Chainer Documentation

Release 1.8.2

Preferred Networks, inc. and Preferred Infrastructure, inc.

May 17, 2016

Contents

This is the Chainer documentation.

CHAPTER 1

Install Guide

1.1 Before installing Chainer

We recommend these platforms.

- Ubuntu 14.04 LTS 64bit
- CentOS 7 64bit

Chainer is supported on Python 2.7.6+, 3.4.3+, 3.5.1+. Chainer uses C++ compiler such as g++. You need to install it before installing Chainer. This is typical installation method for each platform:

```
# Ubuntu 14.04
$ apt-get install g++

# CentOS 7
$ yum install gcc-c++
```

If you use old `setuptools`, upgrade it:

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

1.2 Install Chainer

Chainer depends on these Python packages:

- NumPy 1.9, 1.10, 1.11
- Six 1.9

CUDA support

- CUDA 6.5, 7.0, 7.5
- filelock

cuDNN support

- cuDNN v2, v3, v4

Caffe model support

- Protocol Buffers
- protobuf>=3.0.0 is required for Py3

All these libraries are automatically installed with `pip` or `setup.py`.

HDF5 serialization is optional

- h5py 2.5.0

1.2.1 Install Chainer via pip

We recommend to install Chainer via pip:

```
$ pip install chainer
```

1.2.2 Install Chainer from source

You can use `setup.py` to install Chainer from source:

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

1.2.3 When an error occurs...

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

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

1.2.4 Install Chainer with CUDA

You need to install CUDA Toolkit before installing Chainer. If you have CUDA in a default directory or set `CUDA_PATH` correctly, Chainer installer finds CUDA automatically:

```
$ pip install chainer
```

Note: Chainer installer looks up `CUDA_PATH` environment variable first. If it is empty, the installer looks for `nvcc` command from `PATH` environment variable and use its parent directory as the root directory of CUDA installation. If `nvcc` command is also not found, the installer tries to use the default directory for Ubuntu `/usr/local/cuda`.

If you installed CUDA into a non-default directory, you need to specify the directory with `CUDA_PATH` environment variable:

```
$ CUDA_PATH=/opt/nvidia/cuda pip install chainer
```

Warning: If you want to use `sudo` to install Chainer, note that `sudo` command initializes all environment variables. Please specify `CUDA_PATH` environment variable inside `sudo` like this:

```
$ sudo CUDA_PATH=/opt/nvidia/cuda pip install chainer
```

1.2.5 Install Chainer with CUDA and cuDNN

cuDNN is a library for Deep Neural Networks that NVIDIA provides. Chainer can use cuDNN. If you want to enable cuDNN, install cuDNN and CUDA before installing Chainer. We recommend you to install cuDNN to CUDA directory. For example if you uses Ubuntu Linux, copy `.h` files to `include` directory and `.so` files to `lib64` directory:

```
$ cp /path/to/cudnn.h $CUDA_PATH/include
$ cp /path/to/libcudnn.so* $CUDA_PATH/lib64
```

The destination directories depend on your environment.

1.2.6 Install Chainer for developers

Chainer uses Cython (>=0.23). Developers need to use Cython to regenerate C++ sources from `pyx` files. We recommend to use `pip` with `-e` option for editable mode:

```
$ pip install -U cython
$ cd /path/to/chainer/source
$ pip install -e .
```

Users need not to install Cython as a distribution package of Chainer only contains generated sources.

1.2.7 Support HDF5 serialization

Install h5py manually to activate HDF5 serialization. This feature is optional:

```
$ pip install h5py
```

Before installing h5py, you need to install libhdf5. It depends on your environment:

```
# Ubuntu 14.04
$ apt-get install libhdf5-dev

# CentOS 7
$ yum -y install epel-release
$ yum install hdf5-devel
```

1.3 Uninstall Chainer

Use pip to uninstall Chainer:

```
$ pip uninstall chainer
```

Note: When you upgrade Chainer, `pip` sometimes installed various version of Chainer in `site-packages`. Please uninstall it repeatedly until `pip` returns an error.

1.4 Upgrade Chainer

Just use `pip` with `-U` option:

```
$ pip install -U chainer
```

1.5 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
```

When you install Chainer without CUDA, and after that you want to use CUDA, please reinstall Chainer. You need to reinstall Chainer when you want to upgrade CUDA.

1.6 What "recommend" means?

We tests Chainer automatically with Jenkins. All supported environments are tested in this environment. We cannot guarantee that Chainer works on other environments.

1.7 FAQ

1.7.1 The installer says "hdf5.h is not found"

You don't have libhdf5. Please install hdf5. See *Before installing Chainer*.

1.7.2 MemoryError happens

You maybe failed to install Cython. Please install it manually. See *When an error occurs....*.

1.7.3 Examples says "cuDNN is not enabled"

You failed to build Chainer with cuDNN. If you don't need cuDNN, ignore this message. Otherwise, retry to install Chainer with cuDNN. `-vvvv` option helps you. See *Install Chainer with CUDA and cuDNN*.

CHAPTER 2

Chainer Tutorial

2.1 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

2.1.1 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. Torch7 and Theano-based frameworks).

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

```
import numpy as np
import chainer
from chainer import cuda, Function, gradient_check, Variable, optimizers, serializers, utils
from chainer import Link, Chain, ChainList
import chainer.functions as F
import chainer.links as L
```

These imports appear widely in Chainer's codes and examples. For simplicity, we omit this idiom in this tutorial.

2.1.2 Forward/Backward Computation

As described above, Chainer uses "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 `Variable` object. Here we start with simple `ndarray` with only one element:

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

Warning: Chainer currently only supports 32-bit float for most computations.

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

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

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

```
>>> 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:

```
>>> 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`:

```
>>> 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 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 simply done by setting the `grad` attribute of the output variable:

```
>>> 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.

2.1.3 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. 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 links just like functions with parameters.

Note: Actually, these are corresponding to "parameterized functions" in versions up to v1.4.

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 is corresponding 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:

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

Note: Most functions and links only accept mini-batch input, where the first dimension of input arrays 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 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.

```
>>> 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)
```

Gradients of parameters are computed by *backward()* method. Note that gradients are **accumulated** by the method rather than overwritten. So first you must initialize gradients to zero to renew the computation. It can be done by calling the *zerograds()* method.

```
>>> f.zerograds()
```

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

```
>>> 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)
```

2.1.4 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:

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

Here the `L` indicates the *`chainer.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:

```
>>> class MyProc(object):
...     def __init__(self):
...         self.l1 = L.Linear(4, 3)
...         self.l2 = L.Linear(3, 2)
...
...     def forward(self, x):
...         h = self.l1(x)
...         return self.l2(h)
```

In order to make it more reusable, we want to support parameter management, CPU/GPU migration support, 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 defining the above class as a subclass of Chain:

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

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

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.

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

```
>>> 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 is convenient to use an arbitrary number of links. If the number of links is fixed like above case, the Chain class is recommended as a base class.

2.1.5 Optimizer

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

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

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

There are two ways to run optimization. One is manually computing gradients and then call the *update()* method with no arguments. Do not forget resetting gradients beforehand!

```
>>> model.zerograds()
>>> # compute gradient here...
>>> optimizer.update()
```

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

```
>>> def lossfun(args...):
...     ...
...     return loss
>>> optimizer.update(lossfun, args...)
```

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 by the *update()* method in advance of the actual update. For example, we can set weight decay regularization by running the next line beforehand:

```
>>> 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.

2.1.6 Serializer

The last core feature described in this page is serializer. Serializer is a simple interface to serialize or deserialize an object. *Link* and *Optimizer* supports serialization by serializers.

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 these 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.

2.1.7 Example: Multi-layer Perceptron on MNIST

Now you can solve a multiclass classification task using a multi-layer perceptron. Here we use hand-written digits dataset called MNIST, which is one of the long-standing defacto "hello world" of machine learning. This MNIST example is also found in `examples/mnist` directory of the official repository.

In order to use MNIST, we prepared `load_mnist_data` function at `examples/mnist/data.py`:

```
>>> import data
>>> mnist = data.load_mnist_data()
```

The mnist dataset consists of 70,000 grayscale images of size 28x28 (i.e. 784 pixels) and corresponding digit labels. First, we scale pixels to [0, 1] values, and divide the dataset into 60,000 training samples and 10,000 test samples.

```
>>> x_all = mnist['data'].astype(np.float32) / 255
>>> y_all = mnist['target'].astype(np.int32)
>>> x_train, x_test = np.split(x_all, [60000])
>>> y_train, y_test = np.split(y_all, [60000])
```

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

```
>>> class MLP(Chain):
...     def __init__(self):
...         super(MLP, self).__init__(
...             l1=L.Linear(784, 100),
...             l2=L.Linear(100, 100),
...             l3=L.Linear(100, 10),
```

```
...         )
...
...     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:

```
>>> class Classifier(Chain):
...     def __init__(self, predictor):
...         super(Classifier, self).__init__(predictor=predictor)
...
...     def __call__(self, x, t):
...         y = self.predictor(x)
...         self.loss = F.softmax_cross_entropy(y, t)
...         self.accuracy = F.accuracy(y, t)
...         return self.loss
```

This Classifier class computes accuracy and loss, and returns the loss value. *softmax_cross_entropy()* computes the loss value given prediction and ground truth labels. *accuracy()* computes the prediction accuracy. We can set an arbitrary predictor link to an instance of the classifier.

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

```
>>> model = L.Classifier(MLP())
>>> optimizer = optimizers.SGD()
>>> optimizer.setup(model)
```

Finally, we can write a learning loop as following:

```
>>> batchsize = 100
>>> datasize = 60000
>>> for epoch in range(20):
...     print('epoch %d' % epoch)
...     indexes = np.random.permutation(datasize)
...     for i in range(0, datasize, batchsize):
...         x = Variable(x_train[indexes[i : i + batchsize]])
...         t = Variable(y_train[indexes[i : i + batchsize]])
...         optimizer.update(model, x, t)
epoch 0...
```

Only the last three lines are the code related to Chainer, which are already described above. Note that, in the last line, we pass `model` as a loss function.

These three lines can also be rewritten as follows, with explicit gradient computation:

```
>>> batchsize = 100
>>> datasize = 60000
>>> for epoch in range(20):
...     print('epoch %d' % epoch)
...     indexes = np.random.permutation(datasize)
...     for i in range(0, datasize, batchsize):
...         x = Variable(x_train[indexes[i : i + batchsize]])
```

```
...         t = Variable(y_train[indexes[i : i + batchsize]])
...         model.zerograds()
...         loss = model(x, t)
...         loss.backward()
...         optimizer.update()
epoch 0...
```

You may find that, at each iteration, the network is defined by forward computation, used for backprop, and then disposed. By leveraging this "Define-by-Run" scheme, you can imagine that recurrent nets with variable length input are simply handled by just using loop over different length input for each iteration.

After or during optimization, we want to evaluate the model on the test set. It can be achieved simply by calling forward function:

```
>>> sum_loss, sum_accuracy = 0, 0
>>> for i in range(0, 10000, batchsize):
...     x = Variable(x_test[i : i + batchsize])
...     t = Variable(y_test[i : i + batchsize])
...     loss = model(x, t)
...     sum_loss += loss.data * batchsize
...     sum_accuracy += model.accuracy.data * batchsize
...
>>> mean_loss = sum_loss / 10000
>>> mean_accuracy = sum_accuracy / 10000
```

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

2.2 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 volatile variables to prevent network construction

2.2.1 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 periodically 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:

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

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

```
>>> 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:

```
>>> x2 = Variable(np.random.randn(10, 100).astype(np.float32))
>>> y2 = l(x2)
```

Based on this LSTM link, let's write our recurrent network as a new chain:

```
class RNN(Chain):
    def __init__(self):
        super(RNN, self).__init__(
            embed=L.EmbedID(1000, 100),  # word embedding
            mid=L.LSTM(100, 50),  # the first LSTM layer
            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

rnn = RNN()
model = L.Classifier(rnn)
optimizer = optimizers.SGD()
optimizer.setup(model)
```

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.

```
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:

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

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

```
rnn.reset_state()
optimizer.update(compute_loss, x_list)
```

2.2.2 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:

```
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.zerograds()
        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.

2.2.3 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 passing `volatile` flag to all input variables. Such variables are called *volatile variables*.

Volatile variable is created by passing `volatile='on'` at the construction:

```
x_list = [Variable(..., volatile='on') 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 volatile variable does not remember the computation history.

Volatile variables are also useful to evaluate feed-forward networks to reduce the memory footprint.

Variable's volatility can be changed directly by setting the `Variable.volatile` attribute. This enables us to combine a fixed feature extractor network and a trainable predictor network. 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):

```
x = Variable(x_data, volatile='on')
feat = fixed_func(x)
feat.volatile = 'off'
y = predictor_func(feat)
y.backward()
```

At first, the input variable `x` is volatile, so `fixed_func` is executed in volatile mode, i.e. without memorizing the computation history. Then the intermediate variable `feat` is manually set to non-volatile, so `predictor_func` is executed in non-volatile 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.

Warning: It is not allowed to mix volatile and non-volatile variables as arguments to same function. If you want to create a variable that behaves like a non-volatile variable while can be mixed with volatile ones, use `'auto'` flag instead of `'off'` flag.

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 `examples/ptb` directory implements 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.

2.3 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

2.3.1 Relationship between Chainer and CuPy

Note: As of the release of v1.3.0, Chainer changes its GPU backend from PyCUDA to CuPy. CuPy covers all features of PyCUDA used by Chainer, though their interfaces are not compatible.

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.

2.3.2 Basics of `cupy.ndarray`

Note: CuPy does not require explicit initialization, so `cuda.init()` function is removed as of v1.3.0.

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.cuda.Device* object as follows:

```
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:

```
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:

```
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:

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

It is equivalent to the following code using CuPy:

```
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()* to select a device. It accepts an integer, CuPy array, NumPy array, or None (indicating the current device), and returns an appropriate device object. If the argument is a NumPy array, then *a dummy device object* is returned. The dummy device object supports *with* statements like above which does nothing. Here are some examples:

```
cuda.get_device(1).use()
x_gpu1 = cupy.empty((4, 3), dtype='f')  # 'f' indicates float32

with cuda.get_device(1):
    x_gpu1 = cuda.empty((4, 3), dtype='f')

with cuda.get_device(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:

```
def add1(x):
    with cuda.get_device(x):
        return x + 1
```

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:

```
# 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)))
```

2.3.3 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.

Then we have to transfer each minibatch to the GPU:

```
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)
```

This is almost identical to the code of the original example, we just inserted a call to the *cuda.to_gpu()* function to the minibatch arrays.

2.3.4 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.

```
class ParallelMLP(Chain):
    def __init__(self):
        super(ParallelMLP, self).__init__(
            mlp1_gpu0=MLP(784, 1000, 2000).to_gpu(0),
            mlp1_gpu1=MLP(784, 1000, 2000).to_gpu(1),
            mlp2_gpu0=MLP(2000, 1000, 10).to_gpu(0),
            mlp2_gpu1=MLP(2000, 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
```

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/net.py`.

2.3.5 Data-parallel Computation on Multiple GPUs

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 instance:

```
model_0 = L.Classifier(MLP(784, 1000, 10))
```

Recall that the `MLP` link implements the multi-layer perceptron, and the *Classifier* link wraps it to provide a classifier interface. 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.

```
model_1 = model_0.copy()
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:

```
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:

```
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]]
```

```
        model_0.zerograds()
        model_1.zerograds()

        loss_0 = model_0(Variable(cuda.to_gpu(x_batch[:batchsize//2], 0)),
                         Variable(cuda.to_gpu(y_batch[:batchsize//2], 0)))
        loss_1 = model_1(Variable(cuda.to_gpu(x_batch[batchsize//2:], 1)),
                         Variable(cuda.to_gpu(y_batch[batchsize//2:], 1)))

        loss_0.backward()
        loss_1.backward()

        model_0.addgrads(model_1)
        optimizer.update()

        model_1.copyparams(model_0)
```

Do not forget initializing the gradients of both model copies! One half of the minibatch 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.

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.

2.4 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

2.4.1 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 $f(x, y, z) = x * y + z$. While it is possible to implement this equation using a combination of the $*$ 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:

```
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

```
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

$$\lambda_i = \sum_{j=1}^{m} \frac{\partial y_j}{\partial x_i} \gamma_j \text{ for } i = 1 \dots n$$

where γ is the `grad_outputs`. Note, that the resulting vector λ 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 *forward_gpu()* and *backward_gpu()*:

```
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 *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 *forward()* and *backward()* methods

```
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
        gy = x * gw
        gz = gw
        return gx, gy, gz
```

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:

```
x = Variable(np.random.uniform(-1, 1, (3, 2)).astype(np.float32))
y = Variable(np.random.uniform(-1, 1, (3, 2)).astype(np.float32))
z = Variable(np.random.uniform(-1, 1, (3, 2)).astype(np.float32))
w = 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:

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

w = muladd(x, y, z)
```

2.4.2 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

```
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.

2.4.3 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):
        ...

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

    def forward_gpu(self, inputs):
        cupy = cuda.cupy
        x, y, z = inputs
        w = cuda.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 = cuda.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.*

2.4.4 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(Link):
    def __init__(self, shape):
        # By passing a shape of the parameter, the initializer allocates a
        # parameter variable of the shape.
        super(EltwiseParamProduct, self).__init__(W=shape)
        self.W.data[...] = np.random.randn(*shape)

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

We can also initialize the parameter after the initialization by the *Link.add_param()* method.

```
class EltwiseParamProduct(Link):
    def __init__(self, shape):
        super(EltwiseParamProduct, self).__init__()
        self.add_param('W', shape)
        self.W.data[...] = np.random.randn(*shape)

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

Note that the initializer and the *add_param()* method does not initialize elements of the parameter array. We have to manually initialize the elements by random values, zeros, etc.

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(Link):
    def __init__(self, in_size, out_size):
        super(Linear, self).__init__(W=(out_size, in_size), b=out_size)
        self.W.data[...] = np.random.randn(out_size, in_size) / math.sqrt(in_size)
        self.b.data.fill(0)

    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.

2.4.5 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

```
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 `sum(f(x))`, where `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 *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 *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

```
import unittest

class TestReLU(unittest.TestCase):
    def test_backward_cpu(self):
        x = Variable(np.random.randn(3, 2).astype(np.float32))
        y = F.relu(x)
        y.grad = np.random.randn(3, 2).astype(np.float32)
        y.backward()

        f = lambda: (F.relu(x).data,)
        gx, = gradient_check.numerical_grad(f, (x.data,), (y.grad,))

        gradient_check.assert_allclose(gx, x.grad)
```

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.

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

2.5 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

2.5.1 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`:

```
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:

```
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:

```
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`".

2.5.2 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:

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

You can also check data types with `.dtype`:

```
utils.type_check.expect(x_type.dtype == np.float64)
```

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])
```

2.5.3 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.

2.5.4 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:

```
x_type.shape[0] == utils.type_check.Variable(self.in_size, "in_size")
```

This code can check the equivalent condition below:

```
x_type.shape[0] == self.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: in_types[0].shape[0] == 4  # 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: in_types[0].shape[0] == in_size  # OK, `in_size` is a value that is given to the
Actual: 3 != 4  # You can also check actual value here
```

2.5.5 Call functions

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

```
sum = utils.type_check.Variable(np.sum, 'sum')
utils.type_check.expect(sum(x_type.shape) == 10)
```

Why do we need to wrap the function `numpy.sum` with `utils.type_check.Variable`? `x_type.shape` is not a tuple but an object of *Expr* as we have seen before. Therefore, `numpy.sum(x_type.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: sum(in_types[0].shape) == 10
Actual: 7 != 10
```

2.5.6 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:

```
x_shape = x_type.shape.eval()  # get actual shape (int tuple)
if not more_complicated_condition(x_shape):
    expect_msg = 'Shape is expected to be ...'
    actual_msg = 'Shape is ...'
    raise 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 ...
```

2.5.7 Typical type check example

We show a typical type check for a function.

