs here
    def test_my_two_gpu_func(self):
        ...
```

If your test requires too much time, add `chainer.testing.attr.slow` decorator. The test functions decorated by `slow` are skipped if `--attr='!slow'` is given:

```python
import unittest
from chainer.testing import attr

class TestMyFunc(unittest.TestCase):
...
    @attr.slow
```

462 Chapter 5. Contribution Guide
```python
def test_my_slow_func(self):
    ...
```

**Note:** If you want to specify more than two attributes, separate them with a comma such as `--attr='!gpu,!slow'`. See detail in the document of nose.

Once you send a pull request, your code is automatically tested by Travis-CI with `--attr='!gpu,!slow'` option. Since Travis-CI does not support CUDA, we cannot check your CUDA-related code automatically. The reviewing process starts after the test passes. Note that reviewers will test your code without the option to check CUDA-related code.

**Note:** Some of numerically unstable tests might cause errors irrelevant to your changes. In such a case, we ignore the failures and go on to the review process, so do not worry about it!
This document explains the design policy on compatibilities of Chainer APIs. Development team should follow this policy on deciding to add, extend, and change APIs and their behaviors.

This document is written for both users and developers. Users can decide the level of dependencies on Chainer’s implementations in their codes based on this document. Developers should read through this document before creating pull requests that contain changes on the interface. Note that this document may contain ambiguities on the level of supported compatibilities.

**Targeted Versions**

This policy is applied to Chainer v2.0.0 and higher. Note that this policy is not applied to Chainer of lower versions. For older versions of Chainer, see the old version of API Compatibility Policy.

**Versioning and Backward Compatibility**

The versioning of Chainer follows the PEP 440 and a part of Semantic versioning. See Contribution Guide for details of versioning.

The backward compatibility is kept for revision updates, which are applied to the stable version. A major update from the latest release candidate basically keeps the backward compatibility, although it is not guaranteed. Any pre-releases may break the backward compatibility.

**Breaking the Compatibility**

We sometimes need to break the backward compatibility to improve the framework design and to support new kinds of machine learning methods. Such a change is only made into pre-releases (alpha, beta, and release candidate) and sometimes into the major update.
A change that breaks the compatibility affects user codes. We try to lower the cost of adapting your code to the newer version. The following list shows an example of what we can do to reduce the cost (Note: this is not a promise; what kind of actions we can take depends on the situation).

- When an argument is removed from an existing API, passing the argument to the updated API will emit an error with a special error message. The error message tells you how to fix your code.
- When a function or a class is removed, we make the current stable version emit a deprecation warning. Note that the deprecation warning is not printed by default in Python. You have to manually turn on the deprecation warning by `warnings.simplefilter('always', DeprecationWarning)`.
- When a definition of a link is changed, we try to enable it to deserialize a model dumped with an older version of Chainer. In most cases, we cannot guarantee that a model serialized with a newer version of Chainer is loadable by an older version of Chainer.

Note: Since Chainer v2, we have stopped adopting any solid processes to break backward compatibilities (e.g. a solid schedule for deprecating and removing a feature) in order to keep the development fast enough to support the cutting-edge research. It does not mean we stop taking care of maintainability of user codes. We are still paying much attention to not breaking user codes.

**Experimental APIs**

Thanks to many contributors, we have introduced many new features to Chainer.

However, we have sometimes released new features only to later notice that their APIs are not appropriate. In particular, we sometimes know that the API is likely to be modified in the near future because we do not have enough knowledge about how well the current design fits to the real usages. The objective of experimental APIs is to declare that the APIs are likely to be updated in the near future so that users can decide if they can(not) use them.

Any newly added API can be marked as experimental. Any API that is not experimental is called stable in this document.

Note: Undocumented behaviors are not considered as APIs, so they can be changed at any time (even in a revision update). The treatment of undocumented behaviors are described in Undocumented behaviors section.

When users use experimental APIs for the first time, warnings are raised once for each experimental API, unless users explicitly disable the emission of the warnings in advance.

See the document of `chainer.utils.experimental()` to know how developers mark APIs as experimental and how users enable or disable the warnings practically.

Note: It is up to developers if APIs should be annotated as experimental or not. We recommend to make the APIs experimental if they implement large modules or make a decision from several design choices.

**Supported Backward Compatibility**

This section defines backward compatibilities that revision updates must maintain.
**Documented Interface**

Chainer has the official API documentation. Many applications can be written based on the documented features. We support backward compatibilities of documented features. In other words, codes only based on the documented features run correctly with revision-updated versions.

Developers are encouraged to use apparent names for objects of implementation details. For example, attributes outside of the documented APIs should have one or more underscores at the prefix of their names.

**Note:** Although it is not stated as a rule, we also try to keep the compatibility for any interface that looks like a stable feature. For example, if the name of a symbol (function, class, method, attribute, etc.) is not prefixed by an underscore and the API is not experimental, the API should be kept over revision updates even if it is not documented.

**Undocumented behaviors**

Behaviors of Chainer implementation not stated in the documentation are undefined. Undocumented behaviors are not guaranteed to be stable between different revision versions.

Even revision updates may contain changes to undefined behaviors. One of the typical examples is a bug fix. Another example is an improvement on implementation, which may change the internal object structures not shown in the documentation. As a consequence, **even revision updates do not support compatibility of pickling, unless the full layout of pickled objects is clearly documented**.

**Documentation Error**

Compatibility is basically determined based on the documentation, although it sometimes contains errors. It may make the APIs confusing to assume the documentation always stronger than the implementations. We therefore may fix the documentation errors in any updates that may break the compatibility in regard to the documentation.

**Note:** Developers should not fix the documentation and implementation of the same functionality at the same time in revision updates as a “bug fix” unless the bug is so critical that no users are expected to be using the old version correctly.

**Object Attributes and Properties**

Object attributes and properties are sometimes replaced by each other. It does not break the user codes, except the codes depend on how the attributes and properties are implemented.

**Functions and Methods**

Methods may be replaced by callable attributes keeping the compatibility of parameters and return values. It does not break the user codes, except the codes depend on how the methods and callable attributes are implemented.

**Exceptions and Warnings**

The specifications of raising exceptions are considered as a part of standard backward compatibilities. No exception is raised in the future revision versions with correct usages that the documentation allows.
On the other hand, warnings may be added at any revision updates for any APIs. It means revision updates do not keep backward compatibility of warnings.

**Model Format Compatibility**

Links and chains serialized by official serializers that Chainer provides are correctly loaded with the future versions. They might not be correctly loaded with Chainer of the lower versions.

**Note:** Current serialization APIs do not support versioning. It prevents us from introducing changes in the layout of objects that support serialization. We are discussing versioning in serialization APIs.

**Installation Compatibility**

The installation process is another concern of compatibilities.

Any changes on the set of dependent libraries that force modifications on the existing environments should be done in pre-releases and major updates. Such changes include following cases:

- dropping supported versions of dependent libraries (e.g. dropping cuDNN v2)
- adding new mandatory dependencies (e.g. adding h5py to setup_requires)

**Note:** We sometimes have to narrow the supported versions due to bugs in the specific versions of libraries. In such a case, we may drop the support of those versions even in revision updates unless a workaround is found for the issue.
It takes too long time to compile a computational graph. Can I skip it?

Chainer does not compile computational graphs, so you cannot skip it, or, I mean, you have already skipped it :). It seems you have actually seen on-the-fly compilations of CUDA kernels. CuPy compiles kernels on demand to make kernels optimized to the number of dimensions and element types of input arguments. Pre-compilation is not available, because we have to compile an exponential number of kernels to support all CuPy functionalities. This restriction is unavoidable because Python cannot call CUDA/C++ template functions in generic way. Note that every framework using CUDA require compilation at some point; the difference between other statically-compiled frameworks (such as cutorch) and Chainer is whether a kernel is compiled at installation or at the first use.

These compilations should run only at the first use of the kernels. The compiled binaries are cached to the \$ (HOME) / .cupy/kernel_cache directory by default. If you see that compilations run every time you run the same script, then the caching is failed. Please check that the directory is kept as is between multiple executions of the script. If your home directory is not suited to caching the kernels (e.g. in case that it uses NFS), change the kernel caching directory by setting the CUPY_CACHE_DIR environment variable to an appropriate path. See cupy-overview for more details.

mnist example does not converge in CPU mode on Mac OS X

Many users reported that mnist example does not work correctly on Mac OS X. We are suspecting it is caused by vecLib, that is a default BLAS library installed on Mac OS X.

Note: Mac OS X is not officially supported. I mean it is not tested continuously on our test server.

We recommend to use other BLAS libraries such as OpenBLAS. We empirically found that it fixes this problem. It is necessary to reinstall NumPy to use replaced BLAS library. Here is an instruction to install NumPy with OpenBLAS using Homebrew.
If you want to install NumPy with pip, use site.cfg file.

You can check if NumPy uses OpenBLAS with `numpy.show_config` method. Check if `blas_opt_info` refers to `openblas`.

```python
>>> import numpy
>>> numpy.show_config()
lapack_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/opt/openblas/lib']
    define_macros = [('HAVE_CBLAS', None)]
    language = c
blas_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/opt/openblas/lib']
    define_macros = [('HAVE_CBLAS', None)]
    language = c
openblas_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/opt/openblas/lib']
    define_macros = [('HAVE_CBLAS', None)]
    language = c
openblas_lapack_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/opt/openblas/lib']
    define_macros = [('HAVE_CBLAS', None)]
    language = c
blas_mkl_info:
    NOT AVAILABLE
```

See detail about this problem in issue #704.
CHAPTER 8

Comparison with Other Frameworks

A table for quick comparison

This table compares Chainer with other popular deep learning frameworks. We hope it helps you to choose an appropriate framework for the demand.

Note: This chart may be out-dated, since the developers of Chainer do not perfectly follow the latest development status of each framework. Please report us if you find an out-dated cell. Requests for new comparison axes are also welcome.
<table>
<thead>
<tr>
<th></th>
<th>Chainer</th>
<th>Theano-based</th>
<th>Torch7</th>
<th>Caffe</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Specs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scripting</td>
<td>Python</td>
<td>Python</td>
<td>LuaJIT</td>
<td>Python</td>
</tr>
<tr>
<td>Net definition language</td>
<td>Python</td>
<td>Python</td>
<td>LuaJIT</td>
<td>Protocol Buffers</td>
</tr>
<tr>
<td>Define-by-Run scheme</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Array backend</td>
<td>NumPy</td>
<td>NumPy</td>
<td>Tensor</td>
<td></td>
</tr>
<tr>
<td>GPU Array backend</td>
<td>CuPy</td>
<td>CudaNdarray*</td>
<td>CudaTensor</td>
<td></td>
</tr>
<tr>
<td><strong>NNs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reverse-mode AD</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Basic RNN support</td>
<td>Y</td>
<td>Y</td>
<td>Y(\textit{nnx})</td>
<td>#2033</td>
</tr>
<tr>
<td>Variable-length loops</td>
<td>Y</td>
<td>Y(\textit{scan})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stateful RNNs(^2)</td>
<td>Y</td>
<td>Y</td>
<td>Y(^6)</td>
<td></td>
</tr>
<tr>
<td>Per-batch architectures</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Perf</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA support</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>cuDNN support</td>
<td>Y</td>
<td>Y</td>
<td>Y(\textit{cudnn.torch})</td>
<td>Y</td>
</tr>
<tr>
<td>FFT-based convolution</td>
<td>Y</td>
<td>Y</td>
<td>Y(\textit{fbcunn})</td>
<td>#544</td>
</tr>
<tr>
<td>CPU/GPU generic coding(^3)</td>
<td>Y</td>
<td>(^4)</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Multi GPU (data parallel)</td>
<td>Y</td>
<td>(^7)</td>
<td>Y(\textit{fbcunn})</td>
<td>Y</td>
</tr>
<tr>
<td>Multi GPU (model parallel)</td>
<td>Y</td>
<td>(^8)</td>
<td>Y(\textit{fbcunn})</td>
<td></td>
</tr>
<tr>
<td><strong>Misc</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type checking</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N/A</td>
</tr>
<tr>
<td>Model serialization</td>
<td>Y (\textit{pickle})</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Caffe reference model</td>
<td>Y(^5)</td>
<td>Y(\textit{loadcaffe})</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

**Benchmarks**

We are preparing for the benchmarks.

---

1. They are also developing \texttt{libgpuarray}
2. Stateful RNN is a type of RNN implementation that maintains states in the loops. It should enable us to use the states arbitrarily to update them.
3. Also available in the \texttt{Theano RNN package}
4. This row shows whether each array API supports unified codes for CPU and GPU.
5. The array backend of \texttt{Theano} does not have compatible interface with \texttt{NumPy}, though most users write code on \texttt{Theano} variables, which is generic for CPU and GPU.
6. Via \texttt{Platoon}
7. Experimental as May 2016
8. Depending on the frameworks.
Copyright (c) 2015 Preferred Infrastructure, Inc.