First check the number of arguments:

```
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:

```
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:

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

After that, check each type:

```
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

3.1 Core functionalities

3.1.1 Variable

class chainer.**Variable**(*data*, *volatile=OFF*, *name=None*)

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 may be constructed in two ways: by the user or by some function. When a variable is created by some function as one of its outputs, the variable holds a reference to that function. This reference is used in error backpropagation (a.k.a. backprop). It is also used in *backward unchaining*. A variable that does not hold a reference to its creator is called a *root* variable. A variable is root if it is created by the user, or if the reference is deleted by *unchain_backward()*.

Users can disable this chaining behavior by setting the volatile flag for the initial variables. When a function gets volatile variables as its inputs, the output variables do not hold references to the function. This acts like unchaining on every function application.

Parameters

- **data** (array) – Initial data array.
- **volatile** (Flag) – Volatility flag. String ('on', 'off', or 'auto') or boolean values can be used, too.
- **name** (*str*) – Name of the variable.

Variables

- **data** – Data array of type either numpy.ndarray or *cupy.ndarray*.
- **grad** – Gradient array. It is None until backprop reaches this variable.
- **creator** – The function who creates this variable. It is None if the variable is not created by any function.
- **volatile** – Ternary *Flag* object. If ON, the variable does not keep track of any function applications. See *Flag* for the detail of ternary flags.

__len__()

Returns the number of elements of the data array.

Returns the number of elements of the data array.

Return type int

addgrad(*var*)
Accumulates the gradient array from given source variable.

This method just runs `self.grad += var.grad`, except that the accumulation is even done across the host and different devices.

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 variables to their creators, and from functions to their inputs. The backprop stops at all root variables. Some functions set `None` as gradients of some inputs, where further backprop does not take place at such input variables.

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 model, the purpose of backprop is to compute gradients of parameters, not of variables, so it is recommended to set this flag False.

copydata(*var*)
Copies the data array from given source variable.

This method just copies the data attribute from given variable to this variable, except that the copy is even done across the host and different devices.

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

debug_print()
Display a summary of the stored data and location of the Variable

label
Short text that represents the function.

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.

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.

unchain_backward()
Deletes references between variables and functions backward.

After this method completes, intermediate variables 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,

i.e. this variable 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.

3.1.2 Flag

class chainer.**Flag**
Ternary flag object for variables.

It takes three values: ON, OFF, and AUTO.

ON and OFF flag can be evaluated as a boolean value. These are converted to True and False, respectively. AUTO flag cannot be converted to boolean. In this case, ValueError is raised.

Parameters **name** (*str, bool, or None*) – Name of the flag. Following values are allowed:

- 'on', 'ON', or True for ON value
- 'off', 'OFF', or False for OFF value
- 'auto', 'AUTO', or None for AUTO value

chainer.**ON** = **ON**
Equivalent to Flag('on').

chainer.**OFF** = **OFF**
Equivalent to Flag('off').

chainer.**AUTO** = **AUTO**
Equivalent to Flag('auto').

chainer.flag.**aggregate_flags**(*flags*)
Returns an aggregated flag given a sequence of flags.

If both ON and OFF are found, this function raises an error. Otherwise, either of ON and OFF that appeared is returned. If all flags are AUTO, then it returns AUTO.

Parameters **flags** (*sequence of Flag*) – Input flags.

Returns The result of aggregation.

Return type *Flag*

3.1.3 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 variables to the function and from the function to its inputs.

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

```
>>> 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 <--- f <--- y
```

If an application of another function g occurs as

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

then the graph grows with a branch:

```
    |--- f <--- y
x <-+
    |--- g <--- z
```

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.

Variables

- **inputs** – A tuple or list of input variables.
- **outputs** – A tuple or list of output variables.
- **type_check_enable** – When it is True, the function checks types of input arguments. Set CHAINER_TYPE_CHECK environment variable 0 to disable type check, or set the variable directly in your own program.

__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* objects. The volatile flags of all input variables must agree.

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

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

Parameters

- **hook** (FunctionHook) – the function hook to be registered.
- **name** (*str*) – The 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 numpy.ndarray object(s).
- **grad_outputs** – Tuple of output gradient numpy.ndarray object(s).

Returns Tuple of input gradient numpy.ndarray object(s). Some or all of them can be None, if the function is not differentiable on corresponding 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_gpu(*inputs*, *grad_outputs*)
Applies backprop to output gradient arrays on GPU.

Parameters

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

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

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

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

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

delete_hook(*name*)
Unregisters the function hook.

Parameters

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

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

It delegates the procedure to *forward_cpu()* or *forward_gpu()* by default. Which it selects is determined by the type of input arrays. Implementations of *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.

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.

unchain()
Purges in/out variables and this function itself from the graph.

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

3.1.4 Link and Chain

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. Parameters are variables registered via the *add_param()* method, or given to the initializer method. Persistent values are arrays, scalars, or any other serializable values registered via the *add_persistent()* method.

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
import numpy as np

class LinearLayer(chainer.Link):

    def __init__(self, n_in, n_out):
        # Parameters are initialized as a numpy array of given shape.
        super(LinearLayer, self).__init__(
            W=(n_out, n_in),
            b=(n_out,),
        )
        self.W.data[...] = np.random.randn(n_out, n_in)
        self.b.data.fill(0)

    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.

Parameters **params** – Shapes of initial parameters. The keywords are used as their names. The names are also set to the parameter variables.

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

add_param(*name*, *shape*, *dtype=<type 'numpy.float32'>*)

Registers a parameter to the link.

The registered parameter is saved and loaded on serialization and deserialization, and involved in the optimization. The data and gradient of the variable are initialized by NaN arrays.

The parameter is set to an attribute of the link with the given name.

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.
- **dtype** – Data type of the parameter array.

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.

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.

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()
Returns a generator of all (path, param) pairs under the hierarchy.

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

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

Returns A generator object that generates all parameters.

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

xp
Array module for this link.

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

zerograds()
Initializes all gradient arrays by zero.

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

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 `/`.

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:

```
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):
        # Create and register three layers for this MLP
```

```
            super(MultiLayerPerceptron, self).__init__(
                layer1=L.Linear(n_in, n_hidden),
                layer2=L.Linear(n_hidden, n_hidden),
                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 initializer method. They also can be registered by the *add_link()* method. 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.

__getitem__(*name*)
Equivalent to getattr.

add_link(*name*, *link*)
Registers a child link to this chain.

The registered link is saved and loaded on serialization and deserialization, and involved in the optimization. The registered link is called a child. The child link is set to an attribute of the chain with the given name.

This method also sets the `name` attribute of the registered link. If the given link already has the name attribute set, then it raises an error.

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.

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.

__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 a number of children.

add_link(*link*)
Registers a child link to this chain.

The registered link is saved and loaded on serialization and deserialization, and involved in the optimization. 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.

This method also sets the `name` attribute of the registered link. If the given link already has the name attribute set, then it raises an error.

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

3.1.5 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. An optimizer can use *internal states* each of which is tied to one of the parameters. State is a dictionary of serializable values (typically arrays of size same as the corresponding parameters). In order to use state dictionaries, the optimizer must override *init_state()* method (or its CPU/GPU versions, *init_state_cpu()* and *init_state_gpu()*).

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.

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.

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.

accumulate_grads(*grads*)
Accumulates gradients from other source.

This method just adds given gradient arrays to gradients that this optimizer holds. It is typically used in data-parallel optimization, where gradients for different shards are computed in parallel and aggregated by this method. This method correctly treats multiple GPU devices.

Parameters **grads** (*Iterable*) – Iterable of gradient arrays to be accumulated.

Deprecated since version v1.5: Use the *chainer.Link.addgrads()* method of the target link instead.

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. It accepts the optimizer object.

- **name** (*str*) – Name of the registration. If omitted, `hook.name` is used by default.

call_hooks()
Invokes hook functions in registration order.

clip_grads(*maxnorm*)
Clips the norm of whole gradients up to the threshold.

Parameters **maxnorm** (*float*) – Threshold of gradient L2 norm.

Deprecated since version v1.5: Use the *GradientClipping* hook function instead.

compute_grads_norm()
Computes the norm of whole gradients.

Returns L2 norm of whole gradients, i.e. square root of sum of square of all gradient elements.

Return type float

Warning: This method returns a CPU-computed value, which means that this method synchronizes between CPU and GPU if at least one of the gradients reside on the GPU.

Deprecated since version v1.5.

init_state(*param*, *state*)
Initializes the optimizer state corresponding to the parameter.

This method should add needed items to the `state` dictionary. Each optimizer implementation that uses its own states should override this method or CPU/GPU dedicated versions (*init_state_cpu()* and *init_state_gpu()*).

Parameters

- **param** (`Variable`) – Parameter variable.
- **state** (*dict*) – State dictionary.

See also:

init_state_cpu(), *init_state_gpu()*

init_state_cpu(*param*, *state*)
Initializes the optimizer state on CPU.

This method is called from *init_state()* by default.

Parameters

- **param** (`Variable`) – Parameter variable. Its data array is of type `numpy.ndarray`.
- **state** (*dict*) – State dictionary.

See also:

init_state()

init_state_gpu(*param*, *state*)
Initializes the optimizer state on GPU.

This method is called from *init_state()* by default.

Parameters

- **param** (`Variable`) – Parameter variable. Its data array is of type *cupy.ndarray*.
- **state** (*dict*) – State dictionary.

See also:

init_state()

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.

prepare()
Prepares for an update.

This method initializes missing optimizer states (e.g. for newly added parameters after the set up), and copies arrays in each state dictionary to CPU or GPU according to the corresponding parameter array.

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** (`Link`) – Target link object.

update(*lossfun=None*, **args*, ***kwds*)
Updates the parameters and optimizer states.

This method updates the parameters of the target link and corresponding optimizer states. The behavior of this method is different for the cases either `lossfun` is given or not.

If `lossfun` is given, then this method initializes the gradients by zeros, calls it with given extra arguments, and calls the *backward()* method of its output to compute the gradients. The 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.

weight_decay(*decay*)
Applies weight decay to the parameter/gradient pairs.

Parameters **decay** (*float*) – Coefficient of weight decay

Deprecated since version v1.5: Use the *WeightDecay* hook function instead.

zero_grads()
Fills all gradient arrays by zeros.

Deprecated since version v1.5: Use the *chainer.Link.zerograds()* method for the target link instead.

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.

An implementation of a gradient method must override the following methods:

- `init_state()` or both `init_state_cpu()` and `init_state_gpu()`
- *update_one()* or both *update_one_cpu()* and *update_one_gpu()*

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 use it 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_one()* method (or its CPU/GPU versions, *update_one_cpu()* and *update_one_gpu()*).

update_one(*param*, *state*)
Updates a parameter based on the corresponding gradient and state.

This method calls appropriate one from `update_param_cpu()` or `update_param_gpu()`.

Parameters

- **param** (`Variable`) – Parameter variable.
- **state** (*dict*) – State dictionary.

update_one_cpu(*param*, *state*)
Updates a parameter on CPU.

Parameters

- **param** (`Variable`) – Parameter variable.
- **state** (*dict*) – State dictionary.

update_one_gpu(*param*, *state*)
Updates a parameter on GPU.

Parameters

- **param** (`Variable`) – Parameter variable.
- **state** (*dict*) – State dictionary.

Hook functions

class chainer.optimizer.**WeightDecay**(*rate*)
Optimizer 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.

class chainer.optimizer.**Lasso**(*rate*)
Optimizer 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.

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.

3.1.6 Serializer

class chainer.**AbstractSerializer**
Abstract base class of all serializers and deserializers.

__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.

Parameters

- **key** (*str*) – Name of the serialization entry.
- **value** (*scalar, array, or str*) – Object to be (de)serialized.

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.

class chainer.**Serializer**
Base class of all serializers.

save(*obj*)
Saves an object by this serializer.

This is equivalent to `obj.serialize(self)`.

Parameters **obj** – Target object to be serialized.

class chainer.**Deserializer**
Base class of all deserializers.

load(*obj*)
Loads an object from this deserializer.

This is equivalent to `obj.serialize(self)`.

Parameters **obj** – Target object to be serialized.

3.1.7 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.

chainer.**is_debug**()
Get the debug mode.

Returns Return `True` if Chainer is in debug mode.

Return type bool

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.

3.1.8 FunctionSet (deprecated)

class chainer.**FunctionSet**(***links*)
Set of links (as "parameterized functions").

FunctionSet is a subclass of *Chain*. Function registration is done just by adding an attribute to :class:‘ object.

Deprecated since version v1.5: Use *Chain* instead.

Note: FunctionSet was used for manipulation of one or more parameterized functions. The concept of parameterized function is gone, and it has been replaced by *Link* and *Chain*.

__getitem__(*key*)
Returns an attribute by name.

Parameters **key** (*str*) – Name of the attribute.

Returns Attribute.

Example

```
>>> import chainer.links as L
>>> model = FunctionSet(l1=L.Linear(10, 10), l2=L.Linear(10, 10))
>>> l1 = model['l1']  # equivalent to l1 = model.l1
```

collect_parameters()
Returns a tuple of parameters and gradients.

Returns Tuple (pair) of two tuples. The first element is a tuple of parameter arrays, and the second is a tuple of gradient arrays.

copy_parameters_from(*params*)
Copies parameters from another source without reallocation.

Parameters **params** (*Iterable*) – Iterable of parameter arrays.

gradients
Tuple of gradient arrays of all registered functions.

The order of gradients is consistent with *parameters()* property.

parameters
Tuple of parameter arrays of all registered functions.

The order of parameters is consistent with *parameters()* property.

3.2 Utilities

3.2.1 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 are imported to `cuda` module for convenience (refer to this table when reading chainer's source codes).

imported name	original name
`chainer.cuda.cupy`	*cupy*
`chainer.cuda.ndarray`	*cupy.ndarray*
`chainer.cuda.cupy.cuda`	`cupy.cuda`
`chainer.cuda.Device`	*cupy.cuda.Device*
`chainer.cuda.Event`	*cupy.cuda.Event*
`chainer.cuda.Stream`	*cupy.cuda.Stream*

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

`chainer.cuda.`**get_device**(**args*)
Gets the device from an ID integer or an array object.

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. numpy.ndarray objects are skipped. If all arguments are numpy.ndarray objects, it returns a dummy device object. Otherwise, the first non-numpy object is used to select a device. If it is a *cupy.ndarray* object, its device is returned. Otherwise, the argument is passed to the initializer of *Device* and it is returned.

Returns Device object specified by given args.

See also:

See *cupy.cuda.Device* for the device selection not by arrays.

CuPy array allocation and copy

Note: As of v1.3.0, the following array construction wrappers are marked as deprecated. Use the corresponding functions of the *cupy* module instead. The main difference of them is that the default dtype is changed from float32 to float64.

Deprecated functions	Recommended functions
chainer.cuda.empty	*cupy.empty()*
chainer.cuda.empty_like	*cupy.empty_like()*
chainer.cuda.zeros	*cupy.zeros()*
chainer.cuda.zeros_like	*cupy.zeros_like()*
chainer.cuda.ones	*cupy.ones()*
chainer.cuda.ones_like	*cupy.ones_like()*
chainer.cuda.full	*cupy.full()*
chainer.cuda.full_like	*cupy.full_like()*

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**(*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**(*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.

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

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.**elementwise**()

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**()

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**(**args*)
Gets an appropriate one from numpy or *cupy*.

This is almost equivalent to *cupy.get_array_module()*. The only difference is that this function can be used even if CUDA is not available.

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

3.2.2 Common algorithms

class chainer.utils.**WalkerAlias**(*probs*)
Implementation of Walker's alias method.

This method generates a random sample from given probabilities $p_1, \dots, p_n$ 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 *sum(probs)*.

See: Wikipedia article

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.

to_gpu()
Make a sampler GPU mode.

3.3 Assertion and Testing

Chainer provides some facilities to make debugging easy.

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.

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.

3.3.1 Type checking utilities

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

```
>>> 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:

```
>>> 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.
```

---

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.

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.

Parameters **bool_exprs** (*tuple of Bool expressions*) – Bool expressions you want to evaluate.

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.

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.

size()
Returns an expression representing its length.

Returns An expression object representing length of the tuple.

Return type *Expr*

3.3.2 Gradient checking utilities

chainer.gradient_check.**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.

chainer.gradient_check.**check_backward**(*func*, *x_data*, *y_grad*, *params=()*, *eps=0.001*, *atol=1e-05*, *rtol=0.0001*)
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:

```
>> def test_my_func(self):
>>   func = MyFunc()
>>   x1_data = xp.array(...)
>>   x2_data = xp.array(...)
>>   gy_data = xp.array(...)
>>   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 *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:

```
>>   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:

```
>>   check_backward(my_loss_func, (x1_data, x2_data), None)
```

If `MyFunc` returns multiple outputs, pass all gradients for outputs as a tuple:

```
>>   gy1_data = xp.array(...)
>>   gy2_data = xp.array(...)
>>   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:

```
>>   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` s.

Function objects are acceptable as `func` argument:

```
>>   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,)`.
- **y_grad** (*ndarray or tuple of ndarrays or None*) – A set of `ndarray` s 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*) – 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 *assert_allclose()*.
- **rtol** (*float*) – Relative tolerance to be passed to *assert_allclose()*.

See: *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

3.4 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.

3.4.1 Activation functions

clipped_relu

chainer.functions.**clipped_relu**(*x*, *z=20.0*)

Clipped Rectifier Unit function.

This function is expressed as $ClippedReLU(x, z) = \min(\max(0, x), z)$, where $z(> 0)$ is a clipping value.

Parameters

- **x** (Variable) – Input variable.
- **z** (*float*) – Clipping value. (default = 20.0)

Returns Output variable.

Return type *Variable*

elu

chainer.functions.**elu**(*x*, *alpha=1.0*)

Exponential Linear Unit function.

This function is expressed as

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\exp(x) - 1) & \text{if } x < 0, \end{cases}$$

where α is a parameter. See: http://arxiv.org/abs/1511.07289

Parameters

- **x** (Variable) – Input variable.
- **alpha** (*float*) – Parameter α.

Returns Output variable.

Return type *Variable*

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) – Input variable.
- **slope** (*float*) – Slope value a.

Returns Output variable.

Return type *Variable*

log_softmax

chainer.functions.**log_softmax**(*x*, *use_cudnn=True*)

Channelwise log-softmax function.

This function computes its logarithm of softmax along the second axis. Let $i = (i_1, i_2, \ldots, i_d)^\top$ be the d dimensional index array and $x = f(i)$ be the corresponding d dimensional input array. For each index i of the input array $f(i)$, it computes the logarithm of the probability $\log p(x)$ defined as

$$p(i) = \frac{\exp(f(i))}{\sum_{i'_2} \exp(f(i'))},$$

where $i' = (i_1, i'_2, \ldots, i_d)$.

$$p(x) = \frac{\exp(f(x))}{\sum_{x'} \exp(f(x'))}.$$

This method is theoretically equivalent to `log(softmax(x))` but is more stable.

Note: `log(softmax(x))` may cause underflow when `x` is too small, because `softmax(x)` may returns `0`. `log_softmax` method is more stable.

Parameters

- **x** (Variable) – Input variable.
- **use_cudnn** (*bool*) – If `True`, cuDNN is enabled and cuDNN ver. 3 or later is used, then this function uses cuDNN as the core implementation.

Returns Output variable.

Return type *Variable*

See also:

softmax()

lstm

chainer.functions.**lstm**(*c_prev*, *x*)

Long Short-Term Memory units as an activation function.

This function implements LSTM units with forget gates. Let the previous cell state c_{prev} and the incoming signal x.

First, the incoming signal 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 length of c_{prev}.

The split input signals 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 outputs as:

$$c = \tanh(a)\text{sigmoid}(i) + c_{\text{prev}}\text{sigmoid}(f),$$
$$h = \tanh(c)\text{sigmoid}(o).$$

These are returned as a tuple of two variables.

Parameters

- **c_prev** (Variable) – 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) – Variable that holds the incoming signal. It must have the second dimension 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.