Copyright (c) 2015 Preferred Networks, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
CHAPTER 10

Indices and tables

- genindex
- modindex
- search


[Graves2006] Alex Graves, Santiago Fernandez, Faustino Gomez, Jurgen Schmidhuber, Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks

[Graves2012] Alex Graves, Supervised Sequence Labelling with Recurrent Neural Networks
C
chainer, 47
chainer.computational_graph, 437
chainer.cuda, 104
chainer.dataset, 83
chainer.datasets, 406
chainer.function_hooks, 398
chainer.functions, 120
chainer.initializers, 401
chainer.iterators, 417
chainer.links, 209
chainer.links.caffe, 433
chainer.serializers, 393
chainer.training, 88
chainer.training.extensions, 419
chainer.utils, 466
## Index

### Symbols

<table>
<thead>
<tr>
<th>Method</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__bool__()</code></td>
<td>chainer.Parameter method</td>
</tr>
<tr>
<td><code>__bool__()</code></td>
<td>chainer.Variable method</td>
</tr>
<tr>
<td><code>__bool__()</code></td>
<td>chainer.utils.type_check.Expr method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.AbstractSerializer method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.Deserializer method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.Function method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.Initializer method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.Serializer method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.initializers.Constant method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.initializers.GlorotNormal method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.initializers.GlorotUniform method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.initializers.HeNormal method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.initializers.HeUniform method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.initializers.Identity method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.initializers.LeCunUniform method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.initializers.Normal method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.initializers.Orthogonal method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.initializers.Uniform method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.BatchNormalization method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.Bias method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.Bilinear method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.BinaryHierarchicalSoftmax method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.BlackOut method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.CRF1d method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.Classifier method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.Convolution2D method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.ConvolutionND method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.Deconvolution2D method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.DeconvolutionND method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.DepthwiseConvolution2D method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.DilatedConvolution2D method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.EmbedID method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.GRU method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.GoogLeNet method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.Highway method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.Inception method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.InceptionBN method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.LSTM method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.LayerNormalization method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.Linear method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.MLPConvolution2D method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.Maxout method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.NStepBiGRU method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.NStepBiLSTM method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.NStepBiRNNReLU method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.NStepBiRNNtanh method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.NStepGRU method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.NStepLSTM method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.NStepRNNDense method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.NStepRNNTanh method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.NegativeSampling method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.PReLU method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.ResNet101Layers method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.ResNet152Layers method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.ResNet50Layers method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.Scale method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.SimplifiedDropconnect method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.StatefulGRU method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.StatefulPeepholeLSTM method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.StatelessLSTM method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.VGG16Layers method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.caffe.CaffeFunction method</td>
</tr>
<tr>
<td><code>__call__()</code></td>
<td>chainer.links.model.vision.resnet.ResNetLayers method</td>
</tr>
</tbody>
</table>
__call__() (chainer.optimizer.GradientClipping method), 79
__call__() (chainer.optimizer.GradientNoise method), 80
__call__() (chainer.optimizer.Lasso method), 79
__call__() (chainer.optimizer.WeightDecay method), 78
__call__() (chainer.serializers.DictionarySerializer method), 393
__call__() (chainer.serializers.HDF5Deserializer method), 396
__call__() (chainer.serializers.HDF5Serializer method), 395
__call__() (chainer.serializers.NpzDeserializer method), 394
__call__() (chainer.training.Extensions method), 97
__call__() (chainer.training.extensions.Evaluator method), 421
__call__() (chainer.training.extensions.ExponentialShift method), 423
__call__() (chainer.training.extensions.LinearShift method), 423
__call__() (chainer.training.extensions.LogReport method), 425
__call__() (chainer.training.extensions.PrintReport method), 428
__call__() (chainer.training.extensions.PlottReport method), 429
__call__() (chainer.training.extensions.PrintReport method), 430
__call__() (chainer.training.schedulers.IntervalTrigger method), 431
__call__() (chainer.training.schedulers.ManualScheduleTrigger method), 431
__call__() (chainer.training.schedulers.MaxValueTrigger method), 432
__call__() (chainer.training.schedulers.MinValueTrigger method), 433
__call__() (chainer[trainings.utils.type_check.Expr method), 115
__call__() (chainer.Parameter method), 52
__call__() (chainer.Serializer method), 81
__call__() (chainer.Variable method), 50
__call__() (chainer.utils.type_check.Expr method), 115
__call__() (chainer.Serializers method), 394
__call__() (chainer.variables.Variables method), 48
__call__() (chainer variable.Variable method), 48
__call__() (chainer.utils.type_check.Expr method), 115
__call__() (chainer.Parameter method), 52
__call__() (chainer.Serializer method), 81
__call__() (chainer.Variable method), 48
__call__() (chainer.dataset.DatasetMix method), 84
__call__() (chainer.datasets.DictDataset method), 406
__call__() (chainer.datasets.ImageDataset method), 412
__call__() (chainer.datasets.LabeledImageDataset method), 414
__call__() (chainer.datasets.SubDataset method), 408
__call__() (chainer.datasets.TransformDataset method), 411
__call__() (chainer.links.Classifier method), 348
__call__() (chainer.links.GRU method), 248
__call__() (chainer.links.GoogLeNet method), 357
__call__() (chainer.links.Highway method), 252
__call__() (chainer.links.Inception method), 255
__call__() (chainer.links.InceptionBN method), 258
__call__() (chainer.links.LSTM method), 267
__call__() (chainer.links.MLPConvolution2D method), 270
__call__() (chainer.links.Maxout method), 341
__call__() (chainer.links.NStepBiGRU method), 273
__call__() (chainer.links.NStepBiLSTM method), 277
__call__() (chainer.links.NStepBiRNNeLU method), 280
__call__() (chainer.links.NStepBiRNNTanh method), 283
__call__() (chainer.links.NStepBiGRU method), 287
__call__() (chainer.links.NStepBiLSTM method), 290
__call__() (chainer.links.NStepRNNeLU method), 293
__call__() (chainer.links.NStepRNNTanh method), 296
__call__() (chainer.links.ResNet101Layers method), 370
__call__() (chainer.links.ResNet152Layers method), 374
__call__() (chainer.links.ResNet50Layers method), 366
__call__() (chainer.links.Scale method), 300
Index
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_param() (chainer.links.caffe.CaffeFunction method)</td>
<td>434</td>
</tr>
<tr>
<td>add_param() (chainer.links.Classifier method)</td>
<td>348</td>
</tr>
<tr>
<td>add_param() (chainer.links.GoogLeNet method)</td>
<td>357</td>
</tr>
<tr>
<td>add_param() (chainer.links.GRU method)</td>
<td>248</td>
</tr>
<tr>
<td>add_param() (chainer.links.Highway method)</td>
<td>252</td>
</tr>
<tr>
<td>add_param() (chainer.links.Inception method)</td>
<td>255</td>
</tr>
<tr>
<td>add_param() (chainer.links.InceptionBN method)</td>
<td>258</td>
</tr>
<tr>
<td>add_param() (chainer.links.LSTM method)</td>
<td>267</td>
</tr>
<tr>
<td>add_param() (chainer.links.Maxout method)</td>
<td>341</td>
</tr>
<tr>
<td>add_param() (chainer.links.MLPConvolution2D method)</td>
<td>270</td>
</tr>
<tr>
<td>add_param() (chainer.links.model.vision.resnet.ResNetLayers method)</td>
<td>362</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepBiGRU method)</td>
<td>274</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepBiLSTM method)</td>
<td>277</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepBiRNNReLU method)</td>
<td>280</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepBiRNNTanh method)</td>
<td>283</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepGRU method)</td>
<td>287</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepLSTM method)</td>
<td>290</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepRNNReLU method)</td>
<td>293</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepRNNTanl method)</td>
<td>297</td>
</tr>
<tr>
<td>add_param() (chainer.links.ResNet101Layers method)</td>
<td>370</td>
</tr>
<tr>
<td>add_param() (chainer.links.ResNet152Layers method)</td>
<td>374</td>
</tr>
<tr>
<td>add_param() (chainer.links.ResNet50Layers method)</td>
<td>366</td>
</tr>
<tr>
<td>add_param() (chainer.links.Scale method)</td>
<td>300</td>
</tr>
<tr>
<td>add_param() (chainer.links.StatefulGRU method)</td>
<td>303</td>
</tr>
<tr>
<td>add_param() (chainer.links.StatefulPeepholeLSTM method)</td>
<td>307</td>
</tr>
<tr>
<td>add_param() (chainer.links.StatelessLSTM method)</td>
<td>310</td>
</tr>
<tr>
<td>add_param() (chainer.links.VGG16Layers method)</td>
<td>352</td>
</tr>
<tr>
<td>add_param() (chainer.Chain method)</td>
<td>68</td>
</tr>
<tr>
<td>add_param() (chainer.ChainList method)</td>
<td>70</td>
</tr>
<tr>
<td>add_param() (chainer.Link method)</td>
<td>63</td>
</tr>
<tr>
<td>add_param() (chainer.links.BatchNormalization method)</td>
<td>314</td>
</tr>
<tr>
<td>add_param() (chainer.links.Bias method)</td>
<td>210</td>
</tr>
<tr>
<td>add_param() (chainer.links.Bilinear method)</td>
<td>214</td>
</tr>
<tr>
<td>add_param() (chainer.links.BinaryHierarchicalSoftmax method)</td>
<td>322</td>
</tr>
<tr>
<td>add_param() (chainer.links.BlackOut method)</td>
<td>326</td>
</tr>
<tr>
<td>add_param() (chainer.links.caffe.CaffeFunction method)</td>
<td>435</td>
</tr>
<tr>
<td>add_param() (chainer.links.Confusion2D method)</td>
<td>348</td>
</tr>
<tr>
<td>add_param() (chainer.links.Convolution2D method)</td>
<td>219</td>
</tr>
<tr>
<td>add_param() (chainer.links.ConvolutionND method)</td>
<td>223</td>
</tr>
<tr>
<td>add_param() (chainer.links.CRF1d method)</td>
<td>329</td>
</tr>
<tr>
<td>add_param() (chainer.links.Deconvolution2D method)</td>
<td>228</td>
</tr>
<tr>
<td>add_param() (chainer.links.DeconvolutionND method)</td>
<td>232</td>
</tr>
<tr>
<td>add_param() (chainer.links.DepthwiseConvolution2D method)</td>
<td>236</td>
</tr>
<tr>
<td>add_param() (chainer.links.DilatedConvolution2D method)</td>
<td>236</td>
</tr>
<tr>
<td>add_param() (chainer.links.EmbedID method)</td>
<td>245</td>
</tr>
<tr>
<td>add_param() (chainer.links.GoogLeNet method)</td>
<td>357</td>
</tr>
<tr>
<td>add_param() (chainer.links.GRU method)</td>
<td>249</td>
</tr>
<tr>
<td>add_param() (chainer.links.Highway method)</td>
<td>252</td>
</tr>
<tr>
<td>add_param() (chainer.links.Inception method)</td>
<td>255</td>
</tr>
<tr>
<td>add_param() (chainer.links.InceptionBN method)</td>
<td>259</td>
</tr>
<tr>
<td>add_param() (chainer.links.LayerNormalization method)</td>
<td>318</td>
</tr>
<tr>
<td>add_param() (chainer.links.Linear method)</td>
<td>262</td>
</tr>
<tr>
<td>add_param() (chainer.links.LSTM method)</td>
<td>267</td>
</tr>
<tr>
<td>add_param() (chainer.links.Maxout method)</td>
<td>341</td>
</tr>
<tr>
<td>add_param() (chainer.links.MLPConvolution2D method)</td>
<td>270</td>
</tr>
<tr>
<td>add_param() (chainer.links.model.vision.resnet.ResNetLayers method)</td>
<td>362</td>
</tr>
<tr>
<td>add_param() (chainer.links.NegativeSampling method)</td>
<td>344</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepBiGRU method)</td>
<td>274</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepBiLSTM method)</td>
<td>277</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepBiRNNReLU method)</td>
<td>280</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepBiRNNTanh method)</td>
<td>284</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepGRU method)</td>
<td>287</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepLSTM method)</td>
<td>290</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepRNNReLU method)</td>
<td>293</td>
</tr>
<tr>
<td>add_param() (chainer.links.NStepRNNTanl method)</td>
<td>297</td>
</tr>
<tr>
<td>add_param() (chainer.links.PReLU method)</td>
<td>337</td>
</tr>
<tr>
<td>add_param() (chainer.links.ResNet101Layers method)</td>
<td>370</td>
</tr>
<tr>
<td>add_param() (chainer.links.ResNet152Layers method)</td>
<td>375</td>
</tr>
<tr>
<td>add_param() (chainer.links.ResNet50Layers method)</td>
<td>366</td>
</tr>
<tr>
<td>add_param() (chainer.links.Scale method)</td>
<td>300</td>
</tr>
<tr>
<td>add_param() (chainer.links.SimplifiedDropconnect method)</td>
<td>333</td>
</tr>
<tr>
<td>add_param() (chainer.links.StatefulGRU method)</td>
<td>303</td>
</tr>
<tr>
<td>add_param() (chainer.links.StatefulPeepholeLSTM method)</td>
<td>307</td>
</tr>
<tr>
<td>add_param() (chainer.links.StatelessLSTM method)</td>
<td>311</td>
</tr>
<tr>
<td>add_param() (chainer.links.VGG16Layers method)</td>
<td>352</td>
</tr>
<tr>
<td>add_persistent() (chainer.Chain method)</td>
<td>68</td>
</tr>
<tr>
<td>add_persistent() (chainer.ChainList method)</td>
<td>71</td>
</tr>
<tr>
<td>add_persistent() (chainer.Link method)</td>
<td>64</td>
</tr>
<tr>
<td>add_persistent() (chainer.links.BatchNormalization method)</td>
<td>484</td>
</tr>
</tbody>
</table>
add_persistent() (chainer.links.Bias method), 211
add_persistent() (chainer.links.Bilinear method), 215
add_persistent() (chainer.links.BinaryHierarchicalSoftmax method), 323
add_persistent() (chainer.links.BlackOut method), 326
add_persistent() (chainer.links.Bilinear method), 215
add_persistent() (chainer.links.BinaryHierarchicalSoftmax method), 323
add_persistent() (chainer.links.BlackOut method), 326
add_persistent() (chainer.links.Convolution2D method), 220
add_persistent() (chainer.links.ConvolutionND method), 224
add_persistent() (chainer.links.CRF1d method), 330
add_persistent() (chainer.links.Deconvolution2D method), 229
add_persistent() (chainer.links.DeconvolutionND method), 224
add_persistent() (chainer.links.EmbedID method), 245
add_persistent() (chainer.links.GoogLeNet method), 357
add_persistent() (chainer.links.GRU method), 249
add_persistent() (chainer.links.Highway method), 252
add_persistent() (chainer.links.Inception method), 256
add_persistent() (chainer.links.InceptionBN method), 259
add_persistent() (chainer.links.LayerNormalization method), 319
add_persistent() (chainer.links.Linear method), 263
add_persistent() (chainer.links.LSTM method), 267
add_persistent() (chainer.links.Maxout method), 342
add_persistent() (chainer.links.MLPConvolution2D method), 271
add_persistent() (chainer.links.model.vision.resnet.ResNetLayers method), 362
add_persistent() (chainer.links.NegativeSampling method), 345
add_persistent() (chainer.links.NStepBiGRU method), 274
add_persistent() (chainer.links.NStepBiLSTM method), 277
add_persistent() (chainer.links.NStepBiRNNReLU method), 281
add_persistent() (chainer.links.NStepBiRNNTanh method), 284
add_persistent() (chainer.links.NStepGRU method), 287
add_persistent() (chainer.links.NStepLSTM method), 291
add_persistent() (chainer.links.NStepRNNReLU method), 294
add_persistent() (chainer.links.NStepRNNTanh method), 297
add_persistent() (chainer.links.PReLU method), 338
add_persistent() (chainer.links.ResNet101Layers method), 371
add_persistent() (chainer.links.ResNet152Layers method), 375
add_persistent() (chainer.links.ResNet50Layers method), 367
add_persistent() (chainer.links.Scale method), 300
add_persistent() (chainer.links.SimplifiedDropconnect method), 334
add_persistent() (chainer.links.StatefulGRU method), 304
add_persistent() (chainer.links.StatefulPeepholeLSTM method), 307
add_persistent() (chainer.links.StatelessLSTM method), 311
add_persistent() (chainer.links.VGG16Layers method), 353
addgrad() (chainer.Parameter method), 52
addgrad() (chainer.Variable method), 48
addgrads() (chainer.Chain method), 68
addgrads() (chainer.ChainList method), 71
addgrads() (chainer.Link method), 64
addgrads() (chainer.links.BatchNormalization method), 315
addgrads() (chainer.links.Bias method), 211
addgrads() (chainer.links.Bilinear method), 215
addgrads() (chainer.links.BinaryHierarchicalSoftmax method), 323
addgrads() (chainer.links.Bilinear method), 215
addgrads() (chainer.links.Deconvolution2D method), 229
addgrads() (chainer.links.DeconvolutionND method), 224
addgrads() (chainer.links.EmbedID method), 245
addgrads() (chainer.links.GoogLeNet method), 357
addgrads() (chainer.links.GRU method), 249
addgrads() (chainer.links.Highway method), 252
addgrads() (chainer.links.Inception method), 256
addgrads() (chainer.links.InceptionBN method), 259
addgrads() (chainer.links.LayerNormalization method), 319
addgrads() (chainer.links.Linear method), 263
addgrads() (chainer.links.LSTM method), 267
addgrads() (chainer.links.Maxout method), 342
addgrads() (chainer.links.MLPConvolution2D method), 271
addgrads() (chainer.links.model.vision.resnet.ResNetLayers method), 362
addgrads() (chainer.links.NegativeSampling method), 345
addgrads() (chainer.links.NStepBiGRU method), 274
addgrads() (chainer.links.NStepBiLSTM method), 277
addgrads() (chainer.links.NStepBiRNNReLU method), 281
addgrads() (chainer.links.NStepBiRNNTanh method), 284
addgrads() (chainer.links.NStepGRU method), 287
addgrads() (chainer.links.NStepLSTM method), 291
addgrads() (chainer.links.NStepRNNReLU method), 294
addgrads() (chainer.links.NStepRNNTanh method), 297
addgrads() (chainer.links.PReLU method), 338
addgrads() (chainer.links.ResNet101Layers method), 371
addgrads() (chainer.links.ResNet152Layers method), 375
addgrads() (chainer.links.ResNet50Layers method), 367
addgrads() (chainer.links.Scale method), 300
addgrads() (chainer.links.SimplifiedDropconnect method), 334
addgrads() (chainer.links.StatefulGRU method), 304
addgrads() (chainer.links.StatefulPeepholeLSTM method), 307
addgrads() (chainer.links.StatelessLSTM method), 311
addgrads() (chainer.links.VGG16Layers method), 353
addgrads() (chainer.Parameter method), 52
addgrads() (chainer.Variable method), 48
addgrads() (chainer.Chain method), 68
addgrads() (chainer.ChainList method), 71
addgrads() (chainer.Link method), 64
addgrads() (chainer.links.BatchNormalization method), 315
addgrads() (chainer.links.Bias method), 211
addgrads() (chainer.links.Bilinear method), 215
addgrads() (chainer.links.BinaryHierarchicalSoftmax method), 323
addgrads() (chainer.links.Bilinear method), 215
addgrads() (chainer.links.Deconvolution2D method), 229
addgrads() (chainer.links.DeconvolutionND method), 224
addgrads() (chainer.links.EmbedID method), 245
addgrads() (chainer.links.GoogLeNet method), 357
addgrads() (chainer.links.GRU method), 249
addgrads() (chainer.links.Highway method), 252
addgrads() (chainer.links.Inception method), 256
addgrads() (chainer.links.InceptionBN method), 259
addgrads() (chainer.links.LayerNormalization method), 319
addgrads() (chainer.links.Linear method), 263
addgrads() (chainer.links.LSTM method), 267
addgrads() (chainer.links.Maxout method), 342
addgrads() (chainer.links.MLPConvolution2D method), 271
addgrads() (chainer.links.model.vision.resnet.ResNetLayers method), 362
addgrads() (chainer.links.NegativeSampling method), 345
addgrads() (chainer.links.NStepBiGRU method), 274
addgrads() (chainer.links.NStepBiLSTM method), 277
addgrads() (chainer.links.NStepBiRNNReLU method), 281
addgrads() (chainer.links.NStepBiRNNTanh method), 284
addgrads() (chainer.links.NStepGRU method), 287
addgrads() (chainer.links.NStepLSTM method), 291
addgrads() (chainer.links.NStepRNNReLU method), 294
addgrads() (chainer.links.NStepRNNTanh method), 297
addgrads() (chainer.links.PReLU method), 338
addgrads() (chainer.links.NegativeSampling method), 345
addgrads() (chainer.links.NStepBiGRU method), 274
addgrads() (chainer.links.NStepBiLSTM method), 278
addgrads() (chainer.links.NStepBiRNNReLU method),
addgrads() (chainer.links.NStepBiRNNTanh method),
addgrads() (chainer.links.NStepGRU method), 287
addgrads() (chainer.links.NStepLSTM method), 291
addgrads() (chainer.links.NStepRNNReLU method), 294
addgrads() (chainer.links.NStepRNNTanh method), 297
addgrads() (chainer.links.PReLU method), 338
addgrads() (chainer.links.ResNet101Layers method), 371
addgrads() (chainer.links.ResNet152Layers method), 375
addgrads() (chainer.links.ResNet50Layers method), 367
addgrads() (chainer.links.Scale method), 301
addgrads() (chainer.links.SimplifiedDropconnect method), 334
addgrads() (chainer.links.StatefulGRU method), 304
addgrads() (chainer.links.StatefulPeepholeLSTM method), 307
addgrads() (chainer.links.StatelessLSTM method), 311
addgrads() (chainer.links.VGG16Layers method), 353
alpha (chainer.optimizers.Adam attribute), 383
alpha (chainer.optimizers.RMSprop attribute), 388
alpha (chainer.optimizers.RMSpropGraves attribute), 389
append() (chainer.ChainList method), 71
append() (chainer.Function method), 58
append() (chainer.Parameter method), 52
append() (chainer.Variable method), 49
backward_cpu() (chainer.Function method), 58
backward_postprocess() (chainer.function.FunctionHook method), 398
backward_postprocess() (chainer.function_hooks.PrintHook method), 399
backward_postprocess() (chainer.function_hooks.TimerHook method), 400
backward_preprocess() (chainer.function.FunctionHook method), 398
backward_preprocess() (chainer.function_hooks.PrintHook method), 399
backward_preprocess() (chainer.function_hooks.TimerHook method), 400
batch_inv() (in module chainer.functions), 189
batch_l2_norm_squared() (in module chainer.functions), 190
batch_matmul() (in module chainer.functions), 190
batch_normalization() (in module chainer.functions), 200
BatchNormalization (class in chainer.links), 213
BernoulliNLL (in module chainer.functions), 175
beta1 (chainer.optimizers.Adam attribute), 383
beta2 (chainer.optimizers.Adam attribute), 383
Bias (class in chainer.links), 210
bias() (in module chainer.functions), 190
Bilinear (class in chainer.links), 213
bilinear() (in module chainer.functions), 153
binary_accuracy() (in module chainer.functions), 188
BinaryHierarchicalSoftmax (class in chainer.links), 173
black_out() (in module chainer.functions), 176
BlackOut (class in chainer.links), 325
broadcast() (in module chainer.functions), 132
broadcast() (in module chainer.functions), 132
build_computational_graph() (in module chainer.computational_graph), 438
C

cache_or_load_file() (in module chainer.dataset), 87
cached_download() (in module chainer.dataset), 87
CaffeFunction (class in chainer.links.caffe), 433
call_for_each_param (chainer.optimizer.GradientNoise attribute), 80
call_for_each_param (chainer.optimizer.Lasso attribute), 79
call_for_each_param (chainer.optimizer.WeightDecay attribute), 78
call_hooks() (chainer.GradientMethod method), 77