Example

Assuming y is the current input signal, c is the previous cell state, and h is the previous output signal from an lstm function. Each of y, c and h has n_units channels. Most typical preparation of x is:

```
>>> import chainer, chainer.functions as F
>>> n_units = 100
>>> y = chainer.Variable(numpy.zeros((1, n_units), 'f'))
>>> h = chainer.Variable(numpy.zeros((1, n_units), 'f'))
>>> c = chainer.Variable(numpy.zeros((1, n_units), 'f'))
>>> model = chainer.Chain(w=F.Linear(n_units, 4 * n_units),
...                       v=F.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 sources a, i, f, o from the current input y and the previous output h. Different parameters are used for different kind of input sources.

---

maxout

chainer.functions.**maxout**(*x*, *pool_size*, *axis=1*)

Maxout activation function.

It accepts an input tensor `x`, reshapes the `axis` dimension (say the size being `M * pool_size`) into two dimensions `(M, pool_size)`, and takes maximum along the `axis` dimension. The output of this function is same as `x` except that `axis` dimension is transformed from `M * pool_size` to `M`.

Typically, `x` is the output of a linear layer or a convolution layer. The following is the example where we use *maxout()* in combination with a Linear link.

```
>>> import numpy, chainer, chainer.links as L
>>> in_size, out_size, pool_size = 100, 100, 100
>>> l = L.Linear(in_size, out_size * pool_size)
>>> x = chainer.Variable(numpy.zeros((1, in_size), 'f'))  # prepare data
>>> x = l(x)
>>> y = maxout(x, pool_size)
```

Parameters **x** (Variable) – Input variable. Its first dimension is assumed to be the *minibatch dimension*. The other dimensions are treated as one concatenated dimension.

Returns Output variable.

Return type *Variable*

See also:

Maxout

prelu

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 $PReLU(x) = \max(x, W * x)$, where $*$ 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 batchsize 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

Return type *Variable*

See also:

PReLU

relu

chainer.functions.**relu**(*x*, *use_cudnn=True*)

Rectified Linear Unit function $f(x) = \max(0, x)$.

Parameters

- **x** (Variable) – Input variable.
- **use_cudnn** (*bool*) – If True and cuDNN is enabled, then this function uses cuDNN as the core implementation.

Returns Output variable.

Return type *Variable*

sigmoid

chainer.functions.**sigmoid**(*x*, *use_cudnn=True*)

Elementwise sigmoid logistic function $f(x) = (1 + \exp(-x))^{-1}$.

Parameters

- **x** (Variable) – Input variable.
- **use_cudnn** (*bool*) – If True and cuDNN is enabled, then this function uses cuDNN as the core implementation.

Returns Output variable.

Return type *Variable*

slstm

chainer.functions.**slstm**(*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_1 and c_2, and incoming signals x_1 and x_2.

First both input signals x_1 and x_2 are split into eight arrays a_1, i_1, f_1, o_1, and a_2, i_2, f_2, o_2. They have the same shape along the second axis. It means that x_1 and x_2 's second axis must have 4 times the length of $c_{1\text{prev}}$ and $c_{2\text{prev}}$.

The split input signals 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 outputs as:

$$c = \tanh(a_1 + a_2)\sigma(i_1 + i_2) + c_{1\text{prev}}\sigma(f_1) + c_{2\text{prev}}\sigma(f_2),$$
$$h = \tanh(c)\sigma(o_1 + o_2),$$

where σ is the elementwise sigmoid function. The function returns c and h as a tuple.

Parameters

- **c_prev1** (Variable) – 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) – Variable that holds the previous cell state of the second child node.
- **x1** (Variable) – Variable that holds the incoming signal from the first child node. It must have the second dimension four times of that of the cell state,
- **x2** (Variable) – Variable that holds the incoming signal 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.

softmax

chainer.functions.**softmax**(*x*, *use_cudnn=True*)

Channelwise softmax function.

This function computes its softmax along the second axis. Let $x = (x_1, x_2, \ldots, x_d)^\top$ 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 $p(x) = \frac{\exp(f(x))}{\sum_{x_2} \exp(f(x))}$.

Parameters

- **x** (Variable) – Input variable.
- **use_cudnn** (*bool*) – If True and cuDNN is enabled, then this function uses cuDNN as the core implementation.

Returns Output variable.

Return type *Variable*

softplus

chainer.functions.**softplus**(*x*, *beta=1.0*)

Elementwise softplus function.

This function is expressed as $f(x) = \frac{1}{\beta}\log(1 + \exp(\beta x))$, where β is a parameter.

Parameters

- **x** (Variable) – Input variable.
- **beta** (*float*) – Parameter β.

Returns Output variable.

Return type *Variable*

tanh

chainer.functions.**tanh**(*x*, *use_cudnn=True*)

Elementwise hyperbolic tangent function.

Parameters

- **x** (Variable) – Input variable.
- **use_cudnn** (*bool*) – If True and cuDNN is enabled, then this function uses cuDNN as the core implementation.

Returns Output variable.

Return type *Variable*

3.4.2 Array manipulations

broadcast

chainer.functions.**broadcast**(**args*)

Broadcast given variables.

Parameters **args** (*Variables*) – Variables to be broadcasted.

Returns Tuple of *Variable* objects which are broadcasted from given arguments.

Return type tuple

broadcast_to

chainer.functions.**broadcast_to**(*x*, *shape*)

Broadcast a given variable to a given shape.

Parameters

- **x** (Variable) – Variable to be broadcasted.
- **shape** (*tuple of int*) – The shape of the output variable.

Returns Output variable broadcasted to the given shape.

Return type *Variable*

concat

chainer.functions.**concat**(*xs*, *axis=1*)

Concatenates given variables along an axis.

Parameters

- **xs** (*tuple of Variables*) – Variables to be concatenated.
- **axis** (*int*) – Axis that the input arrays are concatenated along.

Returns Output variable.

Return type *Variable*

copy

chainer.functions.**copy**(*x*, *dst*)
Copies the input variable onto the specified device.

This function copies the array of input variable onto the device specified by `dst` if the original array is on GPU, and otherwise just copies the array within host memory.

Parameters

- **x** (Variable) – Variable to be copied.
- **dst** – Target device specifier.

Returns Output variable.

Return type *Variable*

expand_dims

chainer.functions.**expand_dims**(*x*, *axis*)
Expands dimensions of an input variable without copy.

Parameters

- **x** (Variable) – Input variable.
- **axis** (*int*) – Position where new axis is to be inserted.

Returns Variable that holds a expanded input.

Return type *Variable*

reshape

chainer.functions.**reshape**(*x*, *shape*)
Reshapes an input variable without copy.

Parameters

- **x** (Variable) – Input variable.
- **shape** (*tuple of ints*) – Target shape.

Returns Variable that holds a reshaped version of the input variable.

Return type *Variable*

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*

split_axis

chainer.functions.**split_axis**(*x*, *indices_or_sections*, *axis*, *force_tuple=False*)
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`, this method returns a tuple even 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).

---

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*

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*

where

chainer.functions.**where**(*condition*, *x*, *y*)
Choose elements depending on condition.

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*

3.4.3 Neural network connections

bilinear

chainer.functions.**bilinear**(*e1*, *e2*, *W*, *V1=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 $e^1 \in \mathbb{R}^{I \cdot J}$, $e^2 \in \mathbb{R}^{I \cdot K}$, $W \in \mathbb{R}^{J \cdot K \cdot L}$, $V^1 \in \mathbb{R}^{J \cdot L}$, $V^2 \in \mathbb{R}^{K \cdot L}$, 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

$$y_{il} = \sum_{jk} e^1_{ij} e^2_{ik} W_{jkl} + \sum_j e^1_{ij} V^1_{jl} + \sum_k e^2_{ik} V^2_{kl} + b_l.$$

Note that V1, V2, 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 `e1` or `e2` 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].

convolution_2d

chainer.functions.**convolution_2d**(*x*, *W*, *b=None*, *stride=1*, *pad=0*, *use_cudnn=True*)

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, 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.
- **use_cudnn** (*bool*) – If `True`, then this function uses cuDNN if available.

Returns Output variable.

Return type *Variable*

The two-dimensional convolution function is defined as follows. Then the `Convolution2D` function computes correlations between filters and patches of size (k_H, k_W) 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 `-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, and (p_H, p_W) the spatial padding size. Then, the output size (h_O, w_O) is determined by the following equations:

$$h_O = (h + 2p_H - k_H)/s_Y + 1,$$
$$w_O = (w + 2p_W - k_W)/s_X + 1.$$

If the bias vector is given, then it is added to all spatial locations of the output of convolution.

See also:

`Convolution2D`

deconvolution_2d

`chainer.functions.`**`deconvolution_2d`**(*x*, *W*, *b=None*, *stride=1*, *pad=0*, *outsize=None*, *use_cudnn=True*)

Two dimensional deconvolution function.

This is an implementation of two-dimensional deconvolution. It takes three variables: input image `x`, the filter weight `W`, and the bias vector `b`.

Parameters

- **`x`** (`Variable`) – Input variable of shape (n, c_I, h, w).
- **`W`** (`Variable`) – Weight variable of shape (c_I, c_O, 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.
- **`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.
- **`use_cudnn`** (*`bool`*) – If `True`, then this function uses cuDNN if available.

The filter weight has four dimensions (c_I, c_O, k_H, k_W) which indicate the number of the number of input channels, output channels, height and width of the kernels, respectively.

The bias vector is of size c_O.

Let X be the input tensor of dimensions (n, c_I, h, w), (s_Y, s_X) the stride of filter application, and (p_H, p_W) the spatial padding size. Then, the output size (h_O, w_O) is determined by the following equations:

$$h_O = s_Y(h-1) + k_H - 2p_H,$$
$$w_O = s_X(w-1) + k_W - 2p_W.$$

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

- **`x`** (`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`

linear

`chainer.functions.`**`linear`**(*x*, *W*, *b=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 $Y = xW^\top + b$.

Parameters

- **x** (`Variable`) – Input variable. Its first dimension 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,)`.

Returns Output variable.

Return type *Variable*

See also:

Linear

3.4.4 Evaluation functions

accuracy

`chainer.functions.`**`accuracy`**(*y*, *t*)

Computes muticlass classification accuracy of the minibatch.

Parameters

- **y** (`Variable`) – Variable holding a matrix whose (i, j)-th element indicates the score of the class j at the i-th example.
- **t** (`Variable`) – Variable holding an int32 vector of ground truth labels.

Returns A variable holding a scalar array of the accuracy.

Return type *Variable*

Note: This function is non-differentiable.

3.4.5 Loss functions

bernoulli_nll

chainer.functions.**bernoulli_nll**(*x*, *y*)

Computes the negative log-likelihood of a Bernoulli distribution.

This function calculates the negative log-likelihood of a Bernoulli distribution.

$$-B(x;p) = -\sum_i x_i \log(p_i) + (1-x_i)\log(1-p_i),$$

where $p = \sigma(y)$, and $\sigma(\cdot)$ is a sigmoid function.

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`) – Input variable.
- **y** (`Variable`) – A variable representing the parameter of Bernoulli distribution.

Returns A variable representing negative log-likelihood.

Return type *Variable*

connectionist_temporal_classification

chainer.functions.**connectionist_temporal_classification**(*x*, *t*, *blank_symbol*, *input_length=None*, *label_length=None*)

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]*

Parameters

- **x** (*sequence of Variable*) – RNN output at each time. `x` must be a list of *Variable* s. Each element of `x`, `x[i]` is a *Variable* representing output of RNN at time `i`.
- **t** (`Variable`) – Expected label sequence.
- **blank_symbol** (*int*) – Index of blank_symbol. This value must be non-negative.
- **input_length** (`Variable`) – Length of valid sequence for each of mini batch x (optional). If input_length is skipped, It regards that all of x is valid input.
- **label_length** (`Variable`) – Length of valid sequence for each of mini batch t (optional). If label_length is skipped, It regards that all of t is valid input.

Returns A variable holding a scalar value of the CTC loss.

Return type *Variable*

Note: You need to input `x` without applying to activation functions(e.g. softmax function), because this function applies softmax functions to `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 `x`.

Note: This function supports (batch, sequence, 1-dimensional input)-data.

contrastive

chainer.functions.**contrastive**(*x0*, *x1*, *y*, *margin=1*)

Computes contrastive loss.

It takes a pair of variables and a label as inputs. The label is 1 when those two input variables are similar, or 0 when they are dissimilar. Let N and K denote mini-batchsize and the dimension of input variables, respectively. The shape of both input variables should be `(N, K)`.

$$L = \frac{1}{2N}\left(\sum_{n=1}^{N} y_n d_n^2 + (1-y_n)\max(\text{margin} - d_n, 0)^2\right)$$

where $d_n = \|\mathbf{x_0}_n - \mathbf{x_1}_n\|_2$. N denotes the mini-batch size. Input variables, x0 and x1, have N vectors, and each vector is K-dimensional. Therefore, $\mathbf{x_0}_n$ and $\mathbf{x_1}_n$ are n-th K-dimensional vectors of x0 and x1.

Parameters

- **x0** (Variable) – The first input variable. The shape should be (N, K), where N denotes the minibatch size, and K denotes the dimension of x0.
- **x1** (Variable) – The second input variable. The shape should be the same as x0.
- **y** (Variable) – Labels. All values should be 0 or 1. The shape should be `(N,)`, where N denotes the minibatch size.
- **margin** (*float*) – A parameter for contrastive loss. It should be positive value.

Returns A variable holding a scalar that is the loss value calculated by the above equation.

Return type *Variable*

Note: This cost can be used to train siamese networks. See *Learning a Similarity Metric Discriminatively, with Application to Face Verification* <http://yann.lecun.com/exdb/publis/pdf/chopra-05.pdf> for details.

cross_covariance

chainer.functions.**cross_covariance**(*y*, *z*)

Computes the sum-squared cross-covariance penalty between `y` and `z`

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

Returns A variable holding a scalar of the cross covariance loss.

Return type *Variable*

Note: This cost can be used to disentangle variables. See http://arxiv.org/abs/1412.6583v3 for details.

---

gaussian_kl_divergence

chainer.functions.**gaussian_kl_divergence**(*mean*, *ln_var*)

Computes the KL-divergence of Gaussian variables from the standard one.

Given two variable `mean` representing μ and `ln_var` representing $\log(\sigma^2)$, this function returns a variable representing the KL-divergence between the given multi-dimensional Gaussian $N(\mu, S)$ and the standard Gaussian $N(0, I)$

$$D_{\mathbf{KL}}(N(\mu, S) \| N(0, I)),$$

where S is a diagonal matrix such that $S_{ii} = \sigma_i^2$ and I is an identity matrix.

Parameters

- **mean** (Variable) – A variable representing mean of given gaussian distribution, μ.
- **ln_var** (Variable) – A variable representing logarithm of variance of given gaussian distribution, $\log(\sigma^2)$.

Returns A variable representing KL-divergence between given gaussian distribution and the standard gaussian.

Return type *Variable*

gaussian_nll

chainer.functions.**gaussian_nll**(*x*, *mean*, *ln_var*)

Computes the negative log-likelihood of a Gaussian distribution.

Given two variable `mean` representing μ and `ln_var` representing $\log(\sigma^2)$, this function returns 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 $S_{ii} = \sigma_i^2$.

Parameters

- **x** (Variable) – Input variable.
- **mean** (Variable) – A variable representing mean of a Gaussian distribution, μ.
- **ln_var** (Variable) – A variable representing logarithm of variance of a Gaussian distribution, $\log(\sigma^2)$.

Returns A variable representing the negative log-likelihood.

Return type *Variable*

hinge

chainer.functions.**hinge**(*x*, *t*, *norm='L1'*)

Computes the hinge loss for a one-of-many classification task.

$$L = \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}[\max(0, 1 - \delta\{l_n = k\}t_{nk})]^p$$

where N denotes the batchsize, K is the number of classes of interest,

$$\delta\{\text{condition}\} = \begin{cases} 1 & \text{if condition} \\ -1 & \text{otherwise,} \end{cases}$$

and

$$p = \begin{cases} 1 & \text{if norm} = \text{'L1'} \\ 2 & \text{if norm} = \text{'L2'.} \end{cases}$$

Parameters

- **x** (Variable) – Input variable. The shape of x should be (N, K).
- **t** (Variable) – The N-dimensional label vector l with values $l_n \in \{0, 1, 2, \ldots, K-1\}$. The shape of t should be (N,).
- **norm** (*string*) – Specifies norm type. Only either 'L1' or 'L2' is acceptable.

Returns A variable object holding a scalar array of the hinge loss L.

Return type *Variable*

huber_loss

chainer.functions.**huber_loss**(*x*, *t*, *delta*)

Loss function which is less sensitive to outliers in data than MSE.

$$a = x - t$$

and

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{if } |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta) & \text{otherwise,} \end{cases}$$

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.

Returns A variable object holding a scalar array of the huber loss L_δ.

Return type *Variable*

See: Huber loss - Wikipedia.

mean_squared_error

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.

negative_sampling

chainer.functions.**negative_sampling**(*x*, *t*, *W*, *sampler*, *sample_size*)
Negative sampling loss function.

In natural language processing, especially language modeling, the number of vocabulary is very large. Therefore, you need to spend a lot of time to calculate the gradient of the embedding matrix.

Instead, in negative sampling trick, you only need to calculate the gradient for a few sampled negative examples.

The objective function is below:

$$f(x, p) = \log \sigma(x^\top w_p) + \ kE_{i \sim P(i)}[\log \sigma(-x^\top w_i)],$$

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:

$$f(x, p) \approx \log \sigma(x^\top w_p) + \sum_{n \in N} \log \sigma(-x^\top w_n).$$

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, α 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.
- **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.
- **sample_size** (*int*) – Number of samples.

See: Distributed Representations of Words and Phrases and their Compositionality

See also:

NegativeSampling.

sigmoid_cross_entropy

chainer.functions.**sigmoid_cross_entropy**(*x*, *t*, *use_cudnn=True*, *normalize=True*)
Computes cross entropy loss for 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.

Returns A variable object holding a scalar array of the cross entropy loss.

Return type *Variable*

Note: This function is differentiable only by `x`.

softmax_cross_entropy

chainer.functions.**softmax_cross_entropy**(*x*, *t*, *use_cudnn=True*, *normalize=True*, *cache_score=True*)

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] == -1`, corresponding `x[i]` is ignored.
- **normalize** (Variable) – 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.
- **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.

Returns A variable holding a scalar array of the cross entropy loss.

Return type *Variable*

Note: This function is differentiable only by `x`.

3.4.6 Mathematical 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 determinant 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*

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*

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 $N \times 1$ matrices. A 3-D array of shape `(B, M, N)` is considered as B $M \times N$ 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*

clip

chainer.functions.**clip**(*x*, *x_min*, *x_max*)

Clips (limits) elements of input variable.

Given an interval `[x_min, xmax]`, 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*

cos

chainer.functions.**cos**(*x*)
Elementwise cos function.

exp

chainer.functions.**exp**(*x*)
Elementwise exponential function.

identity

chainer.functions.**identity**(**inputs*)
Just returns input variables.

inv

chainer.functions.**inv**(*a*)
Computes the inverse of of square matrix.

Parameters **a** (Variable) – Input array to compute the determinant 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*

log

chainer.functions.**log**(*x*)
Elementwise natural logarithm function.

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*

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*

min

chainer.functions.**min**(*x*, *axis=None*, *keepdims=False*)
Minimum of array elements over a given axis.

Parameters

- **x** (Variable) – Array to be minimized.
- **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.

Returns Output variable.

Return type *Variable*

sin