call_hooks() (chainer.Optimizer method), 75
call_hooks() (chainer.optimizers.AdaDelta method), 379
call_hooks() (chainer.optimizers.AdaGrad method), 380
call_hooks() (chainer.optimizers.Adam method), 382
call_hooks() (chainer.optimizers.MomentumSGD method), 384
call_hooks() (chainer.optimizers.NesterovAG method), 385
call_hooks() (chainer.optimizers.RMSprop method), 387
call_hooks() (chainer.optimizers.RMSpropGraves method), 388
call_hooks() (chainer.optimizers.SGD method), 390
call_hooks() (chainer.optimizers.SMORMS3 method), 392
cast() (in module chainer.functions), 133
ceil() (in module chainer.functions), 191
Chain (class in chainer), 66
chainer (module), 47
chainer.computational_graph (module), 437
chainer.cuda (module), 104
chainer.dataset (module), 83
chainer.datasets (module), 406
chainer.function_hooks (module), 398
chainer.functions (module), 120
chainer.initializers (module), 401
chainer.iterators (module), 417
chainer.links (module), 209
chainer.links.caffe (module), 433
chainer.serializers (module), 393
chainer.training (module), 419
chainer.utils (module), 466
ChainList (class in chainer), 70
check_backward() (in module chainer.gradient_check), 116
check_type_forward() (chainer.Function method), 59
childen() (chainer.links.BinaryHierarchicalSoftmax method), 323
children() (chainer.links.BlackOut method), 327
children() (chainer.links.caffe.CaffeFunction method), 435
children() (chainer.links.Classifier method), 349
children() (chainer.links.Convolution2D method), 220
children() (chainer.links.ConvolutionND method), 224
children() (chainer.links.CRF1d method), 330
children() (chainer.links.Deconvolution2D method), 229
children() (chainer.links.DeconvolutionND method), 233
children() (chainer.links.DepthwiseConvolution2D method), 237
children() (chainer.links.DilatedConvolution2D method), 242
children() (chainer.links.EmbedID method), 245
children() (chainer.links.GoogLeNet method), 358
children() (chainer.links.GRU method), 249
children() (chainer.links.Highway method), 253
children() (chainer.links.Inception method), 256
children() (chainer.links.InceptionBN method), 259
children() (chainer.links.LayerNormalization method), 319
children() (chainer.links.Linear method), 263
children() (chainer.links.LSTM method), 268
children() (chainer.links.Maxout method), 342
children() (chainer.links.MLPConvolution2D method), 271
children() (chainer.links.model.vision.resnet.ResNetLayers method), 363
children() (chainer.links.NegativeSampling method), 345
children() (chainer.links.NStepBiGRU method), 274
children() (chainer.links.NStepBiLSTM method), 278
children() (chainer.links.NStepBiRNNReLU method), 281
children() (chainer.links.NStepBiRNNTanH method), 284
children() (chainer.links.NStepGRU method), 288
children() (chainer.links.NStepLSTM method), 291
children() (chainer.links.NStepRNNReLU method), 294
children() (chainer.links.NStepRNNTanH method), 298
children() (chainer.links.PRReLU method), 338
children() (chainer.links.ResNet152Layers method), 375
children() (chainer.links.ResNet50Layers method), 367
children() (chainer.links.Scale method), 301
children() (chainer.links.SimplifiedDropconnect method), 334
children() (chainer.links.StatefulGRU method), 304
children() (chainer.links.StatefulPeepholeLSTM method), 308
children() (chainer.links.StatelessLSTM method), 311
children() (chainer.links.VGG16Layers method), 353
classification_summary() (in module chainer.functions), 174

Index 487
copyparams() (chainer.links.NStepBiRNNReLU method), 281
copyparams() (chainer.links.NStepBiRNNNTanh method), 284
copyparams() (chainer.links.NStepGRU method), 288
copyparams() (chainer.links.NStepLSTM method), 291
copyparams() (chainer.links.NStepRNNTanh method), 298
copyparams() (chainer.links.PReLU method), 338
copyparams() (chainer.links.ResNet101Layers method), 371
copyparams() (chainer.links.ResNet152Layers method), 376
copyparams() (chainer.links.ResNet50Layers method), 367
copyparams() (chainer.links.Scale method), 301
copyparams() (chainer.links.SimplifiedDropconnect method), 335
copyparams() (chainer.links.StatefulGRU method), 304
copyparams() (chainer.links.StatefulPeepholeLSTM method), 308
copyparams() (chainer.links.StatelessLSTM method), 312
copyparams() (chainer.links.VGG16Layers method), 353
cos() (in module chainer.functions), 191
cosh() (in module chainer.functions), 192
count() (chainer.utils.type_check.TypeInfoTuple method), 116
cre_damage() (chainer.links.BinaryHierarchicalSoftmax method), 324
creat_update_rule() (chainer.GradientMethod method), 77
create_update_rule() (chainer.optimizers.AdaDelta method), 379
create_update_rule() (chainer.optimizers.AdaGrad method), 380
create_update_rule() (chainer.optimizers.Adam method), 382
create_update_rule() (chainer.optimizers.MomentumSGD method), 384
create_update_rule() (chainer.optimizers.NesterovAG method), 385
create_update_rule() (chainer.optimizers.RMSprop method), 387
create_update_rule() (chainer.optimizers.RMSpropGraves method), 389
create_update_rule() (chainer.optimizers.SGD method), 390
create_update_rule() (chainer.optimizers.SMORMS3 method), 392
creator (chainer.Parameter attribute), 54
creator (chainer.Variable attribute), 50
creator (chainer.variable.VariableNode attribute), 56
crelu() (in module chainer.functions), 122
CRF1d (chainer.links), 329
crf1d() (in module chainer.functions), 178
cross_covariance() (in module chainer.functions), 180

D
data (chainer.Parameter attribute), 54
data (chainer.Variable attribute), 51
data (chainer.variable.VariableNode attribute), 56
DatasetMix (class in chainer.dataset), 83
debug_print() (chainer.Parameter method), 53
debug_print() (chainer.Variable method), 49
DebugMode (class in chainer), 100
Deconvolution2D (class in chainer.links), 226
dehconvolution_2d() (in module chainer.functions), 157
dehconvolution_2d() (in module chainer.functions), 159
DeconvolutionND (class in chainer.links), 231
default_name (chainer.training.extensions.Evaluator attribute), 98
default_name (chainer.training.extensions.ExponentialShift attribute), 423
default_name (chainer.training.extensions.LinearShift attribute), 424
default_name (chainer.training.extensions.LogReport attribute), 425
default_name (chainer.training.extensions.PrintReport attribute), 428
default_name (chainer.training.extensions.PrintReport attribute), 429
default_name (chainer.training.extensions.ProgressBar attribute), 430
delete_hook() (chainer.Function method), 59
dehconvolution_2d() (in module chainer.functions), 135
DepthwiseConvolution2D (in module chainer.functions), 161
DepthwiseConvolution2D (class in chainer.links), 235
Deserializer (class in chainer), 82
DictDataset (class in chainer.datasets), 406
DictionarySerializer (class in chainer.serializers), 393
DictSummary (class in chainer), 112
dilated_convolution_2d() (in module chainer.functions), 162
DilatedConvolution2D (class in chainer.links), 239
disable_update() (chainer.Chain method), 69
disable_update() (chainer.ChainList method), 71
disable_update() (chainer.Link method), 64
disable_update() (chainer.links.BatchNormalization method), 316
disable_update() (chainer.links.Bias method), 211
disable_update() (chainer.links.Bilinear method), 215
disable_update() (chainer.links.BinaryHierarchicalSoftmax method), 324
disable_update() (chainer.links.BlackOut method), 327
disable_update() (chainer.links.caffe.CaffeFunction method), 436
disable_update() (chainer.links.Classifier method), 349
disable_update() (chainer.links.Convolution2D method), 220
disable_update() (chainer.links.ConvolutionND method), 224
disable_update() (chainer.links.CRF1d method), 331
disable_update() (chainer.links.Deconvolution2D method), 229
disable_update() (chainer.links.DeconvolutionND method), 233
disable_update() (chainer.links.DepthwiseConvolution2D method), 237
disable_update() (chainer.links.DilatedConvolution2D method), 242
disable_update() (chainer.links.EmbedID method), 246
disable_update() (chainer.links.GoogLeNet method), 358
disable_update() (chainer.links.GRU method), 249
disable_update() (chainer.links.Highway method), 253
disable_update() (chainer.links.Inception method), 256
disable_update() (chainer.links.InceptionBN method), 259
disable_update() (chainer.links.LayerNormalization method), 320
disable_update() (chainer.links.Linear method), 264
disable_update() (chainer.links.LSTM method), 268
disable_update() (chainer.links.Maxout method), 342
disable_update() (chainer.links.MLPConvolution2D method), 271
disable_update() (chainer.links.model.vision.resnet.ResNetLayers method), 363
disable_update() (chainer.links.NegativeSampling method), 345
disable_update() (chainer.links.NStepBiGRU method), 274
disable_update() (chainer.links.NStepBiLSTM method), 278
disable_update() (chainer.links.NStepBiRNNReLU method), 281
disable_update() (chainer.links.NStepBiRNNTanh method), 284
disable_update() (chainer.links.NStepGRU method), 288
disable_update() (chainer.links.NStepLSTM method), 291
disable_update() (chainer.links.NStepRNNReLU method), 294
disable_update() (chainer.links.NStepRNNNTanh method), 298
disable_update() (chainer.links.PReLU method), 338
disable_update() (chainer.links.ResNet101Layers method), 371
disable_update() (chainer.links.ResNet152Layers method), 376
disable_update() (chainer.links.ResNet50Layers method), 367
disable_update() (chainer.links.Scale method), 301
disable_update() (chainer.links.SimplifiedDropconnect method), 335
disable_update() (chainer.links.StatefulGRU method), 304
disable_update() (chainer.links.StatefulPeepholeLSTM method), 308
disable_update() (chainer.links.StatelessLSTM method), 312
disable_update() (chainer.links.VGG16Layers method), 353
dropout() (in module chainer.functions), 198
dstack() (in module chainer.functions), 136
dtype (chainer.Parameter attribute), 54
dtype (chainer.Variable attribute), 51
dump() (chainer.computational_graph.ComputationalGraph method), 440
dump_graph() (in module chainer.training.extensions), 419

e
elapsed_time (chainer.training.Trainer attribute), 91
elementwise() (in module chainer.cuda), 107
elu() (in module chainer.functions), 122
embed_id() (in module chainer.functions), 163
EmbedID (class in chainer.links), 244
enable_update() (chainer.Chain method), 69
enable_update() (chainer.ChainList method), 71
enable_update() (chainer.Link method), 64
enable_update() (chainer.links.BatchNormalization method), 316
enable_update() (chainer.links.Bias method), 212
enable_update() (chainer.links.Bilinear method), 216
enable_update() (chainer.links.BinaryHierarchicalSoftmax method), 324
enable_update() (chainer.links.BlackOut method), 327
enable_update() (chainer.links.caffe.CaffeFunction method), 436
enable_update() (chainer.links.Classifier method), 349
enable_update() (chainer.links.Convolution2D method), 220
enable_update() (chainer.links.ConvolutionND method), 224
enable_update() (chainer.links.CRF1d method), 331
enable_update() (chainer.links.Deconvolution2D method), 229
enable_update() (chainer.links.DeconvolutionND method), 233
enable_update() (chainer.links.DepthwiseConvolution2D method), 237
enable_update() (chainer.links.DilatedConvolution2D method), 242
enable_update() (chainer.links.EmbedID method), 246
enable_update() (chainer.links.GoogLeNet method), 358
enable_update() (chainer.links.GRU method), 249
enable_update() (chainer.links.Highway method), 253
enable_update() (chainer.links.Inception method), 256
enable_update() (chainer.links.InceptionBN method), 260
enable_update() (chainer.links.LayerNormalization method), 320
enable_update() (chainer.links.Linear method), 264
enable_update() (chainer.links.LSTM method), 268
enable_update() (chainer.links.Maxout method), 342
enable_update() (chainer.links.MLPConvolution2D method), 271
enable_update() (chainer.links.NegativeSampling method), 345
enable_update() (chainer.links.NStepBiGRU method), 275
enable_update() (chainer.links.NStepBiLSTM method), 278
enable_update() (chainer.links.NStepBiRNNReLU method), 281
enable_update() (chainer.links.NStepBiRNNTanh method), 285
enable_update() (chainer.links.NStepGRU method), 288
enable_update() (chainer.links.NStepLSTM method), 291
enable_update() (chainer.links.NStepRNNReLU method), 294
enable_update() (chainer.links.NStepRNNTanh method), 298
enable_update() (chainer.links.PReLU method), 338
enable_update() (chainer.links.ResNet101Layers method), 372