chainer.functions.**sin**(*x*)
Elementwise sin function.

sum

chainer.functions.**sum**(*x*, *axis=None*)
Sum of array elements over a given axis.

Parameters

- **x** (Variable) – Elements to sum.
- **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*

3.4.7 Noise injections

dropout

chainer.functions.**dropout**(*x*, *ratio=0.5*, *train=True*)

Drops elements of input variable randomly.

This function drops input elements randomly with probability `ratio` and scales the remaining elements by factor `1 / (1 - ratio)`. In testing mode, it does nothing and just returns `x`.

Parameters

- **x** (`Variable`) – Input variable.
- **ratio** (*float*) – Dropout ratio.
- **train** (*bool*) – If `True`, executes dropout. Otherwise, does nothing.

Returns Output variable.

Return type *Variable*

See the paper by G. Hinton: Improving neural networks by preventing co-adaptation of feature detectors.

gaussian

chainer.functions.**gaussian**(*mean*, *ln_var*)

Gaussian sampling function.

It takes mean μ and logarithm of variance $\log(\sigma^2)$ as input and output a sample drawn from gaussian $N(\mu, \sigma)$.

Parameters

- **mean** (`Variable`) – Input variable representing mean μ.
- **ln_var** (`Variable`) – Input variable representing logarithm of variance $\log(\sigma^2)$.

Returns Output variable.

Return type *Variable*

3.4.8 Normalization functions

batch_normalization

chainer.functions.**batch_normalization**(*x*, *gamma*, *beta*, *eps=1e-05*)

Batch normalization function.

It takes the input variable `x` and two parameter variables `gamma` and `beta`. The input must have the batch size and the features (or channels) as the first two dimensions of its shape. The input can have more than two dimensions, where the remained dimensions are considered as spatial dimensions, which are considered as a part of the batch size.

Parameters

- **x** (`Variable`) – The input variable.
- **gamma** (`Variable`) – The scaling parameter of normalized data.
- **beta** (`Variable`) – The shifting parameter of scaled normalized data.
- **eps** (*float*) – Epsilon value for numerical stability.

See: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

See also:

`links.BatchNormalization`

fixed_batch_normalization

`chainer.functions.`**`fixed_batch_normalization`**(*x*, *gamma*, *beta*, *mean*, *var*, *eps=1e-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 variables. This is used on testing mode of the batch normalization layer, where batch statistics cannot be used for prediction consistency.

Parameters

- **x** (`Variable`) – The input variable.
- **gamma** (`Variable`) – The scaling parameter of normalized data.
- **beta** (`Variable`) – The shifting parameter of scaled normalized data.
- **mean** (`Variable`) – The shifting parameter of input.
- **var** (`Variable`) – The square of scaling parameter of input.
- **eps** (*float*) – Epsilon value for numerical stability.

See also:

`functions.batch_normalization()`, `links.BatchNormalization`

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

3.4.9 Spatial pooling

average_pooling_2d

chainer.functions.**average_pooling_2d**(*x*, *ksize*, *stride=None*, *pad=0*, *use_cudnn=True*)

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.
- **use_cudnn** (*bool*) – If True and cuDNN is enabled, then this function uses cuDNN as the core implementation.

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.

---

max_pooling_2d

chainer.functions.**max_pooling_2d**(*x*, *ksize*, *stride=None*, *pad=0*, *cover_all=True*, *use_cudnn=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.

- **use_cudnn** (*bool*) – If `True` and cuDNN is enabled, then this function uses cuDNN as the core implementation.

Returns Output variable.

Return type *Variable*

spatial_pyramid_pooling_2d

chainer.functions.**spatial_pyramid_pooling_2d**(*x*, *pyramid_height*, *pooling_class*, *use_cudnn=True*)

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:

$$
\begin{aligned}
k_h^{(i)} &= \lceil b_h / 2^i \rceil, \\
k_w^{(i)} &= \lceil b_w / 2^i \rceil, \\
p_h^{(i)} &= (2^i k_h^{(i)} - b_h)/2, \\
p_w^{(i)} &= (2^i k_w^{(i)} - b_w)/2,
\end{aligned}
$$

where $\lceil \cdot \rceil$ 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.

See detail in paper: Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition.

Parameters

- **x** (*Variable*) – Input variable. The shape of `x` should be `(batchsize, # of channels, height, width)`.
- **pyramid_height** (*int*) – the number of pyramid levels
- **pooling_class** (*MaxPooling2D or AveragePooling2D*) – Only MaxPooling2D class can be available for now.
- **use_cudnn** (*bool*) – If `True` and cuDNN is enabled, then this function uses cuDNN as the core implementation.

Returns Output variable. The shape of the output variable will be $(batchsize, c \sum_{h=0}^{H-1} 2^{2h}, 1, 1)$, 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.

---

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`, all spatial locations are pooled into some output pixels, and the output size is larger than that when cover_all is `False`.

Returns Output variable.

Return type *Variable*

3.5 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.

3.5.1 Learnable connections

Bilinear

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 *bilinear()* functions. It holds parameters `W`, `V1`, `V2`, and `b` corresponding to the arguments of *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 `V1`, `V2`, and `b` are omitted.
- **initialW** (*3-D numpy array*) – Initial value of W. Shape of this argument must be `(left_size, right_size, out_size)`. If `None`, W is initialized by centered Gaussian distribution properly scaled according to the dimension of inputs and outputs.
- **initial_bias** (*tuple*) – Initial values of V^1, V^2 and b. The length this argument must be 3. Each element of this tuple must have the shapes of `(left_size, output_size)`, `(right_size, output_size)`, and `(output_size,)`, respectively. If `None`, V^1 and V^2 is initialized by scaled centered Gaussian distributions and b is set to 0.

See also:

See *chainer.functions.bilinear()* for details.

Variables

- **W** (`Variable`) – Bilinear weight parameter.
- **V1** (`Variable`) – Linear weight parameter for the first argument.
- **V2** (`Variable`) – Linear weight parameter for the second argument.
- **b** (`Variable`) – Bias parameter.

__call__(*e1*, *e2*)

Applies the bilinear function to inputs and the internal parameters.

Parameters

- **e1** (`Variable`) – Left input.
- **e2** (`Variable`) – Right input.

Returns Output variable.

Return type *Variable*

Convolution2D

class chainer.links.**Convolution2D**(*in_channels*, *out_channels*, *ksize*, *stride=1*, *pad=0*, *wscale=1*, *bias=0*, *nobias=False*, *use_cudnn=True*, *initialW=None*, *initial_bias=None*)

Two-dimensional convolutional layer.

This link wraps the *convolution_2d()* function and holds the filter weight and bias vector as parameters.

Parameters

- **in_channels** (*int*) – Number of channels of input arrays.
- **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.
- **wscale** (*float*) – Scaling factor of the initial weight.

- **bias** (*float*) – Initial bias value.
- **nobias** (*bool*) – If `True`, then this link does not use the bias term.
- **use_cudnn** (*bool*) – If `True`, then this link uses cuDNN if available.
- **initialW** (*4-D array*) – Initial weight value. If `None`, then this function uses to initialize `wscale`.
- **initial_bias** (*1-D array*) – Initial bias value. If `None`, then this function uses to initialize `bias`.

See also:

See *chainer.functions.convolution_2d()* for the definition of two-dimensional convolution.

Variables

- **W** (`Variable`) – Weight parameter.
- **b** (`Variable`) – Bias parameter.

__call__(*x*)

Applies the convolution layer.

Parameters **x** (`Variable`) – Input image.

Returns Output of the convolution.

Return type *Variable*

Deconvolution2D

class chainer.links.**Deconvolution2D**(*in_channels*, *out_channels*, *ksize*, *stride=1*, *pad=0*, *wscale=1*, *bias=0*, *nobias=False*, *outsize=None*, *use_cudnn=True*, *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.

Parameters

- **in_channels** (*int*) – Number of channels of input arrays.
- **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.
- **wscale** (*float*) – Scaling factor of the initial weight.
- **bias** (*float*) – Initial bias value.
- **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.

- **use_cudnn** (*bool*) – If `True`, then this function uses cuDNN if available.
- **initialW** (*4-D array*) – Initial weight value. If `None`, then this function uses to initialize `wscale`.
- **initial_bias** (*1-D array*) – Initial bias value. If `None`, then this function uses to initialize `bias`.

The filter weight has four dimensions (c_I, c_O, k_H, k_W) which indicate the number of 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 deviation is scaled by `wscale` if specified.

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.

See also:

See *chainer.functions.deconvolution_2d()* for the definition of two-dimensional convolution.

EmbedID

class `chainer.links.`**EmbedID**(*in_size*, *out_size*, *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.
- **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.

__call__(*x*)

Extracts the word embedding of given IDs.

Parameters **x** (`Variable`) – Batch vectors of IDs.

Returns Batch of corresponding embeddings.

Return type *Variable*

GRU

class `chainer.links.`**GRU**(*n_units*, *n_inputs=None*)

Stateless Gated Recurrent Unit function (GRU).

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 two inputs a previous hidden vector h and an input vector x, GRU returns the next hidden vector h' defined as

$$
\begin{aligned}
r &= \sigma(W_r x + U_r h), \\
z &= \sigma(W_z x + U_z h), \\
\bar{h} &= \tanh(W x + U(r \odot h)), \\
h' &= (1 - z) \odot h + z \odot \bar{h},
\end{aligned}
$$

where σ is the sigmoid function, and $\odot$ is the element-wise product.

GRU does not hold the value of hidden vector h. So this is *stateless*. Use *StatefulGRU* as a *stateful* GRU.

Parameters

- **n_units** (*int*) – Dimension of hidden vector h.
- **n_inputs** (*int*) – Dimension of input vector x. If `None`, it is set to the same value as `n_units`.

See:

- On the Properties of Neural Machine Translation: Encoder-Decoder Approaches [Cho+, SSST2014].
- Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling [Chung+NIPS2014 DLWorkshop].

See also:

StatefulGRU

Inception

class `chainer.links.`**`Inception`**(*in_channels*, *out1*, *proj3*, *out3*, *proj5*, *out5*, *proj_pool*)

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.

Parameters

- **in_channels** (*int*) – Number of channels of input arrays.
- **out1** (*int*) – Output size of 1x1 convolution path.
- **proj3** (*int*) – Projection size of 3x3 convolution path.
- **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.

__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*

InceptionBN

class chainer.links.**InceptionBN**(*in_channels*, *out1*, *proj3*, *out3*, *proj33*, *out33*, *pooltype*, *proj_pool=None*, *stride=1*)

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*) – 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** (*bool*) – If `True`, do projection in the pooling path.
- **stride** (*int*) – Stride parameter of the last convolution of each path.

See also:

Inception

Variables **train** (*bool*) – If `True`, then batch normalization layers are used in training mode. If `False`, they are used in testing mode.

Linear

class chainer.links.**Linear**(*in_size*, *out_size*, *wscale=1*, *bias=0*, *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 $\sqrt{1/}$

Parameters
- **in_size** (*int*) – Dimension of input vectors.
- **out_size** (*int*) – Dimension of output vectors.
- **wscale** (*float*) – Scaling factor of the weight matrix.
- **bias** (*float*) – Initial bias value.
- **nobias** (*bool*) – If `True`, then this function does not use the bias.
- **initialW** (*2-D array*) – Initial weight value. If `None`, then this function uses to initialize `wscale`.
- **initial_bias** (*1-D array*) – Initial bias value. If `None`, then this function uses to initialize `bias`.

See also:

linear()

Variables
- **W** (`Variable`) – Weight parameter.
- **b** (`Variable`) – Bias parameter.

__call__(*x*)
Applies the linear layer.

Parameters **x** (`Variable`) – Batch of input vectors.

Returns Output of the linear layer.

Return type *Variable*

LSTM

class chainer.links.**LSTM**(*in_size*, *out_size*)
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*.

Parameters
- **in_size** (*int*) – Dimensionality of input vectors.
- **out_size** (*int*) – Dimensionality of output vectors.

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.

__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*

reset_state()
Resets the internal state.

It sets None to the c and h attributes.

MLPConvolution2D

class chainer.links.**MLPConvolution2D**(*in_channels*, *out_channels*, *ksize*, *stride=1*, *pad=0*, *wscale=1*, *activation=<function relu>*, *use_cudnn=True*)
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*) – Number of channels of input arrays.
- **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.
- **use_cudnn** (*bool*) – If True, then this link uses cuDNN if available.

See: *Network in Network <http://arxiv.org/abs/1312.4400v3>*.

Variables **activation** (function) – Activation function.

__call__(*x*)
Computes the output of the mlpconv layer.

Parameters **x** (Variable) – Input image.

Returns Output of the mlpconv layer.

Return type *Variable*

StatefulGRU

class chainer.links.**StatefulGRU**(*in_size*, *out_size*)
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

$$
\begin{aligned}
r &= \sigma(W_r x + U_r h), \\
z &= \sigma(W_z x + U_z h), \\
\bar{h} &= \tanh(W x + U(r \odot h)), \\
h' &= (1 - z) \odot h + z \odot \bar{h},
\end{aligned}
$$

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.

Variables **h** (Variable) – Hidden vector that indicates the state of `StatefulGRU`.

See also:

GRU

3.5.2 Activation/loss/normalization functions with parameters

BatchNormalization

class chainer.links.**BatchNormalization**(*size*, *decay=0.9*, *eps=1e-05*, *dtype=<type 'numpy.float32'>*)

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.

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** (Variable) – Population mean.
- **avg_var** (Variable) – 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.

__call__(*x*, *test=False*, *finetune=False*)

Invokes the forward propagation of BatchNormalization.

BatchNormalization accepts additional arguments, which controls three different running mode.

Parameters

- **x** (Variable) – An input variable.
- **test** (*bool*) – If `True`, BatchNormalization runs in testing mode; it normalizes the input using pre-computed statistics.
- **finetune** (*bool*) – If `True`, BatchNormalization runs in fine-tuning mode; it accumulates the input array to compute population statistics for normalization, and normalizes the input using batch statistics.

If `test` and `finetune` are both `False`, then BatchNormalization runs in training mode; it computes moving averages of mean and variance for evaluation during training, and normalizes the input using batch statistics.

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.

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 $\mathrm{path}(x) =$

$((e_1, b_1), \ldots, (e_m, b_m))$ 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].

__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*

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_coutns`.

PReLU

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:

chainer.functions.prelu()

Variables **W** (Variable) – Coefficient of parametric ReLU.

__call__(*x*)
Applies the parametric ReLU activation function.

Parameters **x** (Variable) – Input variable.

Returns Output of the parametric ReLU function.

Return type *Variable*

Maxout

class chainer.links.**Maxout**(*in_size*, *out_size*, *pool_size*, *wscale=1*, *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 * P, N)` weight matrix and an optional `M * 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.
- **wscale** (*float*) – Scaling factor of the weight matrix.
- **initialW** (*3-D array or None*) – Initial weight value. If None, then this function uses wscale to initialize.
- **initial_bias** (*2-D array, float or None*) – Initial bias value. If it is float, initial bias is filled with this value. If it is None, bias is omitted.

Variables ***linear*** (Link) – The Linear link that performs affine transformation.

See also:

maxout()

See also:

Goodfellow, I., Warde-farley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). Maxout Networks. In Proceedings of the 30th International Conference on Machine Learning (ICML-13) (pp. 1319-1327). URL

__call__(*x*)
Applies the maxout layer.

Parameters **x** (Variable) – Batch of input vectors.

Returns Output of the maxout layer.

Return type *Variable*

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 α.

See also:

negative_sampling() for more detail.

Variables **W** (Variable) – Weight parameter matrix.

__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*

3.5.3 Machine learning models

Classifier

class chainer.links.**Classifier**(*predictor*, *lossfun=<function softmax_cross_entropy>*)
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.

Variables

- **predictor** (Link) – Predictor network.
- **lossfun** (function) – Loss function.
- **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`.

__call__(*x*, *t*)
Computes the loss value for an input and label pair.

It also computes accuracy and stores it to the attribute.

Parameters

- **x** (`Variable`) – Input minibatch.
- **t** (`Variable`) – Corresponding ground truth labels.

Returns Loss value.

Return type *Variable*

3.5.4 Deprecated links

Parameter

class chainer.links.**Parameter**(*array*)
Link that just holds a parameter and returns it.

Deprecated since version v1.5: The parameters are stored as variables as of v1.5. Use them directly instead.

Parameters **array** – Initial parameter array.

Variables **W** (`Variable`) – Parameter variable.

__call__(*volatile='off'*)
Returns the parameter variable.

Parameters **volatile** (`Flag`) – The volatility of the returned variable.

Returns A copy of the parameter variable with given volatility.

Return type *Variable*

3.6 Optimizers

class chainer.optimizers.**AdaDelta**(*rho=0.95*, *eps=1e-06*)
Zeiler's ADADELTA.

See: http://www.matthewzeiler.com/pubs/googleTR2012/googleTR2012.pdf

class chainer.optimizers.**AdaGrad**(*lr=0.001*, *eps=1e-08*)
AdaGrad implementation.

See: http://jmlr.org/papers/v12/duchi11a.html

class chainer.optimizers.**Adam**(*alpha=0.001*, *beta1=0.9*, *beta2=0.999*, *eps=1e-08*)
Adam optimization algorithm.

See: http://arxiv.org/abs/1412.6980v8

class chainer.optimizers.**MomentumSGD**(*lr=0.01*, *momentum=0.9*)
Classical momentum SGD.

class chainer.optimizers.**NesterovAG**(*lr=0.01*, *momentum=0.9*)
Nesterov's Accelerated Gradient.

Formulated as the linear combination coefficients of the velocity and gradient contributions at each iteration.

See: http://arxiv.org/abs/1212.0901

class chainer.optimizers.**RMSprop**(*lr=0.01*, *alpha=0.99*, *eps=1e-08*)
Hinton's RMSprop.

class chainer.optimizers.**RMSpropGraves**(*lr=0.0001*, *alpha=0.95*, *momentum=0.9*, *eps=0.0001*)
Alex Graves's RMSprop.

See http://arxiv.org/abs/1308.0850

class chainer.optimizers.**SGD**(*lr=0.01*)
Vanilla Stochastic Gradient Descent.

3.7 Serializers

3.7.1 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.

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.

class chainer.serializers.**NpzDeserializer**(*npz*, *path=''*)
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.

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**(*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.

3.7.2 Serialization in HDF5 format

class chainer.serializers.**HDF5Serializer**(*group*, *compression=4*)
Serializer for HDF5 format.

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.

class chainer.serializers.**HDF5Deserializer**(*group*)
Deserializer for HDF5 format.

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.

chainer.serializers.**save_hdf5**(*filename*, *obj*, *compression=4*)
Saves an object to the file in HDF5 format.

This is a short-cut function to save only one object into an HDF5 file. If you want to save multiple objects to one HDF5 file, use *HDF5Serializer* directly by passing appropriate `h5py.Group` objects.

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**(*filename*, *obj*)
Loads an object from the file in HDF5 format.

This is a short-cut function to load from an HDF5 file that contains only one object. If you want to load multiple objects from one HDF5 file, use *HDF5Deserializer* directly by passing appropriate `h5py.Group` objects.

Parameters

- **filename** (*str*) – Name of the file to be loaded.
- **obj** – Object to be deserialized. It must support serialization protocol.

3.8 Function hooks

Chainer provides a function-hook mechanism that enriches the behavior of forward and backward propagation of *Function*.

3.8.1 Base class

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.

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*.

```
>>> import chainer, chainer.links as L, chainer.functions as F
... class Model(chainer.Chain):
...     def __call__(x):
...         x = self.l(x)
...         return F.exp(x)
... model1 = Model(l=L.Linear(10, 10))
... model2 = Model(l=L.Linear(10, 10))
... x = chainer.Variable(numpy.zeros((1, 10), 'f'))
... with chainer.function_hooks.TimerHook() as m:
...     y = model1(x)
...     y = model2(x)
...     print(m.total_time())
... model3 = Model(l=L.Linear(10, 10))
... z = model3(y)
```

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.

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.

3.8.2 Concrete function hooks

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).

The basic usage is to use it with `with` statement.

```
>>> import chainer, chainer.functions as F, chainer.links as L
... l = L.Linear(10, 10)
... x = chainer.Variable(numpy.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 propagations of `LinearFunction` (which is implicitly called by `l`) and `Sum` (called by `F.sum`) and backward propagations of `z` and `y`.

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.

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.

total_time()

Returns total elapsed time in seconds.

3.9 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 *Function* implementations. This functionality is provided by the *chainer.functions.caffe.CaffeFunction* class.

class `chainer.functions.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: This class only supports Python 2.7, since the compiled module for protocol buffers only supports Python 2. The `__init__` function raises an exception in 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:

```
# 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

- **fs** (FunctionSet) – A set of functions corresponding to parameterized layers of Caffe. The names of its attributes are same as the layer names of the given network.
- **forwards** (*dict*) – A mapping from layer names to corresponding functions.

__call__(*inputs*, *outputs*, *disable=()*, *train=True*)

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.

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.
- **train** (*bool*) – If `True`, this function emulates the TRAIN phase of the Caffe layers. Otherwise, it emulates the TEST phase.

Returns A tuple of output *Variable* objects corresponding to elements of the *outputs* argument.

Return type tuple

3.10 Visualization of Computational Graph

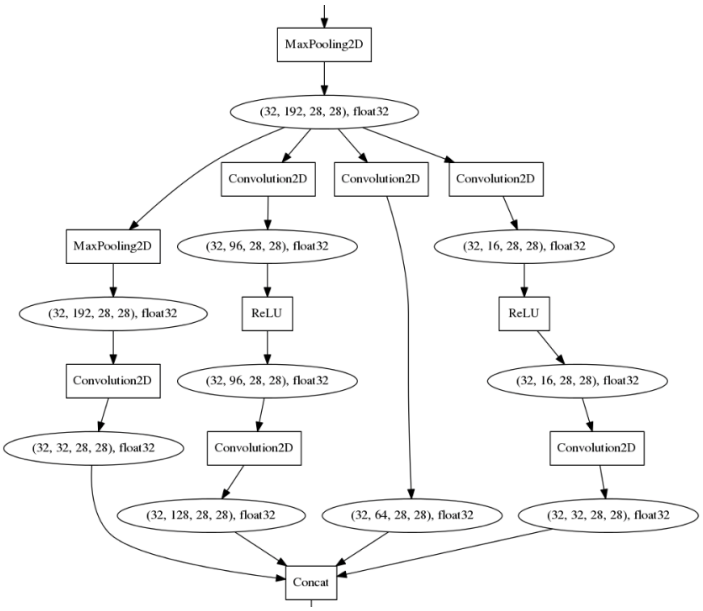
As neural networks get larger and complicated, it gets much harder to confirm if their architectures are constructed properly. Chainer supports visualization of computational graphs. Users can generate computational graphs by invoking *build_computational_graph()*. Generated computational graphs are dumped to specified format (Currently Dot Language is supported).

Basic usage is as follows:

```
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`.



chainer.computational_graph.**build_computational_graph**(*outputs*, *remove_split=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 or `Function` object.
- **remove_split** (*bool*) – It must be `True`. This argument is left for backward compatibility.

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
x --+
    |--> 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*

class `chainer.computational_graph.`**ComputationalGraph**(*nodes*, *edges*)

Class that represents computational graph.

Note: We assume that the computational graph is directed and acyclic.

dump(*format='dot'*)

Dumps graph as a text.

Args format(str): The graph language name of the output. Currently, it must be 'dot'.

Returns str: The graph in specified format.

CHAPTER 4

CuPy Reference Manual

This is the official documentation of CuPy, a multi-dimensional array on CUDA with a subset of NumPy interface.

4.1 CuPy Overview

CuPy is an implementation of NumPy-compatible multi-dimensional array on CUDA. CuPy consists of the core multi-dimensional array class, *cupy.ndarray*, and many functions on it. It supports a subset of `numpy.ndarray` interface that is enough for Chainer.

The following is a brief overview of supported subset of NumPy interface:

- Basic indexing (indexing by ints, slices, newaxes, and Ellipsis)
- Element types (dtypes): bool_, (u)int{8, 16, 32, 64}, float{16, 32, 64}
- Most of the array creation routines
- Reshaping and transposition
- All operators with broadcasting
- All Universal functions (a.k.a. ufuncs) for elementwise operations except those for complex numbers
- Dot product functions (except einsum) using cuBLAS
- Reduction along axes (sum, max, argmax, etc.)

CuPy also includes following features for performance:

- Customizable memory allocator, and a simple memory pool as an example
- User-defined elementwise kernels
- User-defined reduction kernels
- cuDNN utilities

CuPy uses on-the-fly kernel synthesis: when a kernel call is required, it compiles a kernel code optimized for the shapes and dtypes of given arguments, sends it to the GPU device, and executes the kernel. The compiled code is cached to `$(HOME)/.cupy/kernel_cache` directory (this cache path can be overwritten by setting the `CUPY_CACHE_DIR` environment variable). It may make things slower at the first kernel call, though this slow down will be resolved at the second execution. CuPy also caches the kernel code sent to GPU device within the process, which reduces the kernel transfer time on further calls.

4.1.1 A list of supported attributes, properties, and methods of ndarray

Memory layout

base ctypes *itemsize* *flags* *nbytes* *shape* size *strides*

Data type

dtype

Other attributes

T

Array conversion

tolist() *tofile()* *dump()* *dumps()* *astype()* *copy()* *view()* *fill()*

Shape manipulation

reshape() *transpose()* *swapaxes()* *ravel()* *squeeze()*

Item selection and manipulation

take() *diagonal()*

Calculation

max() *argmax()* *min()* *argmin()* *clip()* *trace()* *sum()* *mean()* *var()* *std()* *prod()* *dot()*

Arithmetic and comparison operations

__lt__() *__le__()* *__gt__()* *__ge__()* *__eq__()* *__ne__()* *__nonzero__()* *__neg__()* *__pos__()* *__abs__()* *__invert__()* *__add__()* *__sub__()* *__mul__()* *__div__()* *__truediv__()* *__floordiv__()* *__mod__()* *__divmod__()* *__pow__()* *__lshift__()* *__rshift__()* *__and__()* *__or__()* *__xor__()* *__iadd__()* *__isub__()* *__imul__()* *__idiv__()* *__itruediv__()* *__ifloordiv__()* *__imod__()* *__ipow__()* *__ilshift__()* *__irshift__()* *__iand__()* *__ior__()* *__ixor__()*

Special methods

__copy__() __deepcopy__() __reduce__() __array__() *__len__()* *__getitem__()* *__setitem__()* *__int__()* *__long__()* *__float__()* *__oct__()* *__hex__()* *__repr__()* *__str__()*

Memory transfer

get() *set()*

4.1.2 A list of supported routines of cupy module

Array creation routines

empty() empty_like() eye() identity() ones() ones_like() zeros() zeros_like() full() full_like()

array() asarray() ascontiguousarray() copy()

arange() linspace()

diag() diagflat()

Array manipulation routines

copyto()

reshape() ravel()

rollaxis() swapaxes() transpose()

atleast_1d() atleast_2d() atleast_3d() broadcast broadcast_arrays() broadcast_to() expand_dims() squeeze()

column_stack() concatenate() dstack() hstack() vstack()

array_split() dsplit() hsplit() split() vsplit()

roll()

Binary operations

bitwise_and bitwise_or bitwise_xor invert left_shift right_shift

Indexing routines

take() diagonal()

Input and output

load() save() savez() savez_compressed()

array_repr() array_str()

Linear algebra

dot() vdot() inner() outer() tensordot()

trace()

Logic functions

isfinite isinf isnan

logical_and logical_or logical_not logical_xor

greater greater_equal less less_equal equal not_equal

Mathematical functions

sin cos tan arcsin arccos arctan hypot arctan2 deg2rad rad2deg degrees radians

sinh cosh tanh arcsinh arccosh arctanh

rint floor ceil trunc

sum() prod()

exp expm1 exp2 log log10 log2 log1p logaddexp logaddexp2

signbit copysign ldexp frexp nextafter

add reciprocal negative multiply divide power subtract true_divide floor_divide fmod mod modf remainder

clip() sqrt square absolute sign maximum minimum fmax fmin

Sorting, searching, and counting

argmax() argmin() count_nonzero() where()

Statistics

amin() amax()

mean() var() std()

bincount()

Other

asnumpy()

4.2 Multi-Dimensional Array (ndarray)

class cupy.**ndarray**

Multi-dimensional array on a CUDA device.

This class implements a subset of methods of numpy.ndarray. The difference is that this class allocates the array content on the current GPU device.

Parameters

- **shape** (*tuple of ints*) – Length of axes.
- **dtype** – Data type. It must be an argument of numpy.dtype.

- **memptr** (cupy.cuda.MemoryPointer) – Pointer to the array content head.
- **strides** (*tuple of ints*) – The strides for axes.

Variables

- **base** (*None or cupy.ndarray*) – Base array from which this array is created as a view.
- **data** (cupy.cuda.MemoryPointer) – Pointer to the array content head.
- **dtype** (*numpy.dtype*) – Dtype object of element type.

  See also:

  Data type objects (dtype)
- ***size*** (*int*) – Number of elements this array holds.

  This is equivalent to product over the shape tuple.

  See also:

  numpy.ndarray.size

T

Shape-reversed view of the array.

If ndim < 2, then this is just a reference to the array itself.

__abs__

__add__
x.__add__(y) <==> x+y

__and__
x.__and__(y) <==> x&y

__delitem__
x.__delitem__(y) <==> del x[y]

__div__
x.__div__(y) <==> x/y

__divmod__

__eq__
x.__eq__(y) <==> x==y

__float__

__floordiv__
x.__floordiv__(y) <==> x//y

__ge__
x.__ge__(y) <==> x>=y

__getitem__
x.__getitem__(y) <==> x[y]

__gt__
x.__gt__(y) <==> x>y

__hex__

__iadd__
x.__iadd__(y) <==> x+=y

__iand__
x.__iand__(y) <==> x&=y

__idiv__
x.__idiv__(y) <==> x/=y

__ifloordiv__
x.__ifloordiv__(y) <==> x//y

__ilshift__
x.__ilshift__(y) <==> x<<=y

__imod__
x.__imod__(y) <==> x%=y

__imul__
x.__imul__(y) <==> x*=y

__int__

__invert__
x.__invert__() <==> ~x

__ior__
x.__ior__(y) <==> x|=y

__ipow__
x.__ipow__(y) <==> x**=y

__irshift__
x.__irshift__(y) <==> x>>=y

__isub__
x.__isub__(y) <==> x-=y

__itruediv__
x.__itruediv__(y) <==> x/y

__ixor__
x.__ixor__(y) <==> x^=y

__le__
x.__le__(y) <==> x<=y

__len__

__long__

__lshift__
x.__lshift__(y) <==> x<<y

__lt__
x.__lt__(y) <==> x<y

__mod__
x.__mod__(y) <==> x%y

__mul__
x.__mul__(y) <==> x*y

__ne__
x.__ne__(y) <==> x!=y

__neg__
x.__neg__() <==> -x

__nonzero__
x.__nonzero__() <==> x != 0

__oct__

__or__
x.__or__(y) <==> x|y

__pos__
x.__pos__() <==> +x

__pow__

__radd__
x.__radd__(y) <==> y+x

__rand__
x.__rand__(y) <==> y&x

__rdiv__
x.__rdiv__(y) <==> y/x

__rdivmod__

__repr__

__rfloordiv__
x.__rfloordiv__(y) <==> y//x

__rlshift__
x.__rlshift__(y) <==> y<<x

__rmod__
x.__rmod__(y) <==> y%x

__rmul__
x.__rmul__(y) <==> y*x

__ror__
x.__ror__(y) <==> y|x

__rpow__

__rrshift__
x.__rrshift__(y) <==> y>>x

__rshift__
x.__rshift__(y) <==> x>>y

__rsub__
x.__rsub__(y) <==> y-x

__rtruediv__
x.__rtruediv__(y) <==> y/x

__rxor__
x.__rxor__(y) <==> y^x

__setitem__
x.__setitem__(i, y) <==> x[i]=y

__str__

__sub__
x.__sub__(y) <==> x-y

__truediv__
x.__truediv__(y) <==> x/y

__xor__
x.__xor__(y) <==> x^y

argmax()
Returns the indices of the maximum along a given axis.

See also:

cupy.argmax() for full documentation, numpy.ndarray.argmax()

argmin()
Returns the indices of the minimum along a given axis.

See also:

cupy.argmin() for full documentation, numpy.ndarray.argmin()

astype()
Casts the array to given data type.

Parameters

- **dtype** – Type specifier.
- **copy** (*bool*) – If it is False and no cast happens, then this method returns the array itself. Otherwise, a copy is returned.

Returns If copy is False and no cast is required, then the array itself is returned. Otherwise, it returns a (possibly casted) copy of the array.

Note: This method currently does not support order, casting, and subok arguments.

See also:

numpy.ndarray.astype()

clip()
Returns an array with values limited to [a_min, a_max].

See also:

cupy.clip() for full documentation, numpy.ndarray.clip()

copy()
Returns a copy of the array.

See also:

cupy.copy() for full documentation, numpy.ndarray.copy()

cstruct
C representation of the array.

This property is used for sending an array to CUDA kernels. The type of returned C structure is different for different dtypes and ndims. The definition of C type is written in cupy/carray.cuh.

device
CUDA device on which this array resides.

diagonal()
Returns a view of the specified diagonals.

See also:

cupy.diagonal() for full documentation, numpy.ndarray.diagonal()

dot()
Returns the dot product with given array.

See also:

cupy.dot() for full documentation, numpy.ndarray.dot()

dump()
Dumps a pickle of the array to a file.

Dumped file can be read back to *cupy.ndarray* by *cupy.load()*.

dumps()
Dumps a pickle of the array to a string.

fill()
Fills the array with a scalar value.

Parameters **value** – A scalar value to fill the array content.

See also:

numpy.ndarray.fill()

flags
Object containing memory-layout information.

It only contains c_contiguous, f_contiguous, and owndata attributes. All of these are read-only. Accessing by indexes is also supported.

See also:

numpy.ndarray.flags

flatten()
Returns a copy of the array flatten into one dimension.

It currently supports C-order only.

Returns A copy of the array with one dimension.

Return type *cupy.ndarray*

See also:

numpy.ndarray.flatten()

get()
Returns a copy of the array on host memory.

Parameters **stream** (cupy.cuda.Stream) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

Returns Copy of the array on host memory.

Return type numpy.ndarray

itemsize
Size of each element in bytes.

See also:

numpy.ndarray.itemsize

max()
Returns the maximum along a given axis.

See also:

cupy.amax() for full documentation, numpy.ndarray.max()

mean()
Returns the mean along a given axis.

See also:

cupy.mean() for full documentation, numpy.ndarray.mean()

min()
Returns the minimum along a given axis.

See also:

cupy.amin() for full documentation, numpy.ndarray.min()

nbytes
Size of whole elements in bytes.

It does not count skips between elements.

See also:

numpy.ndarray.nbytes

ndim
Number of dimensions.

`a.ndim` is equivalent to `len(a.shape)`.

See also:

numpy.ndarray.ndim

prod()
Returns the product along a given axis.

See also:

cupy.prod() for full documentation, numpy.ndarray.prod()

ravel()
Returns an array flattened into one dimension.

See also:

cupy.ravel() for full documentation, numpy.ndarray.ravel()

reduced_view()
Returns a view of the array with minimum number of dimensions.

Parameters **dtype** – Data type specifier. If it is given, then the memory sequence is reinterpreted as the new type.

Returns A view of the array with reduced dimensions.

Return type *cupy.ndarray*

repeat()
Returns an array with repeated arrays along an axis.

See also:

cupy.repeat() for full documentation, numpy.ndarray.repeat()

reshape()
Returns an array of a different shape and the same content.

See also:

cupy.reshape() for full documentation, numpy.ndarray.reshape()

set()
Copies an array on the host memory to *cupy.ndarray*.

Parameters

- **arr** (*numpy.ndarray*) – The source array on the host memory.
- **stream** (cupy.cuda.Stream) – CUDA stream object. If it is given, the copy runs asynchronously. Otherwise, the copy is synchronous.

shape
Lengths of axes.

Setter of this property involves reshaping without copy. If the array cannot be reshaped without copy, it raises an exception.

squeeze()
Returns a view with size-one axes removed.

See also:

cupy.squeeze() for full documentation, numpy.ndarray.squeeze()

std()
Returns the standard deviation along a given axis.

See also:

cupy.std() for full documentation, numpy.ndarray.std()

strides
Strides of axes in bytes.

See also:

numpy.ndarray.strides

sum()
Returns the sum along a given axis.

See also:

cupy.sum() for full documentation, numpy.ndarray.sum()

swapaxes()
Returns a view of the array with two axes swapped.

See also:

cupy.swapaxes() for full documentation, numpy.ndarray.swapaxes()

take()
Returns an array of elements at given indices along the axis.

See also:

cupy.take() for full documentation, numpy.ndarray.take()

tofile()
Writes the array to a file.

See also:

numpy.ndarray.tolist()

tolist()
Converts the array to a (possibly nested) Python list.

Returns The possibly nested Python list of array elements.

Return type list

See also:

numpy.ndarray.tolist()

trace()
Returns the sum along diagonals of the array.

See also:

cupy.trace() for full documentation, numpy.ndarray.trace()

transpose()
Returns a view of the array with axes permuted.

See also:

cupy.transpose() for full documentation, numpy.ndarray.reshape()

var()
Returns the variance along a given axis.

See also:

cupy.var() for full documentation, numpy.ndarray.var()

view()
Returns a view of the array.

Parameters **dtype** – If this is different from the data type of the array, the returned view reinterpret the memory sequence as an array of this type.

Returns A view of the array. A reference to the original array is stored at the base attribute.

Return type *cupy.ndarray*

See also:

numpy.ndarray.view()

cupy.**asnumpy**(*a*, *stream=None*)
Returns an array on the host memory from an arbitrary source array.

Parameters

- **a** – Arbitrary object that can be converted to numpy.ndarray.
- **stream** (cupy.cuda.Stream) – CUDA stream object. If it is specified, then the device-to-host copy runs asynchronously. Otherwise, the copy is synchronous. Note that if a is not a *cupy.ndarray* object, then this argument has no effect.

Returns Converted array on the host memory.

Return type numpy.ndarray

4.3 Universal Functions (ufunc)

CuPy provides universal functions (a.k.a. ufuncs) to support various elementwise operations. CuPy's ufunc supports following features of NumPy's one:

- Broadcasting
- Output type determination
- Casting rules

CuPy's ufunc currently does not provide methods such as `reduce`, `accumulate`, `reduceat`, `outer`, and `at`.

4.3.1 Ufunc class

class `cupy.`**`ufunc`**
Universal function.

Variables

- **`name`** (*`str`*) – The name of the universal function.
- **`nin`** (*`int`*) – Number of input arguments.
- **`nout`** (*`int`*) – Number of output arguments.
- **`nargs`** (*`int`*) – Number of all arguments.

`__call__``()`
Applies the universal function to arguments elementwise.

Parameters

- **`args`** – Input arguments. Each of them can be a *`cupy.ndarray`* object or a scalar. The output arguments can be omitted or be specified by the `out` argument.
- **`out`** (`cupy.ndarray`) – Output array. It outputs to new arrays default.
- **`dtype`** – Data type specifier.

Returns Output array or a tuple of output arrays.

`types`
A list of type signatures.

Each type signature is represented by type character codes of inputs and outputs separated by '->'.

4.3.2 Available ufuncs

Math operations

`add subtract multiply divide logaddexp logaddexp2 true_divide floor_divide negative power remainder mod fmod absolute rint sign exp exp2 log log2 log10 expm1 log1p sqrt square reciprocal`

Trigonometric functions

`sin cos tan arcsin arccos arctan arctan2 hypot sinh cosh tanh arcsinh arccosh arctanh deg2rad rad2deg`

Bit-twiddling functions

bitwise_and bitwise_or bitwise_xor invert left_shift right_shift

Comparison functions

greater greater_equal less less_equal not_equal equal logical_and logical_or logical_xor logical_not maximum minimum fmax fmin

Floating point values

isfinite isinf isnan signbit copysign nextafter modf ldexp frexp fmod floor ceil trunc

4.4 Routines

The following pages describe NumPy-compatible routines. These functions cover a subset of NumPy routines.

4.4.1 Array Creation Routines

Basic creation routines

cupy.**empty**(*shape*, *dtype=<type 'float'>*)

Returns an array without initializing the elements.

This function currently does not support `order` option.

Parameters

- **shape** (*tuple of ints*) – Dimensionalities of the array.
- **dtype** – Data type specifier.

Returns A new array with elements not initialized.

Return type *cupy.ndarray*

See also:

`numpy.empty()`

cupy.**empty_like**(*a*, *dtype=None*)

Returns a new array with same shape and dtype of a given array.

This function currently does not support `order` and `subok` options.

Parameters

- **a** (`cupy.ndarray`) – Base array.
- **dtype** – Data type specifier. The data type of `a` is used by default.

Returns A new array with same shape and dtype of `a` with elements not initialized.

Return type *cupy.ndarray*

See also:

`numpy.empty_like()`

cupy.**eye**(*N*, *M=None*, *k=0*, *dtype=<type 'float'>*)
Returns a 2-D array with ones on the diagonals and zeros elsewhere.

Parameters

- **N** (*int*) – Number of rows.
- **M** (*int*) – Number of columns. M == N by default.
- **k** (*int*) – Index of the diagonal. Zero indicates the main diagonal, a positive index an upper diagonal, and a negative index a lower diagonal.
- **dtype** – Data type specifier.