enable_update() (chainer.links.ResNet152Layers method), 376
enable_update() (chainer.links.ResNet50Layers method), 367
enable_update() (chainer.links.Scale method), 301
enable_update() (chainer.links.SimplifiedDropconnect method), 335
enable_update() (chainer.links.StatefulGRU method), 304
enable_update() (chainer.links.StatefulPeepholeLSTM method), 308
enable_update() (chainer.links.StatelessLSTM method), 312
enable_update() (chainer.links.VGG16Layers method), 353
epoch (chainer.training.ParallelUpdater attribute), 492
epoch (chainer.training.StandardUpdater attribute), 95
eval() (chainer.utils.type_check.Expr method), 115
evaluate() (chainer.training.extensions.Evaluator method), 421
Evaluator (class in chainer.training.extensions), 420
exp() (in module chainer.functions), 192
expand_dims() (in module chainer.functions), 137
expect() (in module chainer.utils.type_check), 115
ExperimentalShift (class in chainer.training.extensions), 422
Expr (class in chainer.utils.type_check), 114
finalize() (chainer.dataset.Iterator method), 85
finalize() (chainer.iterators.Iterator method), 85
finalize() (chainer.iterators.SerialIterator method), 301
finalize() (chainer.iterators.MultiprocessIterator method), 418
finalize() (chainer.iterators.SerialIterator method), 417
finalize() (chainer.training.Extension method), 97
finalize() (chainer.training.extensions.Evaluator method), 421
finalize() (chainer.training.extensions.ExponentialShift method), 423
finalize() (chainer.training.extensions.LinearShift method), 423
finalize() (chainer.training.extensions.LogReport method), 425
finalize() (chainer.training.extensions.PrintReport method), 428
finalize() (chainer.training.extensions.PlotReport method), 429

492 Index
GlorotUniform (class in chainer.initializers), 405
GoogLeNet (class in chainer.links), 356
grad (chainer.Parameter attribute), 54
grad (chainer.Variable attribute), 51
grad (chainer.variable.VariableNode attribute), 56
GradientClipping (class in chainer.optimizer), 79
GradientMethod (class in chainer), 76
GradientNoise (class in chainer.optimizer), 79
GRU (class in chainer.links), 248
hard_sigmoid() (in module chainer.functions), 123
HDF5Deserializer (class in chainer.serializers), 395
HDF5Serializer (class in chainer.serializers), 395
HeNormal (class in chainer.initializers), 403
HeUniform (class in chainer.initializers), 405
Highway (class in chainer.links), 251
hinge() (in module chainer.functions), 182
hstack() (in module chainer.functions), 139
huber_loss() (in module chainer.functions), 182
Identity (class in chainer.initializers), 401
Identity (in module chainer.functions), 192
ignore_label (chainer.links.EmbedID attribute), 248
im2col() (in module chainer.functions), 140
ImageDataset (class in chainer.datasets), 412
Inception (class in chainer.links), 254
InceptionBN (class in chainer.links), 258
index() (chainer.utils.type_check.TypeInfoTuple method), 116
init_hx() (chainer.links.NStepBiGRU method), 275
init_hx() (chainer.links.NStepBiLSTM method), 278
init_hx() (chainer.links.NStepBiRNNReLU method), 281
init_hx() (chainer.links.NStepBiRNNTanh method), 285
init_hx() (chainer.links.NStepGRU method), 288
init_hx() (chainer.links.NStepLSTM method), 291
init_hx() (chainer.links.NStepRNNReLU method), 295
init_hx() (chainer.links.NStepRNNTanh method), 298
init_scope() (chainer.Chain method), 69
init_scope() (chainer.ChainList method), 72
init_scope() (chainer.Link method), 64
init_scope() (chainer.links.BatchNormalization method), 316
init_scope() (chainer.links.Bias method), 212
init_scope() (chainer.links.Bilinear method), 216
init_scope() (chainer.links.BinaryHierarchicalSoftmax method), 324
init_scope() (chainer.links.Blackout method), 327
init_scope() (chainer.links.caffe.CaffeFunction method), 436
init_scope() (chainer.links.Classifier method), 349
init_scope() (chainer.links.Convolution2D method), 221
init_scope() (chainer.links.ConvolutionND method), 225
init_scope() (chainer.links.CRF1d method), 331
init_scope() (chainer.links.Deconvolution2D method), 230
init_scope() (chainer.links.DeconvolutionND method), 233
init_scope() (chainer.links.DepthwiseConvolution2D method), 237
init_scope() (chainer.links.DilatedConvolution2D method), 242
init_scope() (chainer.links.EmbedID method), 246
init_scope() (chainer.links.GoogLeNet method), 359
init_scope() (chainer.links.GRU method), 250
init_scope() (chainer.links.Highway method), 253
init_scope() (chainer.links.Inception method), 256
init_scope() (chainer.links.InceptionBN method), 260
init_scope() (chainer.links.LayerNormalization method), 320
init_scope() (chainer.links.Linear method), 264
init_scope() (chainer.links.LSTM method), 268
init_scope() (chainer.links.Maxout method), 342
init_scope() (chainer.links.MLPConvolution2D method), 271
init_scope() (chainer.links.model.vision.resnet.ResNetLayers method), 364
init_scope() (chainer.links.NegativeSampling method), 346
init_scope() (chainer.links.NStepBiGRU method), 275
init_scope() (chainer.links.NStepBiLSTM method), 278
init_scope() (chainer.links.NStepBiRNNReLU method), 281
init_scope() (chainer.links.NStepBiRNNTanh method), 285
init_scope() (chainer.links.NStepGRU method), 288
init_scope() (chainer.links.NStepLSTM method), 291
init_scope() (chainer.links.NStepRNNReLU method), 295
init_scope() (chainer.links.NStepRNNTanh method), 298
init_scope() (chainer.links.PReLU method), 338
init_scope() (chainer.links.ResNet101Layers method), 372
init_scope() (chainer.links.ResNet152Layers method), 376
init_scope() (chainer.links.ResNet50Layers method), 368
init_scope() (chainer.links.Scale method), 301
init_scope() (chainer.links.SimplifiedDropconnect method), 335
init_scope() (chainer.links.StatefulGRU method), 304
init_scope() (chainer.links.StatefulPeepholeLSTM method), 308
init_scope() (chainer.links.StatelessLSTM method), 312
init_scope() (chainer.links.VGG16Layers method), 354
initialize() (chainer.Parameter method), 53
initialize() (chainer.training.Extension method), 98
<table>
<thead>
<tr>
<th>Method / Attribute / Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize() (Evaluator method)</td>
<td>422</td>
</tr>
<tr>
<td>initialize() (ExponentialShift method)</td>
<td>423</td>
</tr>
<tr>
<td>initialize() (LinearShift method)</td>
<td>424</td>
</tr>
<tr>
<td>initialize() (LogReport method)</td>
<td>425</td>
</tr>
<tr>
<td>initialize() (PlotReport method)</td>
<td>428</td>
</tr>
<tr>
<td>initialize() (PrintReport method)</td>
<td>429</td>
</tr>
<tr>
<td>initialize() (ProgressBar method)</td>
<td>430</td>
</tr>
<tr>
<td>initializer (Parameter attribute)</td>
<td>54</td>
</tr>
<tr>
<td>IntervalTrigger (class in chainer.triggers)</td>
<td>431</td>
</tr>
<tr>
<td>inv() (in module chainer.functions)</td>
<td>193</td>
</tr>
<tr>
<td>is_debug() (in module chainer)</td>
<td>99</td>
</tr>
<tr>
<td>is_new_epoch (ParallelUpdater attribute)</td>
<td>95</td>
</tr>
<tr>
<td>is_new_epoch (StandardUpdater attribute)</td>
<td>94</td>
</tr>
<tr>
<td>is_new_epoch (Updater attribute)</td>
<td>97</td>
</tr>
<tr>
<td>Iterator (class in chainer.dataset)</td>
<td>85</td>
</tr>
<tr>
<td>LabeledImageDataset (class in chainer.datasets)</td>
<td>413</td>
</tr>
<tr>
<td>Lasso (class in chainer.optimizer)</td>
<td>79</td>
</tr>
<tr>
<td>LayerNormalization (class in chainer.links)</td>
<td>318</td>
</tr>
<tr>
<td>LeakyReLU (in module chainer.functions)</td>
<td>123</td>
</tr>
<tr>
<td>LeCunUniform (class in chainer.initializers)</td>
<td>404</td>
</tr>
<tr>
<td>Linear (class in chainer.links)</td>
<td>261</td>
</tr>
<tr>
<td>linear() (in module chainer.functions)</td>
<td>163</td>
</tr>
<tr>
<td>linear_interpolate() (in module chainer.functions)</td>
<td>193</td>
</tr>
<tr>
<td>LinearShift (class in chainer.training.extensions)</td>
<td>423</td>
</tr>
<tr>
<td>Link (class in chainer)</td>
<td>62</td>
</tr>
<tr>
<td>link() (Chain method)</td>
<td>69</td>
</tr>
<tr>
<td>link() (ChainList method)</td>
<td>72</td>
</tr>
<tr>
<td>link() (Link method)</td>
<td>65</td>
</tr>
<tr>
<td>link() (BatchNormalization method)</td>
<td>316</td>
</tr>
<tr>
<td>link() (Bias method)</td>
<td>212</td>
</tr>
<tr>
<td>link() (Bilinear method)</td>
<td>216</td>
</tr>
<tr>
<td>link() (BinaryHierarchicalSoftmax method)</td>
<td>324</td>
</tr>
<tr>
<td>link() (BlackOut method)</td>
<td>328</td>
</tr>
<tr>
<td>link() (caffe.CaffeFunction method)</td>
<td>436</td>
</tr>
<tr>
<td>link() (Classifier method)</td>
<td>350</td>
</tr>
<tr>
<td>link() (Convolution2D method)</td>
<td>221</td>
</tr>
<tr>
<td>links() (ConvolutionND method)</td>
<td>225</td>
</tr>
<tr>
<td>links() (CRF1d method)</td>
<td>331</td>
</tr>
<tr>
<td>links() (Deconvolution2D method)</td>
<td>230</td>
</tr>
<tr>
<td>links() (DeconvolutionND method)</td>
<td>234</td>
</tr>
<tr>
<td>links() (DepthwiseConvolution2D method)</td>
<td>238</td>
</tr>
<tr>
<td>links() (DilatedConvolution2D method)</td>
<td>243</td>
</tr>
<tr>
<td>links() (EmbedID method)</td>
<td>246</td>
</tr>
<tr>
<td>links() (GoogLeNet method)</td>
<td>359</td>
</tr>
<tr>
<td>links() (GRU method)</td>
<td>250</td>
</tr>
<tr>
<td>links() (Highway method)</td>
<td>253</td>
</tr>
<tr>
<td>links() (Inception method)</td>
<td>257</td>
</tr>
<tr>
<td>links() (InceptionBN method)</td>
<td>260</td>
</tr>
<tr>
<td>links() (LayerNormalization method)</td>
<td>320</td>
</tr>
<tr>
<td>links() (Linear method)</td>
<td>264</td>
</tr>
<tr>
<td>links() (LSTM method)</td>
<td>268</td>
</tr>
<tr>
<td>links() (Maxout method)</td>
<td>343</td>
</tr>
<tr>
<td>links() (MLPConvolution2D method)</td>
<td>272</td>
</tr>
<tr>
<td>links() (model.vision.resnet.ResNetLayers method)</td>
<td>364</td>
</tr>
<tr>
<td>links() (NegativeSampling method)</td>
<td>346</td>
</tr>
<tr>
<td>links() (NSkipBiGRU method)</td>
<td>275</td>
</tr>
<tr>
<td>links() (NSkipBiLSTM method)</td>
<td>278</td>
</tr>
<tr>
<td>links() (NSkipBiRNReLU method)</td>
<td>282</td>
</tr>
<tr>
<td>links() (NSkipBiRNNNTanh method)</td>
<td>285</td>
</tr>
<tr>
<td>links() (NSkipGRU method)</td>
<td>288</td>
</tr>
<tr>
<td>links() (NSkipLSTM method)</td>
<td>292</td>
</tr>
<tr>
<td>links() (NSkipRNReLU method)</td>
<td>295</td>
</tr>
<tr>
<td>links() (NSkipRNNNTanh method)</td>
<td>298</td>
</tr>
<tr>
<td>links() (PRELU method)</td>
<td>339</td>
</tr>
<tr>
<td>links() (ResNet101Layers method)</td>
<td>372</td>
</tr>
<tr>
<td>links() (ResNet152Layers method)</td>
<td>377</td>
</tr>
<tr>
<td>links() (ResNet50Layers method)</td>
<td>368</td>
</tr>
<tr>
<td>links() (Scale method)</td>
<td>301</td>
</tr>
<tr>
<td>links() (SimplifiedDropconnect method)</td>
<td>335</td>
</tr>
<tr>
<td>links() (StatefulGRU method)</td>
<td>305</td>
</tr>
<tr>
<td>links() (StatefulPeepholeLSTM method)</td>
<td>308</td>
</tr>
<tr>
<td>links() (StatelessLSTM method)</td>
<td>312</td>
</tr>
<tr>
<td>links() (VGG16Layers method)</td>
<td>354</td>
</tr>
<tr>
<td>load() (Deserializer method)</td>
<td>82</td>
</tr>
<tr>
<td>load() (HDF5Deserializer method)</td>
<td>396</td>
</tr>
<tr>
<td>load() (NpzDeserializer method)</td>
<td>394</td>
</tr>
<tr>
<td>load_hdf5() (in module chainer.serializers)</td>
<td>396</td>
</tr>
<tr>
<td>load_npz() (in module chainer.serializers)</td>
<td>394</td>
</tr>
<tr>
<td>load_function_hooks (Function attribute)</td>
<td>61</td>
</tr>
<tr>
<td>load_response_normalization() (in module chainer.functions)</td>
<td>201</td>
</tr>
<tr>
<td>LocalConfig (class in chainer.configuration)</td>
<td>103</td>
</tr>
<tr>
<td>log (LogReport attribute)</td>
<td>425</td>
</tr>
<tr>
<td>log() (in module chainer.functions)</td>
<td>193</td>
</tr>
</tbody>
</table>
log10() (in module chainer.functions), 193
log1p() (in module chainer.functions), 193
log2() (in module chainer.functions), 194
log_softmax() (in module chainer.functions), 124
LogReport (class in chainer.training.extensions), 424
logsumexp() (in module chainer.functions), 194
lr (chainer.optimizers.AdaGrad attribute), 381
lr (chainer.optimizers.Adam attribute), 383
lr (chainer.optimizers.MomentumSGD attribute), 385
lr (chainer.optimizers.NesterovAG attribute), 386
lr (chainer.optimizers.RMSprop attribute), 388
lr (chainer.optimizers.RMSpropGraves attribute), 390
lr (chainer.optimizers.SGD attribute), 391
lr (chainer.optimizers.SMORMS3 attribute), 393
LSTM (class in chainer.links), 266
lstm() (in module chainer.functions), 125

M
make_extension() (in module chainer.training), 98