Returns A 2-D array with given diagonals filled with ones and zeros elsewhere.

Return type *cupy.ndarray*

See also:

numpy.eye()

cupy.**identity**(*n*, *dtype=<type 'float'>*)
Returns a 2-D identity array.

It is equivalent to `eye(n, n, dtype)`.

Parameters

- **n** (*int*) – Number of rows and columns.
- **dtype** – Data type specifier.

Returns A 2-D identity array.

Return type *cupy.ndarray*

See also:

numpy.identity()

cupy.**ones**(*shape*, *dtype=<type 'float'>*)
Returns a new array of given shape and dtype, filled with ones.

This function currently does not support `order` option.

Parameters

- **shape** (*tuple of ints*) – Dimensionalities of the array.
- **dtype** – Data type specifier.

Returns An array filled with ones.

Return type *cupy.ndarray*

See also:

numpy.ones()

cupy.**ones_like**(*a*, *dtype=None*)
Returns an array of ones with same shape and dtype as a given array.

This function currently does not support `order` and `subok` options.

Parameters

- **a** (cupy.ndarray) – Base array.
- **dtype** – Data type specifier. The dtype of `a` is used by default.

Returns An array filled with ones.

Return type *cupy.ndarray*

See also:

`numpy.ones_like()`

`cupy.`**`zeros`**(*shape*, *dtype=<type 'float'>*)

Returns a new array of given shape and dtype, filled with zeros.

This function currently does not support `order` option.

Parameters

- **`shape`** (*`tuple of ints`*) – Dimensionalities of the array.
- **`dtype`** – Data type specifier.

Returns An array filled with ones.

Return type *cupy.ndarray*

See also:

`numpy.zeros()`

`cupy.`**`zeros_like`**(*a*, *dtype=None*)

Returns an array of zeros with same shape and dtype as a given array.

This function currently does not support `order` and `subok` options.

Parameters

- **`a`** (`cupy.ndarray`) – Base array.
- **`dtype`** – Data type specifier. The dtype of `a` is used by default.

Returns An array filled with ones.

Return type *cupy.ndarray*

See also:

`numpy.zeros_like()`

`cupy.`**`full`**(*shape*, *fill_value*, *dtype=None*)

Returns a new array of given shape and dtype, filled with a given value.

This function currently does not support `order` option.

Parameters

- **`shape`** (*`tuple of ints`*) – Dimensionalities of the array.
- **`fill_value`** – A scalar value to fill a new array.
- **`dtype`** – Data type specifier.

Returns An array filled with `fill_value`.

Return type *cupy.ndarray*

See also:

`numpy.full()`

cupy.**full_like**(*a*, *fill_value*, *dtype=None*)
Returns a full array with same shape and dtype as a given array.

This function currently does not support `order` and `subok` options.

Parameters

- **a** (cupy.ndarray) – Base array.
- **fill_value** – A scalar value to fill a new array.
- **dtype** – Data type specifier. The dtype of `a` is used by default.

Returns An array filled with `fill_value`.

Return type *cupy.ndarray*

See also:

numpy.full_like()

Creation from other data

cupy.**array**(*obj*, *dtype=None*, *copy=True*, *ndmin=0*)
Creates an array on the current device.

This function currently does not support the `order` and `subok` options.

Parameters

- **obj** – *cupy.ndarray* object or any other object that can be passed to numpy.array().
- **dtype** – Data type specifier.
- **copy** (*bool*) – If `False`, this function returns `obj` if possible. Otherwise this function always returns a new array.
- **ndmin** (*int*) – Minimum number of dimensions. Ones are inserted to the head of the shape if needed.

Returns An array on the current device.

Return type *cupy.ndarray*

See also:

numpy.array()

cupy.**asarray**(*a*, *dtype=None*)
Converts an object to array.

This is equivalent to `array(a, dtype, copy=False)`. This function currently does not support the `order` option.

Parameters

- **a** – The source object.
- **dtype** – Data type specifier. It is inferred from the input by default.

Returns An array on the current device. If `a` is already on the device, no copy is performed.

Return type *cupy.ndarray*

See also:

numpy.asarray()

cupy.**asanyarray**(*a*, *dtype=None*)
Converts an object to array.

This is currently equivalent to *asarray()*, since there is no subclass of ndarray in CuPy. Note that the original numpy.asanyarray() returns the input array as is if it is an instance of a subtype of numpy.ndarray.

See also:

cupy.asarray(), numpy.asanyarray()

cupy.**ascontiguousarray**(*a*, *dtype=None*)
Returns a C-contiguous array.

Parameters

- **a** (cupy.ndarray) – Source array.
- **dtype** – Data type specifier.

Returns If no copy is required, it returns a. Otherwise, it returns a copy of a.

Return type *cupy.ndarray*

See also:

numpy.ascontiguousarray()

cupy.**copy**(*a*)
Creates a copy of a given array on the current device.

This function allocates the new array on the current device. If the given array is allocated on the different device, then this function tries to copy the contents over the devices.

Parameters **a** (cupy.ndarray) – The source array.

Returns The copy of a on the current device.

Return type *cupy.ndarray*

See: numpy.copy(), *cupy.ndarray.copy()*

Numerical ranges

cupy.**arange**(*start*, *stop=None*, *step=1*, *dtype=None*)
Returns an array with evenly spaced values within a given interval.

Values are generated within the half-open interval [start, stop). The first three arguments are mapped like the range built-in function, i.e. start and step are optional.

Parameters

- **start** – Start of the interval.
- **stop** – End of the interval.
- **step** – Step width between each pair of consecutive values.
- **dtype** – Data type specifier. It is inferred from other arguments by default.

Returns The 1-D array of range values.

Return type *cupy.ndarray*

See also:

numpy.arange()

cupy.**linspace**(*start*, *stop*, *num=50*, *endpoint=True*, *retstep=False*, *dtype=None*)
Returns an array with evenly-spaced values within a given interval.

Instead of specifying the step width like *cupy.arange()*, this function requires the total number of elements specified.

Parameters

- **start** – Start of the interval.
- **stop** – End of the interval.
- **num** – Number of elements.
- **endpoint** (*bool*) – If True, the stop value is included as the last element. Otherwise, the stop value is omitted.
- **retstep** (*bool*) – If True, this function returns (array, step). Otherwise, it returns only the array.
- **dtype** – Data type specifier. It is inferred from the start and stop arguments by default.

Returns The 1-D array of ranged values.

Return type *cupy.ndarray*

Matrix creation

cupy.**diag**(*v*, *k=0*)
Returns a diagonal or a diagonal array.

Parameters

- **v** (*array-like*) – Array or array-like object.
- **k** (*int*) – Index of diagonals. Zero indicates the main diagonal, a positive value an upper diagonal, and a negative value a lower diagonal.

Returns If v indicates a 1-D array, then it returns a 2-D array with the specified diagonal filled by v. If v indicates a 2-D array, then it returns the specified diagonal of v. In latter case, if v is a *cupy.ndarray* object, then its view is returned.

Return type *cupy.ndarray*

See also:

numpy.diag()

cupy.**diagflat**(*v*, *k=0*)
Creates a diagonal array from the flattened input.

Parameters

- **v** (*array-like*) – Array or array-like object.
- **k** (*int*) – Index of diagonals. See *cupy.diag()* for detail.

Returns A 2-D diagonal array with the diagonal copied from v.

Return type *cupy.ndarray*

4.4.2 Array Manipulation Routines

Basic manipulations

cupy.**copyto**(*dst*, *src*, *casting='same_kind'*, *where=None*)
Copies values from one array to another with broadcasting.

This function can be called for arrays on different devices. In this case, casting, `where`, and broadcasting is not supported, and an exception is raised if these are used.

Parameters

- **dst** (cupy.ndarray) – Target array.
- **src** (cupy.ndarray) – Source array.
- **casting** (*str*) – Casting rule. See numpy.can_cast() for detail.
- **where** (*cupy.ndarray of bool*) – If specified, this array acts as a mask, and an element is copied only if the corresponding element of `where` is True.

See also:

numpy.copyto()

Shape manipulation

cupy.**reshape**(*a*, *newshape*)
Returns an array with new shape and same elements.

It tries to return a view if possible, otherwise returns a copy.

This function currently does not support `order` option.

Parameters

- **a** (cupy.ndarray) – Array to be reshaped.
- **newshape** (*int or tuple of ints*) – The new shape of the array to return. If it is an integer, then it is treated as a tuple of length one. It should be compatible with `a.size`. One of the elements can be -1, which is automatically replaced with the appropriate value to make the shape compatible with `a.size`.

Returns A reshaped view of `a` if possible, otherwise a copy.

Return type *cupy.ndarray*

See also:

numpy.reshape()

cupy.**ravel**(*a*)
Returns a flattened array.

It tries to return a view if possible, otherwise returns a copy.

This function currently does not support `order` option.

Parameters **a** (cupy.ndarray) – Array to be flattened.

Returns A flattened view of `a` if possible, otherwise a copy.

Return type *cupy.ndarray*

See also:

numpy.ravel()

Transposition

cupy.**rollaxis**(*a*, *axis*, *start=0*)
Moves the specified axis backwards to the given place.

Parameters

- **a** (cupy.ndarray) – Array to move the axis.
- **axis** (*int*) – The axis to move.
- **start** (*int*) – The place to which the axis is moved.

Returns A view of a that the axis is moved to start.

Return type *cupy.ndarray*

See also:

numpy.rollaxis()

cupy.**swapaxes**(*a*, *axis1*, *axis2*)
Swaps the two axes.

Parameters

- **a** (cupy.ndarray) – Array to swap the axes.
- **axis1** (*int*) – The first axis to swap.
- **axis2** (*int*) – The second axis to swap.

Returns A view of a that the two axes are swapped.

Return type *cupy.ndarray*

See also:

numpy.swapaxes()

cupy.**transpose**(*a*, *axes=None*)
Permutes the dimensions of an array.

Parameters

- **a** (cupy.ndarray) – Array to permute the dimensions.
- **axes** (*tuple of ints*) – Permutation of the dimensions. This function reverses the shape by default.

Returns A view of a that the dimensions are permuted.

Return type *cupy.ndarray*

See also:

numpy.transpose()

Edit dimensionalities

cupy.**atleast_1d**(**arys*)
Converts arrays to arrays with dimensions >= 1.

Parameters **arys** (*tuple of arrays*) – Arrays to be converted. All arguments must be *cupy.ndarray* objects. Only zero-dimensional array is affected.

Returns If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

See also:

numpy.atleast_1d()

cupy.**atleast_2d**(**arys*)
Converts arrays to arrays with dimensions >= 2.

If an input array has dimensions less than two, then this function inserts new axes at the head of dimensions to make it have two dimensions.

Parameters **arys** (*tuple of arrays*) – Arrays to be converted. All arguments must be *cupy.ndarray* objects.

Returns If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

See also:

numpy.atleast_2d()

cupy.**atleast_3d**(**arys*)
Converts arrays to arrays with dimensions >= 3.

If an input array has dimensions less than three, then this function inserts new axes to make it have three dimensions. The place of the new axes are following:

- If its shape is `()`, then the shape of output is `(1, 1, 1)`.
- If its shape is `(N,)`, then the shape of output is `(1, N, 1)`.
- If its shape is `(M, N)`, then the shape of output is `(M, N, 1)`.
- Otherwise, the output is the input array itself.

Parameters **arys** (*tuple of arrays*) – Arrays to be converted. All arguments must be *cupy.ndarray* objects.

Returns If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

See also:

numpy.atleast_3d()

class cupy.**broadcast**
Object that performs broadcasting.

CuPy actually uses this class to support broadcasting in various operations. Note that this class does not provide an iterator.

Parameters **arrays** (*tuple of arrays*) – Arrays to be broadcasted.

Variables

- **shape** (*tuple of ints*) – The broadcasted shape.
- **nd** (*int*) – Number of dimensions of the broadcasted shape.
- **size** (*int*) – Total size of the broadcasted shape.
- **values** (*list of arrays*) – The broadcasted arrays.

See also:

numpy.broadcast

cupy.**broadcast_arrays**(**args*)
Broadcasts given arrays.

Parameters **args** (*tuple of arrays*) – Arrays to broadcast for each other.

Returns A list of broadcasted arrays.

Return type list

See also:

numpy.broadcast_arrays()

cupy.**broadcast_to**(*array*, *shape*)
Broadcast an array to a given shape.

Parameters

- **array** (cupy.ndarray) – Array to broadcast.
- **shape** (*tuple of int*) – The shape of the desired array.

Returns Broadcasted view.

Return type *cupy.ndarray*

See also:

numpy.broadcast_to()

cupy.**expand_dims**(*a*, *axis*)
Expands given arrays.

Parameters

- **a** (cupy.ndarray) – Array to be expanded.
- **axis** (*int*) – Position where new axis is to be inserted.

Returns The number of dimensions is one greater than that of the input array.

Return type *cupy.ndarray*

See also:

numpy.expand_dims()

cupy.**squeeze**(*a*, *axis=None*)
Removes size-one axes from the shape of an array.

Parameters

- **a** (cupy.ndarray) – Array to be reshaped.
- **axis** (*int or tuple of ints*) – Axes to be removed. This function removes all size-one axes by default. If one of the specified axes is not of size one, an exception is raised.

Returns An array without (specified) size-one axes.

Return type *cupy.ndarray*

See also:

`numpy.squeeze()`

Changing kind of array

`cupy.`**`asfortranarray`**(*a*, *dtype=None*)
Return an array laid out in Fortran order in memory.

Parameters

- **a** (`ndarray`) – The input array.
- **dtype** (*str or dtype object, optional*) – By default, the data-type is inferred from the input data.

Returns The input *a* in Fortran, or column-major, order.

Return type *ndarray*

See also:

`numpy.asfortranarray()`

Joining arrays along axis

`cupy.`**`column_stack`**(*tup*)
Stacks 1-D and 2-D arrays as columns into a 2-D array.

A 1-D array is first converted to a 2-D column array. Then, the 2-D arrays are concatenated along the second axis.

Parameters **tup** (*sequence of arrays*) – 1-D or 2-D arrays to be stacked.

Returns A new 2-D array of stacked columns.

Return type *cupy.ndarray*

See also:

`numpy.column_stack()`

`cupy.`**`concatenate`**(*tup*, *axis=0*)
Joins arrays along an axis.

Parameters

- **tup** (*sequence of arrays*) – Arrays to be joined. All of these should have same dimensionalities except the specified axis.
- **axis** (*int*) – The axis to join arrays along.

Returns Joined array.

Return type *cupy.ndarray*

See also:

`numpy.concatenate()`

cupy.**vstack**(*tup*)
Stacks arrays vertically.

If an input array has one dimension, then the array is treated as a horizontal vector and stacked along the additional axis at the head. Otherwise, the array is stacked along the first axis.

Parameters **tup** (*sequence of arrays*) – Arrays to be stacked. Each array is converted by *cupy.atleast_2d()* before stacking.

Returns Stacked array.

Return type *cupy.ndarray*

See also:

numpy.dstack()

cupy.**hstack**(*tup*)
Stacks arrays horizontally.

If an input array has one dimension, then the array is treated as a horizontal vector and stacked along the first axis. Otherwise, the array is stacked along the second axis.

Parameters **tup** (*sequence of arrays*) – Arrays to be stacked.

Returns Stacked array.

Return type *cupy.ndarray*

See also:

numpy.hstack()

cupy.**dstack**(*tup*)
Stacks arrays along the third axis.

Parameters **tup** (*sequence of arrays*) – Arrays to be stacked. Each array is converted by *cupy.atleast_3d()* before stacking.

Returns Stacked array.

Return type *cupy.ndarray*

See also:

numpy.dstack()

Splitting arrays along axis

cupy.**array_split**(*ary*, *indices_or_sections*, *axis=0*)
Splits an array into multiple sub arrays along a given axis.

This function is almost equivalent to *cupy.split()*. The only difference is that this function allows an integer sections that does not evenly divide the axis.

See also:

cupy.split() for more detail, numpy.array_split()

cupy.**split**(*ary*, *indices_or_sections*, *axis=0*)
Splits an array into multiple sub arrays along a given axis.

Parameters

- **ary** (cupy.ndarray) – Array to split.

- **indices_or_sections** (*int or sequence of ints*) – A value indicating how to divide the axis. If it is an integer, then is treated as the number of sections, and the axis is evenly divided. Otherwise, the integers indicate indices to split at. Note that the sequence on the device memory is not allowed.
- **axis** (*int*) – Axis along which the array is split.

Returns A list of sub arrays. Each array is a view of the corresponding input array.

See also:

`numpy.split()`

`cupy.`**vsplit**(*ary*, *indices_or_sections*)
Splits an array into multiple sub arrays along the first axis.

This is equivalent to `split` with `axis=0`.

See also:

`cupy.split()` for more detail, `numpy.dsplit()`

`cupy.`**hsplit**(*ary*, *indices_or_sections*)
Splits an array into multiple sub arrays horizontally.

This is equivalent to `split` with `axis=0` if `ary` has one dimension, and otherwise that with `axis=1`.

See also:

`cupy.split()` for more detail, `numpy.hsplit()`

`cupy.`**dsplit**(*ary*, *indices_or_sections*)
Splits an array into multiple sub arrays along the third axis.

This is equivalent to `split` with `axis=2`.

See also:

`cupy.split()` for more detail, `numpy.dsplit()`

4.4.3 Repeating part of arrays along axis

`cupy.`**tile**(*A*, *reps*)
Construct an array by repeating A the number of times given by reps.

Parameters

- **A** (`cupy.ndarray`) – Array to transform.
- **reps** (*int or tuple*) – The number of repeats.

Returns Transformed array with repeats.

Return type *cupy.ndarray*

See also:

`numpy.tile()`

`cupy.`**repeat**(*a*, *repeats*, *axis=None*)
Repeat arrays along an axis.

Parameters

- **a** (`cupy.ndarray`) – Array to transform.
- **repeats** (*int, list or tuple*) – The number of repeats.

- **axis** (*int*) – The axis to repeat.

Returns Transformed array with repeats.

Return type *cupy.ndarray*

See also:

`numpy.repeat()`

4.4.4 Rearranging elements

`cupy.`**`roll`**(*a*, *shift*, *axis=None*)
Roll array elements along a given axis.

Parameters

- **a** (`ndarray`) – Array to be rolled.
- **shift** (*int*) – The number of places by which elements are shifted.
- **axis** (*int or None*) – The axis along which elements are shifted. If `axis` is `None`, the array is flattened before shifting, and after that it is reshaped to the original shape.

Returns Output array.

Return type *ndarray*

See also:

`numpy.roll()`

4.4.5 Binary Operations

Elementwise bit operations

`cupy.`**`bitwise_and`** = <ufunc 'cupy_bitwise_and'>
Computes the bitwise AND of two arrays elementwise.

Only integer and boolean arrays are handled.

See also:

`numpy.bitwise_and`

`cupy.`**`bitwise_or`** = <ufunc 'cupy_bitwise_or'>
Computes the bitwise OR of two arrays elementwise.

Only integer and boolean arrays are handled.

See also:

`numpy.bitwise_or`

`cupy.`**`bitwise_xor`** = <ufunc 'cupy_bitwise_xor'>
Computes the bitwise XOR of two arrays elementwise.

Only integer and boolean arrays are handled.

See also:

`numpy.bitwise_xor`

cupy.**invert** = <ufunc 'cupy_invert'>
Computes the bitwise NOT of an array elementwise.

Only integer and boolean arrays are handled.

See also:

numpy.invert

cupy.**left_shift** = <ufunc 'cupy_left_shift'>
Shifts the bits of each integer element to the left.

Only integer arrays are handled.

See also:

numpy.left_shift

cupy.**right_shift** = <ufunc 'cupy_right_shift'>
Shifts the bits of each integer element to the right.

Only integer arrays are handled

See also:

numpy.right_shift

4.4.6 Indexing Routines

cupy.**take**(*a*, *indices*, *axis=None*, *out=None*)
Takes elements of an array at specified indices along an axis.

This is an implementation of "fancy indexing" at single axis.

This function does not support `mode` option.

Parameters

- **a** (cupy.ndarray) – Array to extract elements.
- **indices** (*int or array-like*) – Indices of elements that this function takes.
- **axis** (*int*) – The axis along which to select indices. The flattened input is used by default.
- **out** (cupy.ndarray) – Output array. If provided, it should be of appropriate shape and dtype.