make_statistics() (chainer.DictSummary method), 113
make_statistics() (chainer.Summary method), 112
ManualScheduleTrigger (class in chainer.training.triggers), 431
matmul() (in module chainer.functions), 194
max() (in module chainer.functions), 195
max_pooling_2d() (in module chainer.functions), 204
max_pooling_nd() (in module chainer.functions), 204
maximum() (in module chainer.functions), 195
Maxout (class in chainer.links), 340
maxout() (in module chainer.functions), 126
MaxValueTrigger (class in chainer.training.triggers), 432
mean() (in module chainer.functions), 195
mean_absolute_error() (in module chainer.functions), 183
mean_squared_error() (in module chainer.functions), 183
memoize() (in module chainer.cuda), 106
min() (in module chainer.functions), 195
minimum() (in module chainer.functions), 196
MinValueTrigger (class in chainer.training.triggers), 432
MLPConvolution2D (class in chainer.links), 269
momentum (chainer.optimizers.MomentumSGD attribute), 385
momentum (chainer.optimizers.NesterovAG attribute), 386
momentum (chainer.optimizers.RMSpropGraves attribute), 390
MomentumSGD (class in chainer.optimizers), 383
MultiprocessIterator (class in chainer.iterators), 418
MultiprocessParallelUpdater (class in chainer.training.updaters), 95

N
n_step_bigru() (in module chainer.functions), 164
n_step_bilstm() (in module chainer.functions), 165
n_step_birnn() (in module chainer.functions), 167
n_step_gru() (in module chainer.functions), 169
n_step_lstm() (in module chainer.functions), 170
n_step_rnn() (in module chainer.functions), 171
name (chainer.function.FunctionHook attribute), 398
name (chainer.function_hooks.PrintHook attribute), 400
name (chainer.function_hooks.TimerHook attribute), 400
name (chainer.optimizer.GradientClipping attribute), 79
name (chainer.optimizer.GradientNoise attribute), 80
name (chainer.optimizer.Lasso attribute), 79
name (chainer.optimizer.WeightDecay attribute), 78
name (chainer.Parameter attribute), 55
name (chainer.Variable attribute), 51
namedlinks() (chainer.Chain method), 69
tablelinks() (chainer.ChainList method), 72
namedlinks() (chainer.Link method), 65
namedlinks() (chainer.links.BatchNormalization method), 316
namedlinks() (chainer.links.Bias method), 212
namedlinks() (chainer.links.Bilinear method), 216
namedlinks() (chainer.links.BinaryHierarchicalSoftmax method), 324
namedlinks() (chainer.links.BlackOut method), 328
namedlinks() (chainer.links.caffe.CaffeFunction method), 436
namedlinks() (chainer.links.Classifier method), 350
namedlinks() (chainer.links.Convolution2D method), 221
namedlinks() (chainer.links.ConvolutionND method), 225
namedlinks() (chainer.links.CRF1d method), 331
namedlinks() (chainer.links.Deconvolution2D method), 230
namedlinks() (chainer.links.DeconvolutionND method), 234
namedlinks() (chainer.links.DepthwiseConvolution2D method), 238
namedlinks() (chainer.links.DilatedConvolution2D method), 243
namedlinks() (chainer.links.EmbedID method), 247
namedlinks() (chainer.links.GoogLeNet method), 359
namedlinks() (chainer.links.GRU method), 250
namedlinks() (chainer.links.Highway method), 253
namedlinks() (chainer.links.Inception method), 257
namedlinks() (chainer.links.InceptionBN method), 260
namedlinks() (chainer.links.LayerNormalization method), 320
namedlinks() (chainer.links.Linear method), 264
namedlinks() (chainer.links.LSTM method), 268
namedlinks() (chainer.links.Maxout method), 343
namedlinks() (chainer.links.MultiConvolution2D method), 272
namedlinks() (chainer.models.vision.resnet.ResNetLayers method), 364
namedlinks() (chainer.links.NegativeSampling method), 346
namedlinks() (chainer.links.NStepBiGRU method), 275
namedlinks() (chainer.links.NStepBiLSTM method), 278
namedlinks() (chainer.links.NStepBiRNNReLU method), 282
namedlinks() (chainer.links.NStepBiRNNTanh method), 285
namedlinks() (chainer.links.NStepGRU method), 288
namedlinks() (chainer.links.NStepLSTM method), 292
namedlinks() (chainer.links.NStepRNNTanh method), 295
namedlinks() (chainer.links.NStepRNNReLU method), 298
namedlinks() (chainer.links.PReLU method), 301
namedlinks()(chainer.links.ResNet101Layers method), 305
namedparams() (chainer.links.StatefulGRU method), 308
namedparams() (chainer.links.StatefulPeepholeLSTM method), 312
namedparams() (chainer.links.VGG16Layers method), 316
namedparams() (chainer.links.Bias method), 318
namedparams() (chainer.links.LayerNormalization method), 320
namedparams() (chainer.links.Linear method), 324
namedparams() (chainer.CaffeFunction method), 328
namedparams() (chainer.links.CRF1d method), 331
namedparams() (chainer.links.Convolution2D method), 334
namedparams() (chainer.GluonCell method), 337
namedparams() (chainer.links.ConvolutionND method), 340
namedparams() (chainer.PReLU method), 343
namedparams() (chainer.GLU method), 346
namedparams() (chainer.CaffeFunction method), 349
namedparams() (chainer.LinearWithBias method), 352
namedparams() (chainer.links.Bilinear method), 354
namedparams() (chainer.links.Classifier method), 357
namedparams() (chainer.links.Caffe.CaffeFunction method), 360
namedparams() (chainer.links.EmbedID method), 363
namedparams() (chainer.GluonCell method), 366
namedparams() (chainer.variables.Value method), 369
namedparams() (chainer.links.Dropout method), 372
namedparams() (chainer.ChainList method), 375
namedparams() (chainer.links.Classifier method), 378
namedparams() (chainer.LinearWithBias method), 381
namedparams() (chainer.LSTMCell method), 384
namedparams() (chainer.layers.reflective.ReflPool2D method), 387
namedparams() (chainer.LSTM method), 390
namedparams() (chainer.LinearWithBias method), 393
namedparams() (chainer.LinearWithBias method), 396
namedparams() (chainer.LinearWithBias method), 399
new_epoch() (chainer.GradientMethod method), 77
new_epoch() (chainer.Optimizer method), 75
ndim (chainer.Parameter attribute), 51
negative_sampling() (in module chainer.functions), 184
NesterovAG (class in chainer.optimizers), 344
NesterovAG (class in chainer.optimizers), 347
negative_sampling() (in module chainer.functions), 185
new_epoch() (chainer.GradientMethod method), 77
new_epoch() (chainer.Optimizer method), 75
new_epoch() (chainer.optimizers.AdaDelta method), 379
new_epoch() (chainer.optimizers.AdaGrad method), 380
new_epoch() (chainer.optimizers.Adam method), 382
new_epoch() (chainer.optimizers.MomentumSGD method), 384
new_epoch() (chainer.optimizers.NesterovAG method), 385
new_epoch() (chainer.optimizers.RMSprop method), 387
new_epoch() (chainer.optimizers.RMSpropGraves method), 389
new_epoch() (chainer.optimizers.SGD method), 390
new_epoch() (chainer.optimizers.SMORMS3 method), 392
next() (chainer.dataset.Iterator method), 85
next() (chainer.iterators.MultiprocessIterator method), 418
next() (chainer.iterators.SerialIterator method), 417
no_backprop_mode() (in module chainer), 61
node (chainer.Parameter attribute), 55
node (chainer.Variable attribute), 51
Normal (class in chainer.initializers), 402
normalize() (in module chainer.functions), 202
NpzDeserializer (class in chainer.serializers), 394
NStepBiGRU (class in chainer.links), 273
NStepBiLSTM (class in chainer.links), 276
NStepBiRNNReLU (class in chainer.links), 279
NStepBiRNNTanh (class in chainer.links), 282
NStepGRU (class in chainer.links), 286
NStepLSTM (class in chainer.links), 289
NStepRNNReLU (class in chainer.links), 292
NStepRNNTanh (class in chainer.links), 296
numerical_grad() (in module chainer.gradient_check), 118

O
observe_lr() (in module chainer.training.extensions), 425
observe_value() (in module chainer.training.extensions), 425
One() (in module chainer.initializers), 402
Optimizer (class in chainer), 75
Orthogonal (class in chainer.initializers), 403

P
pad() (in module chainer.functions), 141
pad_sequence() (in module chainer.functions), 142
ParallelUpdater (class in chainer.training), 94
Parameter (class in chainer), 51
params() (chainer.Chain method), 69
params() (chainer.ChainList method), 72
params() (chainer.Link method), 65
params() (chainer.links.BatchNormalization method), 317
params() (chainer.links.Bias method), 212
params() (chainer.links.Bilinear method), 216
params() (chainer.links.BinaryHierarchicalSoftmax method), 325
params() (chainer.links.BlackOut method), 328
params() (chainer.links.caffe.CaffeFunction method), 436
params() (chainer.links.Classifier method), 350
params() (chainer.links.Convolution2D method), 221
params() (chainer.links.ConvolutionND method), 225
params() (chainer.links.CRF1d method), 331
params() (chainer.links.Deconvolution2D method), 230
params() (chainer.links.DeconvolutionND method), 234
params() (chainer.links.DepthwiseConvolution2D method), 238
params() (chainer.links.DilatedConvolution2D method), 243
params() (chainer.links.EmbedID method), 247
params() (chainer.links.GoogLeNet method), 359
params() (chainer.links.GRU method), 250
params() (chainer.links.Highway method), 253
params() (chainer.links.Inception method), 257
params() (chainer.links.InceptionBN method), 260
params() (chainer.links.LayerNormalization method), 321
params() (chainer.links.Linear method), 264
params() (chainer.links.LSTM method), 268
params() (chainer.links.Maxout method), 343
params() (chainer.links.MLPConvolution2D method), 272
params() (chainer.links.model.vision.resnet.ResNetLayers method), 364
params() (chainer.links.NegativeSampling method), 346
params() (chainer.links.NStepBiGRU method), 275
params() (chainer.links.NStepBiLSTM method), 278
params() (chainer.links.NStepBiRNNReLU method), 282
params() (chainer.links.NStepBiRNNTanh method), 285
params() (chainer.links.NStepBiGRU method), 288
params() (chainer.links.NStepBiLSTM method), 292
params() (chainer.links.NStepBiRNNReLU method), 295
params() (chainer.links.NStepBiRNNTanh method), 298
params() (chainer.links.PReLU method), 339
params() (chainer.links.ResNet101Layers method), 373
params() (chainer.links.ResNet152Layers method), 377
params() (chainer.links.ResNet50Layers method), 368
params() (chainer.links.Scale method), 301
params() (chainer.links.SimplifiedDropconnect method), 336
params() (chainer.links.StatefulGRU method), 305
params() (chainer.links.StatefulPeepholeLSTM method), 308
params() (chainer.links.StatelessLSTM method), 312
params() (chainer.links.VGG16Layers method), 354
permutate() (in module chainer.functions), 142
PlotReport (class in chainer.training.extensions), 427
predict() (chainer.links.GoogLeNet method), 359
predict() (chainer.links.model.vision.resnet.ResNetLayers method), 364
predict() (chainer.links.ResNet101Layers method), 373
predict() (chainer.links.ResNet152Layers method), 377
predict() (chainer.links.ResNet50Layers method), 368
predict() (chainer.links.VGG16Layers method), 354
PReLU (class in chainer.links), 337
prelu() (in module chainer.functions), 127
prepare() (in module chainer.links.model.vision.googlenet), 360
prepare() (in module chainer.links.model.vision.resnet), 378
prepare() (in module chainer.links.model.vision.vgg), 355
previous_epoch_detail (chainer.iterators.MultiprocessIterator attribute), 419
previous_epoch_detail (chainer.iterators.SerialIterator attribute), 418
previous_epoch_detail (chainer.training.ParallelUpdater attribute), 95
previous_epoch_detail (chainer.training.StandardUpdater attribute), 94
previous_epoch_detail (chainer.training.updaters.MultiprocessParallelUpdater attribute), 97
PrintHook (class in chainer.function_hooks), 399
PrintReport (class in chainer.training.extensions), 428
priority (chainer.training.extensions.Evaluator attribute), 98
priority (chainer.training.extensions.ExponentialShift attribute), 422
priority (chainer.training.extensions.LinearShift attribute), 424
priority (chainer.training.extensions.LogReport attribute), 425
priority (chainer.training.extensions.PrintReport attribute), 428
priority (chainer.training.extensions.ProgressBar attribute), 429
ProgressBar (class in chainer.training.extensions), 429
reallocate_cleared_grads() (chainer.optimizers.Adam method), 382
reallocate_cleared_grads() (chainer.optimizers.MomentumSGD method), 384
reallocate_cleared_grads() (chainer.optimizers.NesterovAG method), 385
reallocate_cleared_grads() (chainer.optimizers.RMSprop method), 387
reallocate_cleared_grads() (chainer.optimizers.RMSpropGraves method), 389
reallocate_cleared_grads() (chainer.optimizers.SMORMS3 method), 392
reduce() (in module chainer.cuda), 107
register_persistent() (chainer.Chain method), 69
register_persistent() (chainer.ChainList method), 72
register_persistent() (chainer.Link method), 65
register_persistent() (chainer.links.BatchNormalization method), 317
register_persistent() (chainer.links.Bias method), 212
register_persistent() (chainer.links.Bilinear method), 216
register_persistent() (chainer.links.BinaryHierarchicalSoftmax method), 325
register_persistent() (chainer.links.BlackOut method), 328
register_persistent() (chainer.links.caffe.CaffeFunction method), 436
register_persistent() (chainer.links.Classifier method), 350
register_persistent() (chainer.links.Convolution2D method), 221
register_persistent() (chainer.links.ConvolutionND method), 225
register_persistent() (chainer.links.CRF1d method), 332
register_persistent() (chainer.links.Deconvolution2D method), 230
register_persistent() (chainer.links.DeconvolutionND method), 234
register_persistent() (chainer.links.DepthwiseConvolution2D method), 238
register_persistent() (chainer.links.DilatedConvolution2D method), 243
register_persistent() (chainer.links.EmbedID method), 247
register_persistent() (chainer.links.GoogLeNet method), 359
register_persistent() (chainer.links.GRU method), 250
register_persistent() (chainer.links.Highway method), 253
register_persistent() (chainer.links.Inception method), 499
<table>
<thead>
<tr>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_cpu() (chainer.utils.WalkerAlias method)</td>
<td>108</td>
</tr>
<tr>
<td>sample_gpu() (chainer.utils.WalkerAlias method)</td>
<td>108</td>
</tr>
<tr>
<td>save() (chainer.Serializer method)</td>
<td>82</td>
</tr>
<tr>
<td>save() (chainer.serializers.DictionarySerializer method)</td>
<td>393</td>
</tr>
<tr>
<td>save() (chainer.serializers.HDF5Serializer method)</td>
<td>395</td>
</tr>
<tr>
<td>save_npz() (in module chainer.serializers)</td>
<td>394</td>
</tr>
<tr>
<td>Scope (class in chainer.links)</td>
<td>299</td>
</tr>
<tr>
<td>scale() (in module chainer.functions)</td>
<td>196</td>
</tr>
<tr>
<td>select_item() (in module chainer.functions)</td>
<td>144</td>
</tr>
<tr>
<td>separate() (in module chainer.functions)</td>
<td>144</td>
</tr>
<tr>
<td>SerialIterator (class in chainer.iterators)</td>
<td>417</td>
</tr>
<tr>
<td>serialize() (chainer.Chain method)</td>
<td>69</td>
</tr>
<tr>
<td>serialize() (chainer.ChainList method)</td>
<td>72</td>
</tr>
<tr>
<td>serialize() (chainer.dataset.Iterator method)</td>
<td>85</td>
</tr>
<tr>
<td>serialize() (chainer.GradientMethod method)</td>
<td>77</td>
</tr>
<tr>
<td>serialize() (chainer.iterators.SequentialIterator method)</td>
<td>418</td>
</tr>
<tr>
<td>serialize() (chainer.iterators.SerialIterator method)</td>
<td>418</td>
</tr>
<tr>
<td>serialize() (chainer.Link method)</td>
<td>65</td>
</tr>
<tr>
<td>serialize() (chainer.links.BatchNormalization method)</td>
<td>317</td>
</tr>
<tr>
<td>serialize() (chainer.links.Bias method)</td>
<td>213</td>
</tr>
<tr>
<td>serialize() (chainer.links.Bilinear method)</td>
<td>217</td>
</tr>
<tr>
<td>serialize() (chainer.links.BinaryHierarchicalSoftmax method)</td>
<td>325</td>
</tr>
<tr>
<td>serialize() (chainer.links.BlackOut method)</td>
<td>328</td>
</tr>
<tr>
<td>serialize() (chainer.links.CaffeFunction method)</td>
<td>436</td>
</tr>
<tr>
<td>serialize() (chainer.links.Classifier method)</td>
<td>350</td>
</tr>
<tr>
<td>serialize() (chainer.links.Convolution2D method)</td>
<td>222</td>
</tr>
<tr>
<td>serialize() (chainer.links.ConvolutionND method)</td>
<td>226</td>
</tr>
<tr>
<td>serialize() (chainer.links.CRF1d method)</td>
<td>322</td>
</tr>
<tr>
<td>serialize() (chainer.links.Deconvolution2D method)</td>
<td>231</td>
</tr>
<tr>
<td>serialize() (chainer.links.DeconvolutionND method)</td>
<td>234</td>
</tr>
<tr>
<td>serialize() (chainer.links.DepthwiseConvolution2D method)</td>
<td>238</td>
</tr>
<tr>
<td>serialize() (chainer.links.DilatedConvolution2D method)</td>
<td>243</td>
</tr>
<tr>
<td>serialize() (chainer.links.EmbedID method)</td>
<td>247</td>
</tr>
<tr>
<td>serialize() (chainer.links.GoogLeNet method)</td>
<td>359</td>
</tr>
<tr>
<td>serialize() (chainer.links.GRU method)</td>
<td>250</td>
</tr>
<tr>
<td>serialize() (chainer.links.Highway method)</td>
<td>254</td>
</tr>
<tr>
<td>serialize() (chainer.links.Inception method)</td>
<td>257</td>
</tr>
<tr>
<td>serialize() (chainer.links.InceptionBN method)</td>
<td>260</td>
</tr>
<tr>
<td>serialize() (chainer.links.LayerNormalization method)</td>
<td>321</td>
</tr>
<tr>
<td>serialize() (chainer.links.Linear method)</td>
<td>265</td>
</tr>
<tr>
<td>serialize() (chainer.links.LSTM method)</td>
<td>269</td>
</tr>
<tr>
<td>serialize() (chainer.links.Maxout method)</td>
<td>343</td>
</tr>
<tr>
<td>serialize() (chainer.links.MLPConvolution2D method)</td>
<td>272</td>
</tr>
<tr>
<td>serialize() (chainer.links.model.vision.resnet.ResNetLayers method)</td>
<td>364</td>
</tr>
<tr>
<td>serialize() (chainer.links.NegativeSampling method)</td>
<td>347</td>
</tr>
<tr>
<td>serialize() (chainer.links.NStepBiGRU method)</td>
<td>275</td>
</tr>
<tr>
<td>serialize() (chainer.links.NStepBiLSTM method)</td>
<td>279</td>
</tr>
<tr>
<td>serialize() (chainer.links.NStepBiRNNeLU method)</td>
<td>282</td>
</tr>
<tr>
<td>serialize() (chainer.links.NStepBiRNNNTanh method)</td>
<td>285</td>
</tr>
<tr>
<td>serialize() (chainer.links.NStepGRU method)</td>
<td>289</td>
</tr>
<tr>
<td>serialize() (chainer.links.NStepLSTM method)</td>
<td>292</td>
</tr>
<tr>
<td>serialize() (chainer.links.NStepRNNReLU method)</td>
<td>295</td>
</tr>
<tr>
<td>serialize() (chainer.links.NStepRNNTanh method)</td>
<td>298</td>
</tr>
<tr>
<td>serialize() (chainer.links.PReLU method)</td>
<td>339</td>
</tr>
<tr>
<td>serialize() (chainer.links.ResNet101Layers method)</td>
<td>373</td>
</tr>
<tr>
<td>serialize() (chainer.links.ResNet152Layers method)</td>
<td>377</td>
</tr>
<tr>
<td>serialize() (chainer.links.ResNet50Layers method)</td>
<td>369</td>
</tr>
<tr>
<td>serialize() (chainer.links.Scale method)</td>
<td>302</td>
</tr>
<tr>
<td>serialize() (chainer.links.SimplifiedDropconnect method)</td>
<td>336</td>
</tr>
<tr>
<td>serialize() (chainer.links.StatefulGRU method)</td>
<td>305</td>
</tr>
<tr>
<td>serialize() (chainer.links.StatefulPeepholeLSTM method)</td>
<td>309</td>
</tr>
<tr>
<td>serialize() (chainer.links StatelessLSTM method)</td>
<td>312</td>
</tr>
<tr>
<td>serialize() (chainer.links.VGG16Layers method)</td>
<td>355</td>
</tr>
<tr>
<td>serialize() (chainer.Optimizer method)</td>
<td>75</td>
</tr>
<tr>
<td>serialize() (chainer.optimizers.AdaDelta method)</td>
<td>379</td>
</tr>
<tr>
<td>serialize() (chainer.optimizers.AdaGrad method)</td>
<td>381</td>
</tr>
<tr>
<td>serialize() (chainer.optimizers.Adam method)</td>
<td>382</td>
</tr>
<tr>
<td>serialize() (chainer.optimizers.MomentumSGD method)</td>
<td>384</td>
</tr>
<tr>
<td>serialize() (chainer.optimizers.NesterovAG method)</td>
<td>386</td>
</tr>
<tr>
<td>serialize() (chainer.optimizers.RMSprop method)</td>
<td>387</td>
</tr>
<tr>
<td>serialize() (chainer.optimizers.RMSpropGraves method)</td>
<td>389</td>
</tr>
<tr>
<td>serialize() (chainer.optimizers.SGD method)</td>
<td>390</td>
</tr>
<tr>
<td>serialize() (chainer.optimizers.SMORSM3 method)</td>
<td>392</td>
</tr>
<tr>
<td>serialize() (chainer.training.Extension method)</td>
<td>98</td>
</tr>
<tr>
<td>serialize() (chainer.training.extensions.Evaluator method)</td>
<td>422</td>
</tr>
<tr>
<td>serialize() (chainer.training.extensions.ExponentialShift method)</td>
<td>423</td>
</tr>
<tr>
<td>serialize() (chainer.training.extensions.LinearShift method)</td>
<td>424</td>
</tr>
<tr>
<td>serialize() (chainer.training.extensions.LogReport method)</td>
<td>425</td>
</tr>
<tr>
<td>serialize() (chainer.training.extensions.PrintReport method)</td>
<td>428</td>
</tr>
<tr>
<td>serialize() (chainer.training.extensions.ProgressBar method)</td>
<td>429</td>
</tr>
<tr>
<td>serialize() (chainer.training.extensions.ProgressBar method)</td>
<td>430</td>
</tr>
<tr>
<td>serialize() (chainer.training.ParallelUpdater method)</td>
<td>95</td>
</tr>
<tr>
<td>serialize() (chainer.training.StandardUpdater method)</td>
<td>93</td>
</tr>
<tr>
<td>serialize() (chainer.training.Trainer method)</td>
<td>91</td>
</tr>
</tbody>
</table>
serialize() (chainer.training.Updater method), 92
serialize() (chainer.training.updaters.MultiprocessParallelUpdater method), 96
serialize() (chainer.UpdateRule method), 74
Serializer (class in chainer), 81
set_creator() (chainer.Parameter method), 53
set_creator() (chainer.Variable method), 49
set_creator() (chainer.variable.VariableNode method), 56
set_dataset_root() (in module chainer.dataset), 87
set_debug() (in module chainer), 100
set_max_workspace_size() (in module chainer.cuda), 108
set_state() (chainer.links.GRU method), 250
set_state() (chainer.links.LSTM method), 269
set_state() (chainer.links.StatefulGRU method), 305
setup() (chainer.GradientMethod method), 78
setup() (chainer.Optimizer method), 76
setup() (chainer.optimizers.AdaDelta method), 379
setup() (chainer.optimizers.AdaGrad method), 381
setup() (chainer.optimizers.Adam method), 383
setup() (chainer.optimizers.MomentumSGD method), 384
setup() (chainer.optimizers.NesterovAG method), 386
setup() (chainer.optimizers.RMSprop method), 387
setup() (chainer.optimizers.RMSpropGraves method), 389
setup() (chainer.optimizers.SGD method), 391
setup() (chainer.optimizers.SMORMS3 method), 392
setup_workers() (chainer.training.updaters.MultiprocessParallelUpdater method), 96
SGD (class in chainer.optimizers), 390
shape (chainer.Parameter attribute), 55
shape (chainer.Variable attribute), 51
show() (chainer.configuration.GlobalConfig method), 103
show() (chainer.configuration.LocalConfig method), 103
sigmoid() (in module chainer.functions), 128
sigmoid_cross_entropy() (in module chainer.functions), 184
simplified_dropconnect() (in module chainer.functions), 199
SimplifiedDropconnect (class in chainer.links), 333
sin() (in module chainer.functions), 197
sinh() (in module chainer.functions), 197
size (chainer.Parameter attribute), 55
size (chainer.Variable attribute), 51
size() (chainer.utils.type_check.TypeInfoTuple method), 116
slstm() (in module chainer.functions), 129
SMORMS3 (class in chainer.optimizers), 391
snapshot() (in module chainer.training.extensions), 426
snapshot_object() (in module chainer.training.extensions), 426
softmax() (in module chainer.functions), 130
softmax_cross_entropy() (in module chainer.functions), 185
softplus() (in module chainer.functions), 130
spatial2depth() (in module chainer.functions), 145
spatial_pyramid_pooling_2d() (in module chainer.functions), 205
spatial_transformer_grid() (in module chainer.functions), 145
spatial_transformer_sampler() (in module chainer.functions), 146
split_axis() (in module chainer.functions), 147
split_dataset() (in module chainer.datasets), 409
split_dataset_random() (in module chainer.datasets), 409
sqrt() (in module chainer.functions), 197
square() (in module chainer.functions), 198
squared_difference() (in module chainer.functions), 198
squared_error() (in module chainer.functions), 186
squeeze() (in module chainer.functions), 147
stack (chainer.Function attribute), 61
stack() (in module chainer.functions), 148
StandardUpdater (class in chainer.training), 92
start_finetuning() (chainer.links.BatchNormalization method), 317
state (chainer.UpdateRule attribute), 74
StatefulGRU (class in chainer.links), 302
StatefulPeepholeLSTM (class in chainer.links), 306
StatelessLSTM (class in chainer.links), 309
SubDataset (class in chainer.datasets), 407
sum() (in module chainer.functions), 198
Tanh (class in chainer), 112
summary() (chainer.Parameter method), 53
summary() (chainer.Variable method), 49
swapaxes() (in module chainer.functions), 149
T

tan() (in module chainer.functions), 198
tanh() (in module chainer.functions), 131
tile() (in module chainer.functions), 149
TimerHook (class in chainer.function_hooks), 400
to_cpu() (chainer.Chain method), 69
to_cpu() (chainer.ChainList method), 72
to_cpu() (chainer.Link method), 66
to_cpu() (chainer.links.BatchNormalization method), 317
to_cpu() (chainer.links.Bias method), 213
to_cpu() (chainer.links.Bilinear method), 217
to_cpu() (chainer.links.BinaryHierarchicalSoftmax method), 325
to_cpu() (chainer.links.BlackOut method), 328
to_cpu() (chainer.links.caffe.CaffeFunction method), 436
to_cpu() (chainer.links.Classifier method), 350