Returns The result of fancy indexing.

Return type *cupy.ndarray*

See also:

numpy.take()

cupy.**diagonal**(*a*, *offset=0*, *axis1=0*, *axis2=1*)
Returns specified diagonals.

This function extracts the diagonals along two specified axes. The other axes are not changed. This function returns a writable view of this array as NumPy 1.10 will do.

Parameters

- **a** (cupy.ndarray) – Array from which the diagonals are taken.
- **offset** (*int*) – Index of the diagonals. Zero indicates the main diagonals, a positive value upper diagonals, and a negative value lower diagonals.

- **axis1** (*int*) – The first axis to take diagonals from.
- **axis2** (*int*) – The second axis to take diagonals from.

Returns A view of the diagonals of `a`.

Return type *cupy.ndarray*

See also:

`numpy.diagonal()`

4.4.7 Input and Output

NPZ files

`cupy.`**load**(*file*, *mmap_mode=None*)

Loads arrays or pickled objects from `.npy`, `.npz` or pickled file.

This function just calls `numpy.load` and then sends the arrays to the current device. NPZ file is converted to NpzFile object, which defers the transfer to the time of accessing the items.

Parameters

- **file** (*file-like object or string*) – The file to read.
- **mmap_mode** (*None, 'r+', 'r', 'w+', 'c'*) – If not `None`, memory-map the file to construct an intermediate `numpy.ndarray` object and transfer it to the current device.

Returns CuPy array or NpzFile object depending on the type of the file. NpzFile object is a dictionary-like object with the context manager protocol (which enables us to use *with* statement on it).

See also:

`numpy.load()`

`cupy.`**save**(*file*, *arr*)

Saves an array to a binary file in `.npy` format.

Parameters

- **file** (*file or str*) – File or filename to save.
- **arr** (*array_like*) – Array to save. It should be able to feed to *cupy.asnumpy()*.

See also:

`numpy.save()`

`cupy.`**savez**(*file*, **args*, ***kwds*)

Saves one or more arrays into a file in uncompressed `.npz` format.

Arguments without keys are treated as arguments with automatic keys named `arr_0`, `arr_1`, etc. corresponding to the positions in the argument list. The keys of arguments are used as keys in the `.npz` file, which are used for accessing NpzFile object when the file is read by *cupy.load()* function.

Parameters

- **file** (*file or str*) – File or filename to save.
- ***args** – Arrays with implicit keys.
- ****kwds** – Arrays with explicit keys.

See also:

numpy.savez()

cupy.**savez_compressed**(*file*, **args*, ***kwds*)
Saves one or more arrays into a file in compressed .npz format.

It is equivalent to *cupy.savez()* function except the output file is compressed.

See also:

cupy.savez() for more detail, numpy.savez_compressed()

String formatting

cupy.**array_repr**(*arr*, *max_line_width=None*, *precision=None*, *suppress_small=None*)
Returns the string representation of an array.

Parameters

- **arr** (*array_like*) – Input array. It should be able to feed to *cupy.asnumpy()*.
- **max_line_width** (*int*) – The maximum number of line lengths.
- **precision** (*int*) – Floating point precision. It uses the current printing precision of NumPy.
- **suppress_small** (*bool*) – If True, very small numbers are printed as zeros

Returns The string representation of arr.

Return type str

See also:

numpy.array_repr()

cupy.**array_str**(*arr*, *max_line_width=None*, *precision=None*, *suppress_small=None*)
Returns the string representation of the content of an array.

Parameters

- **arr** (*array_like*) – Input array. It should be able to feed to *cupy.asnumpy()*.
- **max_line_width** (*int*) – The maximum number of line lengths.
- **precision** (*int*) – Floating point precision. It uses the current printing precision of NumPy.
- **suppress_small** (*bool*) – If True, very small number are printed as zeros.

See also:

numpy.array_str()

4.4.8 Linear Algebra

Matrix and vector products

cupy.**dot**(*a*, *b*, *out=None*)
Returns a dot product of two arrays.

For arrays with more than one axis, it computes the dot product along the last axis of `a` and the second-to-last axis of `b`. This is just a matrix product if the both arrays are 2-D. For 1-D arrays, it uses their unique axis as an axis to take dot product over.

Parameters

- **a** (`cupy.ndarray`) – The left argument.
- **b** (`cupy.ndarray`) – The right argument.
- **out** (`cupy.ndarray`) – Output array.

Returns The dot product of `a` and `b`.

Return type *cupy.ndarray*

See also:

`numpy.dot()`

`cupy.`**vdot**(*a*, *b*)

Returns the dot product of two vectors.

The input arrays are flattened into 1-D vectors and then it performs inner product of these vectors.

Parameters

- **a** (`cupy.ndarray`) – The first argument.
- **b** (`cupy.ndarray`) – The second argument.

Returns Zero-dimensional array of the dot product result.

Return type *cupy.ndarray*

See also:

`numpy.vdot()`

`cupy.`**inner**(*a*, *b*)

Returns the inner product of two arrays.

It uses the last axis of each argument to take sum product.

Parameters

- **a** (`cupy.ndarray`) – The first argument.
- **b** (`cupy.ndarray`) – The second argument.

Returns The inner product of `a` and `b`.

Return type *cupy.ndarray*

See also:

`numpy.inner()`

`cupy.`**outer**(*a*, *b*, *out=None*)

Returns the outer product of two vectors.

The input arrays are flattened into 1-D vectors and then it performs outer product of these vectors.

Parameters

- **a** (`cupy.ndarray`) – The first argument.
- **b** (`cupy.ndarray`) – The second argument.
- **out** (`cupy.ndarray`) – Output array.

Returns 2-D array of the outer product of `a` and `b`.

Return type *cupy.ndarray*

See also:

`numpy.outer()`

`cupy.`**`tensordot`**(*a*, *b*, *axes=2*)

Returns the tensor dot product of two arrays along specified axes.

This is equivalent to compute dot product along the specified axes which are treated as one axis by reshaping.

Parameters

- **a** (`cupy.ndarray`) – The first argument.
- **b** (`cupy.ndarray`) – The second argument.
- **axes** –
  - If it is an integer, then `axes` axes at the last of `a` and the first of `b` are used.
  - If it is a pair of sequences of integers, then these two sequences specify the list of axes for `a` and `b`. The corresponding axes are paired for sum-product.
- **out** (`cupy.ndarray`) – Output array.

Returns The tensor dot product of `a` and `b` along the axes specified by `axes`.

Return type *cupy.ndarray*

See also:

`numpy.tensordot()`

Norms etc.

`cupy.`**`trace`**(*a*, *offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Returns the sum along the diagonals of an array.

It computes the sum along the diagonals at `axis1` and `axis2`.

Parameters

- **a** (`cupy.ndarray`) – Array to take trace.
- **offset** (*int*) – Index of diagonals. Zero indicates the main diagonal, a positive value an upper diagonal, and a negative value a lower diagonal.
- **axis1** (*int*) – The first axis along which the trace is taken.
- **axis2** (*int*) – The second axis along which the trace is taken.
- **dtype** – Data type specifier of the output.
- **out** (`cupy.ndarray`) – Output array.

Returns The trace of `a` along axes `(axis1, axis2)`.

Return type *cupy.ndarray*

See also:

`numpy.trace()`

4.4.9 Logic Functions

Infinities and NaNs

cupy.**isfinite** = <ufunc 'cupy_isfinite'>
Tests finiteness elementwise.

Each element of returned array is `True` only if the corresponding element of the input is finite (i.e. not an infinity nor NaN).

See also:

`numpy.isfinite`

cupy.**isinf** = <ufunc 'cupy_isinf'>
Tests if each element is the positive or negative infinity.

See also:

`numpy.isinf`

cupy.**isnan** = <ufunc 'cupy_isnan'>
Tests if each element is a NaN.

See also:

`numpy.isnan`

Logic operations

cupy.**logical_and** = <ufunc 'cupy_logical_and'>
Computes the logical AND of two arrays.

See also:

`numpy.logical_and`

cupy.**logical_or** = <ufunc 'cupy_logical_or'>
Computes the logical OR of two arrays.

See also:

`numpy.logical_or`

cupy.**logical_not** = <ufunc 'cupy_logical_not'>
Computes the logical NOT of an array.

See also:

`numpy.logical_not`

cupy.**logical_xor** = <ufunc 'cupy_logical_xor'>
Computes the logical XOR of two arrays.

See also:

`numpy.logical_xor`

Comparison operations

cupy.**greater** = <ufunc 'cupy_greater'>
Tests elementwise if `x1 > x2`.

See also:

`numpy.greater`

cupy.**greater_equal** = <ufunc 'cupy_greater_equal'>
Tests elementwise if `x1 >= x2`.

See also:

`numpy.greater_equal`

cupy.**less** = <ufunc 'cupy_less'>
Tests elementwise if `x1 < x2`.

See also:

`numpy.less`

cupy.**less_equal** = <ufunc 'cupy_less_equal'>
Tests elementwise if `x1 <= x2`.

See also:

`numpy.less_equal`

cupy.**equal** = <ufunc 'cupy_equal'>
Tests elementwise if `x1 == x2`.

See also:

`numpy.equal`

cupy.**not_equal** = <ufunc 'cupy_not_equal'>
Tests elementwise if `x1 != x2`.

See also:

`numpy.equal`

4.4.10 Mathematical Functions

Trigonometric functions

cupy.**sin** = <ufunc 'cupy_sin'>
Elementwise sine function.

See also:

`numpy.sin`

cupy.**cos** = <ufunc 'cupy_cos'>
Elementwise cosine function.

See also:

`numpy.cos`

cupy.**tan = <ufunc 'cupy_tan'>**
Elementwise tangent function.

See also:

numpy.tan

cupy.**arcsin = <ufunc 'cupy_arcsin'>**
Elementwise inverse-sine function (a.k.a. arcsine function).

See also:

numpy.arcsin

cupy.**arccos = <ufunc 'cupy_arccos'>**
Elementwise inverse-cosine function (a.k.a. arccosine function).

See also:

numpy.arccos

cupy.**arctan = <ufunc 'cupy_arctan'>**
Elementwise inverse-tangent function (a.k.a. arctangent function).

See also:

numpy.arctan

cupy.**hypot = <ufunc 'cupy_hypot'>**
Computes the hypoteneous of orthogonal vectors of given length.

This is equivalent to `sqrt(x1 **2 + x2 ** 2)`, while this function is more efficient.

See also:

numpy.hypot

cupy.**arctan2 = <ufunc 'cupy_arctan2'>**
Elementwise inverse-tangent of the ratio of two arrays.

See also:

numpy.arctan2

cupy.**deg2rad = <ufunc 'cupy_deg2rad'>**
Converts angles from degrees to radians elementwise.

See also:

numpy.deg2rad, numpy.radians

cupy.**rad2deg = <ufunc 'cupy_rad2deg'>**
Converts angles from radians to degrees elementwise.

See also:

numpy.rad2deg, numpy.degrees

cupy.**degrees = <ufunc 'cupy_rad2deg'>**
Converts angles from radians to degrees elementwise.

See also:

numpy.rad2deg, numpy.degrees

cupy.**radians = <ufunc 'cupy_deg2rad'>**
Converts angles from degrees to radians elementwise.

See also:

numpy.deg2rad, numpy.radians

Hyperbolic functions

cupy.**sinh = <ufunc 'cupy_sinh'>**
Elementwise hyperbolic sine function.

See also:

numpy.sinh

cupy.**cosh = <ufunc 'cupy_cosh'>**
Elementwise hyperbolic cosine function.

See also:

numpy.cosh

cupy.**tanh = <ufunc 'cupy_tanh'>**
Elementwise hyperbolic tangent function.

See also:

numpy.tanh

cupy.**arcsinh = <ufunc 'cupy_arcsinh'>**
Elementwise inverse of hyperbolic sine function.

See also:

numpy.arcsinh

cupy.**arccosh = <ufunc 'cupy_arccosh'>**
Elementwise inverse of hyperbolic cosine function.

See also:

numpy.arccosh

cupy.**arctanh = <ufunc 'cupy_arctanh'>**
Elementwise inverse of hyperbolic tangent function.

See also:

numpy.arctanh

Rounding

cupy.**rint = <ufunc 'cupy_rint'>**
Rounds each element of an array to the nearest integer.

See also:

numpy.rint

cupy.**floor = <ufunc 'cupy_floor'>**
Rounds each element of an array to its floor integer.

See also:

numpy.floor

cupy.**ceil** = <ufunc 'cupy_ceil'>
Rounds each element of an array to its ceiling integer.

See also:

numpy.ceil

cupy.**trunc** = <ufunc 'cupy_trunc'>
Rounds each element of an array towards zero.

See also:

numpy.trunc

Sums and products

cupy.**sum**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the sum of an array along given axes.

Parameters

- **a** (cupy.ndarray) – Array to take sum.
- **axis** (*int or sequence of ints*) – Axes along which the sum is taken.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

Returns The result array.

Return type *cupy.ndarray*

See also:

numpy.sum()

cupy.**prod**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the product of an array along given axes.

Parameters

- **a** (cupy.ndarray) – Array to take product.
- **axis** (*int or sequence of ints*) – Axes along which the product is taken.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

Returns The result array.

Return type *cupy.ndarray*

See also:

numpy.prod()

Exponential and logarithm functions

cupy.**exp = <ufunc 'cupy_exp'>**
Elementwise exponential function.

See also:

numpy.exp

cupy.**expm1 = <ufunc 'cupy_expm1'>**
Computes `exp(x) - 1` elementwise.

See also:

numpy.expm1

cupy.**exp2 = <ufunc 'cupy_exp2'>**
Elementwise exponentiation with base 2.

See also:

numpy.exp2

cupy.**log = <ufunc 'cupy_log'>**
Elementwise natural logarithm function.

See also:

numpy.log

cupy.**log10 = <ufunc 'cupy_log10'>**
Elementwise common logarithm function.

See also:

numpy.log10

cupy.**log2 = <ufunc 'cupy_log2'>**
Elementwise binary logarithm function.

See also:

numpy.log2

cupy.**log1p = <ufunc 'cupy_log1p'>**
Computes `log(1 + x)` elementwise.

See also:

numpy.log1p

cupy.**logaddexp = <ufunc 'cupy_logaddexp'>**
Computes `log(exp(x1) + exp(x2))` elementwise.

See also:

numpy.logaddexp

cupy.**logaddexp2 = <ufunc 'cupy_logaddexp2'>**
Computes `log2(exp2(x1) + exp2(x2))` elementwise.

See also:

numpy.logaddexp2

Floating point manipulations

cupy.**signbit = <ufunc 'cupy_signbit'>**
Tests elementwise if the sign bit is set (i.e. less than zero).

See also:

numpy.signbit

cupy.**copysign = <ufunc 'cupy_copysign'>**
Returns the first argument with the sign bit of the second elementwise.

See also:

numpy.copysign

cupy.**ldexp = <ufunc 'cupy_ldexp'>**
Computes `x1 * 2 ** x2` elementwise.

See also:

numpy.ldexp

cupy.**frexp = <ufunc 'cupy_frexp'>**
Decomposes each element to mantissa and two's exponent.

This ufunc outputs two arrays of the input dtype and the `int` dtype.

See also:

numpy.frexp

cupy.**nextafter = <ufunc 'cupy_nextafter'>**
Computes the nearest neighbor float values towards the second argument.

See also:

numpy.nextafter

Arithmetic operations

cupy.**negative = <ufunc 'cupy_negative'>**
Takes numerical negative elementwise.

See also:

numpy.negative

cupy.**add = <ufunc 'cupy_add'>**
Adds two arrays elementwise.

See also:

numpy.add

cupy.**subtract = <ufunc 'cupy_subtract'>**
Subtracts arguments elementwise.

See also:

numpy.subtract

cupy.**multiply = <ufunc 'cupy_multiply'>**
Multiplies two arrays elementwise.

See also:

numpy.multiply

cupy.**divide** = <ufunc 'cupy_divide'>
Divides arguments elementwise.

See also:

numpy.divide

cupy.**true_divide** = <ufunc 'cupy_true_divide'>
Elementwise true division (i.e. division as floating values).

See also:

numpy.true_divide

cupy.**floor_divide** = <ufunc 'cupy_floor_divide'>
Elementwise floor division (i.e. integer quotient).

See also:

numpy.floor_divide

cupy.**power** = <ufunc 'cupy_power'>
Computes `x1 ** x2` elementwise.

See also:

numpy.power

cupy.**fmod** = <ufunc 'cupy_fmod'>
Computes the remainder of C division elementwise.

See also:

numpy.fmod

cupy.**mod** = <ufunc 'cupy_remainder'>
Computes the remainder of Python division elementwise.

See also:

numpy.remainder

cupy.**remainder** = <ufunc 'cupy_remainder'>
Computes the remainder of Python division elementwise.

See also:

numpy.remainder

cupy.**modf** = <ufunc 'cupy_modf'>
Extracts the fractional and integral parts of an array elementwise.

This ufunc returns two arrays.

See also:

numpy.modf

cupy.**reciprocal** = <ufunc 'cupy_reciprocal'>
Computes `1 / x` elementwise.

See also:

numpy.reciprocal

Miscellaneous

cupy.**clip**(*a*, *a_min*, *a_max*, *out=None*)
Clips the values of an array to a given interval.

This is equivalent to `maximum(minimum(a, a_max), a_min)`, while this function is more efficient.

Parameters

- **a** (cupy.ndarray) – The source array.
- **a_min** (*scalar or cupy.ndarray*) – The left side of the interval.
- **a_max** (*scalar or cupy.ndarray*) – The right side of the interval.
- **out** (cupy.ndarray) – Output array.

Returns Clipped array.

Return type *cupy.ndarray*

See also:

numpy.clip()

cupy.**sqrt = <ufunc 'cupy_sqrt'>**
Elementwise positive square-root function.

Note: This ufunc outputs float32 arrays for float16 arrays input by default as well as NumPy 1.9. If you want to override this behavior, specify the dtype argument explicitly, or use `cupy.math.misc.sqrt_fixed` instead.

See also:

numpy.sqrt

cupy.**square = <ufunc 'cupy_square'>**
Elementwise square function.

See also:

numpy.square

cupy.**absolute = <ufunc 'cupy_absolute'>**
Elementwise absolute value function.

See also:

numpy.absolute

cupy.**sign = <ufunc 'cupy_sign'>**
Elementwise sign function.

It returns -1, 0, or 1 depending on the sign of the input.

See also:

numpy.sign

cupy.**maximum = <ufunc 'cupy_maximum'>**
Takes the maximum of two arrays elementwise.

If NaN appears, it returns the NaN.

See also:

numpy.maximum

cupy.**minimum** = <ufunc 'cupy_minimum'>
Takes the minimum of two arrays elementwise.

If NaN appears, it returns the NaN.

See also:

numpy.minimum

cupy.**fmax** = <ufunc 'cupy_fmax'>
Takes the maximum of two arrays elementwise.

If NaN appears, it returns the other operand.

See also:

numpy.fmax

cupy.**fmin** = <ufunc 'cupy_fmin'>
Takes the minimum of two arrays elementwise.

If NaN appears, it returns the other operand.

See also:

numpy.fmin

4.4.11 Random Sampling (cupy.random)

CuPy's random number generation routines are based on cuRAND. They cover a small fraction of numpy.random.

The big difference of *cupy.random* from numpy.random is that *cupy.random* supports dtype option for most functions. This option enables us to generate float32 values directly without any space overhead.

Sample random data

cupy.random.**rand**(**size*, ***kwarg*)
Returns an array of uniform random values over the interval [0, 1).

Each element of the array is uniformly distributed on the half-open interval [0, 1). All elements are identically and independently distributed (i.i.d.).

Parameters

- **size** (*tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only numpy.float32 and numpy.float64 types are allowed. The default is numpy.float64.

Returns A random array.

Return type *cupy.ndarray*

See also:

numpy.random.rand()

cupy.random.**randn**(**size*, ***kwarg*)
Returns an array of standard normal random values.

Each element of the array is normally distributed with zero mean and unit variance. All elements are identically and independently distributed (i.i.d.).

Parameters

- **size** (*tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed. The default is `numpy.float64`.

Returns An array of standard normal random values.