to_cpu() (chainer.links.Convolution2D method), 222
to_cpu() (chainer.links.ConvolutionND method), 226
to_cpu() (chainer.links.CRF1d method), 332
to_cpu() (chainer.links.Deconvolution2D method), 231
to_cpu() (chainer.links.DeconvolutionND method), 235
transpose() (in module chainer.functions), 151
transpose_sequence() (in module chainer.functions), 151
trigger (chainer.training.Extension attribute), 98
trigger (chainer.training.extensions.Evaluator attribute), 422
trigger (chainer.training.extensions.ExponentialShift attribute), 423
trigger (chainer.training.extensions.LinearShift attribute), 424
trigger (chainer.training.extensions.LogReport attribute), 425
trigger (chainer.training.extensions.PrintReport attribute), 429
trigger (chainer.training.extensions.ProgressBar attribute), 429
triplet() (in module chainer.functions), 186
TupleDataset (class in chainer.datasets), 407
TypeInfo (class in chainer.utils.type_check), 116
TypeInfoTuple (class in chainer.utils.type_check), 116
U
unary_math_function_unittest() (in module chainer.testing), 119
unchain() (chainer.Function method), 61
unchain() (chainer.Parameter method), 54
unchain() (chainer.Variable method), 50
unchain_variable.VariableNode method), 56
unchain_backward() (chainer.Parameter method), 54
unchain_backward() (chainer.Variable method), 50
Uniform (class in chainer.initializers), 404
unpooling_2d() (in module chainer.functions), 206
update() (chainer.GradientMethod method), 78
update() (chainer.Optimizer method), 76
update() (chainer.optimizers.AdaDelta method), 379
update() (chainer.optimizers.AdaGrad method), 381
update() (chainer.optimizers.Adam method), 383
update() (chainer.optimizers.MomentumSGD method), 384
update() (chainer.optimizers.NesterovAG method), 386
update() (chainer.optimizers.RMSprop method), 387
update() (chainer.optimizers.RMSpropGraves method), 389
update() (chainer.optimizers.SGD method), 391
update() (chainer.optimizers.SMORMS3 method), 392
update() (chainer.Parameter method), 54
update() (chainer.training.ParallelUpdater method), 95
update() (chainer.training.StandardUpdater method), 93
update() (chainer.training.updaters.MultiProcessParallelUpdater method), 97
update() (chainer.UpdateRule method), 74
update_core() (chainer.training.ParallelUpdater method), 95
update_core() (chainer.training.StandardUpdater method), 93
update_core() (chainer.training.updaters.MultiProcessParallelUpdater method), 97
update() (chainer.UpdateRule method), 74
update() (chainer.inputs.CRF1d method), 332
update_enabled (chainer.links.Bias attribute), 213
update_enabled (chainer.links.Bilinear attribute), 217
update_enabled (chainer.links.CaffeFunction attribute), 437
update_enabled (chainer.links.Classifier attribute), 350
update_enabled (chainer.links.Convolution2D attribute), 222
update_enabled (chainer.links.ConvolutionND attribute), 226
update_enabled (chainer.links.CRF1d method), 332
update_enabled (chainer.links.Convolution2D attribute), 226
update_enabled (chainer.links.DepthwiseConvolution2D attribute), 239
update_enabled (chainer.links.DilatedConvolution2D attribute), 244
update_enabled (chainer.links.EmbedID attribute), 248
update_enabled (chainer.links.GoogLeNet attribute), 360
update_enabled (chainer.links.GRU attribute), 251
update_enabled (chainer.links.Highway attribute), 254
update_enabled (chainer.links.Inception attribute), 257
update_enabled (chainer.links.InceptionBN attribute), 261
update_enabled (chainer.links.LayerNormalization attribute), 321
update_enabled (chainer.links.Linear attribute), 265
update_enabled (chainer.links.LSTM attribute), 269
update_enabled (chainer.links.Maxout attribute), 343
update_enabled (chainer.links.MLPConvolution2D attribute), 272
update_enabled (chainer.links.model.vision.resnet.ResNetLayers attribute), 365
update_enabled (chainer.links.NegativeSampling attribute), 347
update_enabled (chainer.links.NStepBiGRU attribute), 321

504 Index
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>276</td>
<td>update_enabled (chainer.links.NStepBiLSTM attribute)</td>
</tr>
<tr>
<td>279</td>
<td>update_enabled (chainer.links.NStepBiRNNReLU attribute)</td>
</tr>
<tr>
<td>282</td>
<td>update_enabled (chainer.links.NStepBiRNNNTanh attribute)</td>
</tr>
<tr>
<td>286</td>
<td>update_enabled (chainer.links.NStepGRU attribute)</td>
</tr>
<tr>
<td>289</td>
<td>update_enabled (chainer.links.NStepLSTM attribute)</td>
</tr>
<tr>
<td>292</td>
<td>update_enabled (chainer.links.NStepRNNTanh attribute)</td>
</tr>
<tr>
<td>295</td>
<td>update_enabled (chainer.links.PReLU attribute)</td>
</tr>
<tr>
<td>299</td>
<td>update_enabled (chainer.links.ResNet101Layers attribute)</td>
</tr>
<tr>
<td>302</td>
<td>update_enabled (chainer.links.ResNet152Layers attribute)</td>
</tr>
<tr>
<td>305</td>
<td>update_enabled (chainer.links.ResNet50Layers attribute)</td>
</tr>
<tr>
<td>309</td>
<td>update_enabled (chainer.links.Scale attribute)</td>
</tr>
<tr>
<td>312</td>
<td>update_enabled (chainer.links.SimplifiedDropconnect attribute)</td>
</tr>
<tr>
<td>318</td>
<td>update_enabled (chainer.links.StatefulGRU attribute)</td>
</tr>
<tr>
<td>322</td>
<td>update_enabled (chainer.links.StatefulPeepholeLSTM attribute)</td>
</tr>
<tr>
<td>326</td>
<td>update_enabled (chainer.links.VGG16Layers attribute)</td>
</tr>
<tr>
<td>332</td>
<td>within_init_scope (chainer.Chain attribute)</td>
</tr>
<tr>
<td>336</td>
<td>within_init_scope (chainer.ChainList attribute)</td>
</tr>
<tr>
<td>340</td>
<td>within_init_scope (chainer.DataParallel attribute)</td>
</tr>
<tr>
<td>343</td>
<td>within_init_scope (chainer.DataParallel2D attribute)</td>
</tr>
<tr>
<td>348</td>
<td>within_init_scope (chainer.DataParallel3D attribute)</td>
</tr>
<tr>
<td>350</td>
<td>within_init_scope (chainer.data_source.DictSource attribute)</td>
</tr>
<tr>
<td>355</td>
<td>within_init_scope (chainer.data_source.ListSource attribute)</td>
</tr>
<tr>
<td>360</td>
<td>within_init_scope (chainer.data_source.NpzSource attribute)</td>
</tr>
<tr>
<td>365</td>
<td>within_init_scope (chainer.data_source.PickleSource attribute)</td>
</tr>
</tbody>
</table>

| Index 505 |

**V**
- Variable (class in chainer), 47
- VariableNode (class in chainer.variable), 55
- VGG16Layers (class in chainer.links), 351
- vstack() (in module chainer.functions), 151

**W**
- WalkerAlias (class in chainer.utils), 108
- WeightDecay (class in chainer.optimizer), 78
- where() (in module chainer.functions), 152
- within_init_scope (chainer.Chain attribute), 70
- within_init_scope (chainer.ChainList attribute), 72
- within_init_scope (chainer.Link attribute), 66
- within_init_scope (chainer.links.BatchNormalization attribute), 318
- within_init_scope (chainer.links.Bias attribute), 213
- within_init_scope (chainer.links.Bilinear attribute), 217
- within_init_scope (chainer.links.BinaryHierarchicalSoftmax attribute), 325
- within_init_scope (chainer.links.BlackOut attribute), 329
- within_init_scope (chainer.links.CaffeFunction attribute), 437
- within_init_scope (chainer.links.Classifier attribute), 350
- within_init_scope (chainer.links.Convolution2D attribute), 222
- within_init_scope (chainer.links.ConvolutionND attribute), 226
- within_init_scope (chainer.links.CRF1d attribute), 332
- within_init_scope (chainer.links.Deconvolution2D attribute), 231
- within_init_scope (chainer.links.DeconvolutionND attribute), 235
- within_init_scope (chainer.links.DepthwiseConvolution2D attribute), 239
- within_init_scope (chainer.links.DilatedConvolution2D attribute), 244
- within_init_scope (chainer.links.EmbedID attribute), 248
- within_init_scope (chainer.links.GoogLeNet attribute), 360
- within_init_scope (chainer.links.GRU attribute), 251
- within_init_scope (chainer.links.Highway attribute), 254
- within_init_scope (chainer.links.Inception attribute), 257
- within_init_scope (chainer.links.InceptionBN attribute), 261
- within_init_scope (chainer.links.LayerNormalization attribute), 321
- within_init_scope (chainer.links.Linear attribute), 265
- within_init_scope (chainer.links.LSTM attribute), 269
- within_init_scope (chainer.links.Maxout attribute), 343
- within_init_scope (chainer.links.MMConvolution2D attribute), 272
- within_init_scope (chainer.links.model.layers.Convolution2D attribute), 365
within_init_scope (chainer.links.NegativeSampling attribute), 347
within_init_scope (chainer.links.NStepBiGRU attribute), 276
within_init_scope (chainer.links.NStepBiLSTM attribute), 279
within_init_scope (chainer.links.NStepBiRNReLU attribute), 282
within_init_scope (chainer.links.NStepBiRNNTanh attribute), 286
within_init_scope (chainer.links.NStepGRU attribute), 289
within_init_scope (chainer.links.NStepLSTM attribute), 292
within_init_scope (chainer.links.NStepRNReLU attribute), 296
within_init_scope (chainer.links.NStepRNNTanh attribute), 299
within_init_scope (chainer.links.PReLU attribute), 340
within_init_scope (chainer.links.ResNet101Layers attribute), 373
within_init_scope (chainer.links.ResNet152Layers attribute), 378
within_init_scope (chainer.links.ResNet50Layers attribute), 365
within_init_scope (chainer.links.Scale attribute), 302
within_init_scope (chainer.links.SimplifiedDropconnect attribute), 336
within_init_scope (chainer.links.StatefulGRU attribute), 305
within_init_scope (chainer.links.StatefulPeepholeLSTM attribute), 309
within_init_scope (chainer.links.StatelessLSTM attribute), 313
within_init_scope (chainer.links.VGG16Layers attribute), 355

xp (chainer.Chain attribute), 70
xp (chainer.ChainList attribute), 73
xp (chainer.Link attribute), 66
xp (chainer.links.BatchNormalization attribute), 318
xp (chainer.links.Bias attribute), 213
xp (chainer.links.Bilinear attribute), 217
xp (chainer.links.BinaryHierarchicalSoftmax attribute), 325
xp (chainer.links.BlackOut attribute), 329
xp (chainer.links.caffe.CaffeFunction attribute), 437
xp (chainer.links.Classifier attribute), 350
xp (chainer.links.Convolution2D attribute), 222
xp (chainer.links.ConvolutionND attribute), 226
xp (chainer.links.CRF1d attribute), 332
xp (chainer.links.Deconvolution2D attribute), 231
xp (chainer.links.DeconvolutionND attribute), 235
xp (chainer.links.DepthwiseConvolution2D attribute), 239
xp (chainer.links.DilatedConvolution2D attribute), 244
xp (chainer.links.EmbedID attribute), 248
xp (chainer.links.GoogLeNet attribute), 360
xp (chainer.links.GRU attribute), 251
xp (chainer.links.Highway attribute), 254
zp (chainer.links.Inception attribute), 257
zp (chainer.links.InceptionBN attribute), 261
zp (chainer.links.LayerNormalization attribute), 321
zp (chainer.links.Linear attribute), 265
zp (chainer.links.LSTM attribute), 269
zp (chainer.links.Maxout attribute), 343
zp (chainer.links.MLPConvolution2D attribute), 272
zp (chainer.links.model.vision.resnet.ResNetLayers attribute), 365
zp (chainer.links.NegativeSampling attribute), 347
zp (chainer.links.NStepBiGRU attribute), 276
zp (chainer.links.NStepBiLSTM attribute), 279
zp (chainer.links.NStepBiRNReLU attribute), 282
zp (chainer.links.NStepBiRNNTanh attribute), 286
zp (chainer.links.NStepGRU attribute), 289
zp (chainer.links.NStepLSTM attribute), 292
zp (chainer.links.NStepRNReLU attribute), 296
zp (chainer.links.NStepRNNTanh attribute), 299
zp (chainer.links.PReLU attribute), 340
zp (chainer.links.ResNet101Layers attribute), 373
zp (chainer.links.ResNet152Layers attribute), 378
zp (chainer.links.ResNet50Layers attribute), 365
zp (chainer.links.Scale attribute), 302
zp (chainer.links.SimplifiedDropconnect attribute), 336
zp (chainer.links.StatefulGRU attribute), 305
zp (chainer.links.StatefulPeepholeLSTM attribute), 309
zp (chainer.links.StatelessLSTM attribute), 313
zp (chainer.links.VGG16Layers attribute), 355

Z
Zero() (in module chainer.initializers), 402
zero_grad() (chainer.links.Bilinear method), 217
zero_grad() (chainer.Parameter method), 54
zero_grad() (chainer.Variable method), 50
zero_grads() (chainer.Chain method), 69
zero_grads() (chainer.ChainList method), 72
zero_grads() (chainer.Link method), 66
zero_grads() (chainer.links.BatchNormalization method), 317
zero_grads() (chainer.links.Bias method), 213
zero_grads() (chainer.links.Bilinear method), 217
zero_grads() (chainer.links.BinaryHierarchicalSoftmax method), 325
zero_grads() (chainer.links.BlackOut method), 328
zero_grads() (chainer.links.caffe.CaffeFunction method), 436
zero_grads() (chainer.links.Classifier method), 350