Return type *cupy.ndarray*

See also:

`numpy.random.randn()`

`cupy.random.`**randint**(*low*, *high=None*, *size=None*)

Returns a scalar or an array of integer values over `[low, high)`.

Each element of returned values are independently sampled from uniform distribution over left-close and right-open interval `[low, high)`.

Parameters

- **low** (*int*) – If `high` is not `None`, it is the lower bound of the interval. Otherwise, it is the **upper** bound of the interval and lower bound of the interval is set to `0`.
- **high** (*int*) – Upper bound of the interval.
- **size** (*None or int or tuple of ints*) – The shape of returned value.

Returns If size is `None`, it is single integer sampled. If size is integer, it is the 1D-array of length `size` element. Otherwise, it is the array whose shape specified by `size`.

Return type int or cupy.ndarray of ints

`cupy.random.`**random_integers**(*low*, *high=None*, *size=None*)

Return a scalar or an array of integer values over `[low, high]`

Each element of returned values are independently sampled from uniform distribution over closed interval `[low, high]`.

Parameters

- **low** (*int*) – If `high` is not `None`, it is the lower bound of the interval. Otherwise, it is the **upper** bound of the interval and the lower bound is set to `1`.
- **high** (*int*) – Upper bound of the interval.
- **size** (*None or int or tuple of ints*) – The shape of returned value.

Returns If size is `None`, it is single integer sampled. If size is integer, it is the 1D-array of length `size` element. Otherwise, it is the array whose shape specified by `size`.

Return type int or cupy.ndarray of ints

`cupy.random.`**random_sample**(*size=None*, *dtype=<type 'float'>*)

Returns an array of random values over the interval `[0, 1)`.

This is a variant of *cupy.random.rand()*.

Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

Returns An array of uniformly distributed random values.

Return type *cupy.ndarray*

See also:

`numpy.random.random_sample()`

cupy.random.**random**(*size=None*, *dtype=<type 'float'>*)
Returns an array of random values over the interval `[0, 1)`.

This is a variant of *cupy.random.rand()*.

Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

Returns An array of uniformly distributed random values.

Return type *cupy.ndarray*

See also:

`numpy.random.random_sample()`

cupy.random.**ranf**(*size=None*, *dtype=<type 'float'>*)
Returns an array of random values over the interval `[0, 1)`.

This is a variant of *cupy.random.rand()*.

Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

Returns An array of uniformly distributed random values.

Return type *cupy.ndarray*

See also:

`numpy.random.random_sample()`

cupy.random.**sample**(*size=None*, *dtype=<type 'float'>*)
Returns an array of random values over the interval `[0, 1)`.

This is a variant of *cupy.random.rand()*.

Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

Returns An array of uniformly distributed random values.

Return type *cupy.ndarray*

See also:

`numpy.random.random_sample()`

Distributions

cupy.random.**lognormal**(*mean=0.0*, *sigma=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of samples drawn from a log normal distribution.

The samples are natural log of samples drawn from a normal distribution with mean `mean` and deviation `sigma`.

Parameters

- **mean** (*float*) – Mean of the normal distribution.
- **sigma** (*float*) – Standard deviation of the normal distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

Returns Samples drawn from the log normal distribution.

Return type *cupy.ndarray*

See also:

`numpy.random.lognormal()`

cupy.random.**normal**(*loc=0.0*, *scale=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of normally distributed samples.

Parameters

- **loc** (*float*) – Mean of the normal distribution.
- **scale** (*float*) – Standard deviation of the normal distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier. Only `numpy.float32` and `numpy.float64` types are allowed.

Returns Normally distributed samples.

Return type *cupy.ndarray*

See also:

`numpy.random.normal()`

cupy.random.**standard_normal**(*size=None*, *dtype=<type 'float'>*)
Returns an array of samples drawn from the standard normal distribution.

This is a variant of *cupy.random.randn()*.

Parameters

- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier.

Returns Samples drawn from the standard normal distribution.

Return type *cupy.ndarray*

See also:

`numpy.randomm.standard_normal()`

`cupy.random.`**`uniform`**(*low=0.0*, *high=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of uniformly-distributed samples over an interval.

Samples are drawn from a uniform distribution over the half-open interval `[low, high)`.

Parameters

- **low** (*float*) – Lower end of the interval.
- **high** (*float*) – Upper end of the interval.
- **size** (*int or tuple of ints*) – The shape of the array. If `None`, a zero-dimensional array is generated.
- **dtype** – Data type specifier.

Returns Samples drawn from the uniform distribution.

Return type *cupy.ndarray*

See also:

`numpy.random.uniform()`

Random number generator

`cupy.random.`**`seed`**(*seed=None*)
Resets the state of the random number generator with a seed.

This function resets the state of the global random number generator for the current device. Be careful that generators for other devices are not affected.

Parameters **seed** (*None or int*) – Seed for the random number generator. If `None`, it uses `os.urandom()` if available or `time.clock()` otherwise. Note that this function does not support seeding by an integer array.

`cupy.random.`**`get_random_state`**()
Gets the state of the random number generator for the current device.

If the state for the current device is not created yet, this function creates a new one, initializes it, and stores it as the state for the current device.

Returns The state of the random number generator for the device.

Return type *RandomState*

class `cupy.random.`**`RandomState`**(*seed=None*, *method=100*)
Portable container of a pseudo-random number generator.

An instance of this class holds the state of a random number generator. The state is available only on the device which has been current at the initialization of the instance.

Functions of *cupy.random* use global instances of this class. Different instances are used for different devices. The global state for the current device can be obtained by the *cupy.random.get_random_state()* function.

Parameters

- **seed** (*None or int*) – Seed of the random number generator. See the *seed()* method for detail.

- **method** (*int*) – Method of the random number generator. Following values are available:

```
cupy.cuda.curand.CURAND_RNG_PSEUDO_DEFAULT
cupy.cuda.curand.CURAND_RNG_XORWOW
cupy.cuda.curand.CURAND_RNG_MRG32K3A
cupy.cuda.curand.CURAND_RNG_MTGP32
cupy.cuda.curand.CURAND_RNG_MT19937
cupy.cuda.curand.CURAND_RNG_PHILOX4_32_10
```

interval(*mx*, *size*)
Generate multiple integers independently sampled uniformly from `[0, mx]`.

Parameters

- **mx** (*int*) – Upper bound of the interval
- **size** (*None or int or tuple*) – Shape of the array or the scalar returned.

Returns If `None`, an *cupy.ndarray* with shape `()` is returned. If `int`, 1-D array of length size is returned. If `tuple`, multi-dimensional array with shape `size` is returned. Currently, each element of the array is `numpy.int32`.

Return type int or cupy.ndarray

lognormal(*mean=0.0*, *sigma=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of samples drawn from a log normal distribution.

See also:

cupy.random.lognormal() for full documentation, `numpy.random.RandomState.lognormal()`

normal(*loc=0.0*, *scale=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of normally distributed samples.

See also:

cupy.random.normal() for full documentation, `numpy.random.RandomState.normal()`

rand(**size*, ***kwarg*)
Returns uniform random values over the interval `[0, 1)`.

See also:

cupy.random.rand() for full documentation, `numpy.random.RandomState.rand()`

randn(**size*, ***kwarg*)
Returns an array of standard normal random values.

See also:

cupy.random.randn() for full documentation, `numpy.random.RandomState.randn()`

random_sample(*size=None*, *dtype=<type 'float'>*)
Returns an array of random values over the interval `[0, 1)`.

See also:

cupy.random.random_sample() for full documentation, `numpy.random.RandomState.random_sample()`

seed(*seed=None*)
Resets the state of the random number generator with a seed.

See also:

cupy.random.seed() for full documentation, `numpy.random.RandomState.seed()`

standard_normal(*size=None*, *dtype=<type 'float'>*)
Returns samples drawn from the standard normal distribution.

See also:

cupy.random.standard_normal() for full documentation, numpy.random.RandomState.standard_norm

uniform(*low=0.0*, *high=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of uniformly-distributed samples over an interval.

See also:

cupy.random.uniform() for full documentation, numpy.random.RandomState.uniform()

4.4.12 Sorting, Searching, and Counting

cupy.**argmax**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the indices of the maximum along an axis.

Parameters

- **a** (cupy.ndarray) – Array to take argmax.
- **axis** (*int*) – Along which axis to find the maximum. a is flattened by default.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis axis is preserved as an axis of length one.

Returns The indices of the maximum of a along an axis.

Return type *cupy.ndarray*

See also:

numpy.argmax()

cupy.**argmin**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the indices of the minimum along an axis.

Parameters

- **a** (cupy.ndarray) – Array to take argmin.
- **axis** (*int*) – Along which axis to find the minimum. a is flattened by default.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis axis is preserved as an axis of length one.

Returns The indices of the minimum of a along an axis.

Return type *cupy.ndarray*

See also:

numpy.argmin()

cupy.**count_nonzero**(*x*)
Counts the number of non-zero values in the array.

Parameters **x** (cupy.ndarray) – The array for which to count non-zeros.

Returns Number of non-zero values in the array.

Return type int

cupy.**where**(*condition*, *x=None*, *y=None*)
Return elements, either from x or y, depending on condition.

Note: Currently CuPy doesn't support `where(condition)`, that NumPy supports.

Parameters

- **condition** (cupy.ndarray) – When True, take x, otherwise take y.
- **x** (cupy.ndarray) – Values from which to choose on `True`.
- **y** (cupy.ndarray) – Values from which to choose on `False`.

Returns Each element of output contains elements of `x` when `condition` is `True`, otherwise elements of `y`.

Return type *cupy.ndarray*

4.4.13 Statistics

Order statistics

cupy.**amin**(*a*, *axis=None*, *out=None*, *keepdims=False*, *dtype=None*)
Returns the minimum of an array or the minimum along an axis.

Parameters

- **a** (cupy.ndarray) – Array to take the minimum.
- **axis** (*int*) – Along which axis to take the minimum. The flattened array is used by default.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If `True`, the axis is remained as an axis of size one.
- **dtype** – Data type specifier.

Returns The minimum of `a`, along the axis if specified.

Return type *cupy.ndarray*

See also:

numpy.amin()

cupy.**amax**(*a*, *axis=None*, *out=None*, *keepdims=False*, *dtype=None*)
Returns the maximum of an array or the maximum along an axis.

Parameters

- **a** (cupy.ndarray) – Array to take the maximum.
- **axis** (*int*) – Along which axis to take the maximum. The flattened array is used by default.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If `True`, the axis is remained as an axis of size one.
- **dtype** – Data type specifier.

Returns The maximum of `a`, along the axis if specified.

Return type *cupy.ndarray*

See also:

`numpy.amax()`

Means and variances

`cupy.`**`mean`**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the arithmetic mean along an axis.

Parameters

- **a** (`cupy.ndarray`) – Array to compute mean.
- **axis** (*int*) – Along which axis to compute mean. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (*bool*) – If `True`, the axis is remained as an axis of size one.

Returns The mean of the input array along the axis.

Return type *cupy.ndarray*

See also:

`numpy.mean()`

`cupy.`**`var`**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the variance along an axis.

Parameters

- **a** (`cupy.ndarray`) – Array to compute variance.
- **axis** (*int*) – Along which axis to compute variance. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (`cupy.ndarray`) – Output array.
- **keepdims** (*bool*) – If `True`, the axis is remained as an axis of size one.

Returns The variance of the input array along the axis.

Return type *cupy.ndarray*

See also:

`numpy.var()`

`cupy.`**`std`**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the standard deviation along an axis.

Parameters

- **a** (`cupy.ndarray`) – Array to compute standard deviation.
- **axis** (*int*) – Along which axis to compute standard deviation. The flattened array is used by default.
- **dtype** – Data type specifier.

- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.

Returns The standard deviation of the input array along the axis.

Return type *cupy.ndarray*

See also:

numpy.std()

Histograms

cupy.**bincount**(*x*, *weights=None*, *minlength=None*)

Count number of occurrences of each value in array of non-negative ints.

Parameters

- **x** (cupy.ndarray) – Input array.
- **weights** (cupy.ndarray) – Weights array which has the same shape as x.
- **minlength** (*int*) – A minimum number of bins for the output array.

Returns The result of binning the input array. The length of output is equal to max(cupy.max(x) + 1, minlength).

Return type *cupy.ndarray*

See also:

numpy.bincount()

4.5 NumPy-CuPy Generic Code Support

cupy.**get_array_module**(**args*)

Returns the array module for arguments.

This function is used to implement CPU/GPU generic code. If at least one of the arguments is a *cupy.ndarray* object, the *cupy* module is returned.

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

Example

A NumPy/CuPy generic function can be written as follows:

```
>>> def softplus(x):
```

... xp = cupy.get_array_module(x) ... return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))

4.6 Low-Level CUDA Support

4.6.1 Device management

class cupy.cuda.**Device**
Object that represents a CUDA device.

This class provides some basic manipulations on CUDA devices.

It supports the context protocol. For example, the following code is an example of temporarily switching the current device:

```
with Device(0):
    do_something_on_device_0()
```

After the *with* statement gets done, the current device is reset to the original one.

Parameters **device** (*int or cupy.cuda.Device*) – Index of the device to manipulate. Be careful that the device ID (a.k.a. GPU ID) is zero origin. If it is a Device object, then its ID is used. The current device is selected by default.

Variables **id** (*int*) – ID of this device.

__eq__
x.__eq__(y) <==> x==y

__ge__
x.__ge__(y) <==> x>=y

__gt__
x.__gt__(y) <==> x>y

__int__

__le__
x.__le__(y) <==> x<=y

__long__

__lt__
x.__lt__(y) <==> x<y

__ne__
x.__ne__(y) <==> x!=y

__repr__

compute_capability
Compute capability of this device.

The capability is represented by a string containing the major index and the minor index. For example, compute capability 3.5 is represented by the string '35'.

cublas_handle
The cuBLAS handle for this device.

The same handle is used for the same device even if the Device instance itself is different.

synchronize()
Synchronizes the current thread to the device.

use()
Makes this device current.

If you want to switch a device temporarily, use the *with* statement.

4.6.2 Memory management

class cupy.cuda.**Memory**
Memory allocation on a CUDA device.

This class provides an RAII interface of the CUDA memory allocation.

Parameters

- **device** (cupy.cuda.Device) – Device whose memory the pointer refers to.
- **size** (*int*) – Size of the memory allocation in bytes.

__int__
Returns the pointer value to the head of the allocation.

__long__

class cupy.cuda.**MemoryPointer**
Pointer to a point on a device memory.

An instance of this class holds a reference to the original memory buffer and a pointer to a place within this buffer.

Parameters

- **mem** (Memory) – The device memory buffer.
- **offset** (*int*) – An offset from the head of the buffer to the place this pointer refers.

Variables

- ***device*** (cupy.cuda.Device) – Device whose memory the pointer refers to.
- **mem** (Memory) – The device memory buffer.
- **ptr** (*int*) – Pointer to the place within the buffer.

__add__
Adds an offset to the pointer.

__iadd__
Adds an offset to the pointer in place.

__int__
Returns the pointer value.

__isub__
Subtracts an offset from the pointer in place.

__long__

__radd__
x.__radd__(y) <==> y+x

__rsub__
x.__rsub__(y) <==> y-x

__sub__
Subtracts an offset from the pointer.

copy_from()
Copies a memory sequence from a (possibly different) device or host.

This function is a useful interface that selects appropriate one from *copy_from_device()* and *copy_from_host()*.

Parameters

- **mem** (*ctypes.c_void_p or cupy.cuda.MemoryPointer*) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

copy_from_async()
Copies a memory sequence from an arbitrary place asynchronously.

This function is a useful interface that selects appropriate one from *copy_from_device_async()* and *copy_from_host_async()*.

Parameters

- **mem** (*ctypes.c_void_p or cupy.cuda.MemoryPointer*) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (cupy.cuda.Stream) – CUDA stream.

copy_from_device()
Copies a memory sequence from a (possibly different) device.

Parameters

- **src** (cupy.cuda.MemoryPointer) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

copy_from_device_async()
Copies a memory sequence from a (possibly different) device asynchronously.

Parameters

- **src** (cupy.cuda.MemoryPointer) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (cupy.cuda.Stream) – CUDA stream.

copy_from_host()
Copies a memory sequence from the host memory.

Parameters

- **mem** (*ctypes.c_void_p*) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

copy_from_host_async()
Copies a memory sequence from the host memory asynchronously.

Parameters

- **src** (*ctypes.c_void_p*) – Source memory pointer. It must be a pinned memory.
- **size** (*int*) – Size of the sequence in bytes.

copy_to_host()
Copies a memory sequence to the host memory.

Parameters

- **mem** (*ctypes.c_void_p*) – Target memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

copy_to_host_async()
Copies a memory sequence to the host memory asynchronously.

Parameters

- **mem** (*ctypes.c_void_p*) – Target memory pointer. It must be a pinned memory.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (cupy.cuda.Stream) – CUDA stream.

memset()
Fills a memory sequence by constant byte value.

Parameters

- **value** (*int*) – Value to fill.
- **size** (*int*) – Size of the sequence in bytes.

memset_async()
Fills a memory sequence by constant byte value asynchronously.

Parameters

- **value** (*int*) – Value to fill.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (cupy.cuda.Stream) – CUDA stream.

cupy.cuda.**alloc**()
Calls the current allocator.

Use *set_allocator()* to change the current allocator.

Parameters **size** (*int*) – Size of the memory allocation.

Returns Pointer to the allocated buffer.

Return type *MemoryPointer*

cupy.cuda.**set_allocator**()
Sets the current allocator.

Parameters **allocator** (function) – CuPy memory allocator. It must have the same interface as the *cupy.cuda.alloc()* function, which takes the buffer size as an argument and returns the device buffer of that size.

class cupy.cuda.**MemoryPool**
Memory pool for all devices on the machine.

A memory pool preserves any allocations even if they are freed by the user. Freed memory buffers are held by the memory pool as *free blocks*, and they are reused for further memory allocations of the same sizes. The allocated blocks are managed for each device, so one instance of this class can be used for multiple devices.

Note: When the allocation is skipped by reusing the pre-allocated block, it does not call cudaMalloc and therefore CPU-GPU synchronization does not occur. It makes interleaves of memory allocations and kernel invocations very fast.

Note: The memory pool holds allocated blocks without freeing as much as possible. It makes the program hold most of the device memory, which may make other CUDA programs running in parallel out-of-memory situation.

Parameters **allocator** (function) – The base CuPy memory allocator. It is used for allocating new blocks when the blocks of the required size are all in use.

free_all_free()
Release free blocks.

malloc()
Allocates the memory, from the pool if possible.

This method can be used as a CuPy memory allocator. The simplest way to use a memory pool as the default allocator is the following code:

```
set_allocator(MemoryPool().malloc)
```

Parameters **size** (*int*) – Size of the memory buffer to allocate in bytes.

Returns Pointer to the allocated buffer.

Return type *MemoryPointer*

n_free_blocks()
Count the total number of free blocks.

Returns The total number of free blocks.

Return type int

4.6.3 Streams and events

class cupy.cuda.**Stream**(*null=False*, *non_blocking=False*)
CUDA stream.

This class handles the CUDA stream handle in RAII way, i.e., when an Stream instance is destroyed by the GC, its handle is also destroyed.

Parameters

- **null** (*bool*) – If True, the stream is a null stream (i.e. the default stream that synchronizes with all streams). Otherwise, a plain new stream is created.
- **non_blocking** (*bool*) – If True, the stream does not synchronize with the NULL stream.

Variables **ptr** (*cupy.cuda.runtime.Stream*) – Raw stream handle. It can be passed to the CUDA Runtime API via ctypes.

add_callback(*callback*, *arg*)
Adds a callback that is called when all queued work is done.

Parameters

- **callback** (function) – Callback function. It must take three arguments (Stream object, int error status, and user data object), and returns nothing.
- **arg** (*object*) – Argument to the callback.

done
True if all work on this stream has been done.

record(*event=None*)
Records an event on the stream.

Parameters **event** (*None or cupy.cuda.Event*) – CUDA event. If None, then a new plain event is created and used.

Returns The recorded event.

Return type *cupy.cuda.Event*

See also:

cupy.cuda.Event.record()

synchronize()
Waits for the stream completing all queued work.

wait_event(*event*)
Makes the stream wait for an event.

The future work on this stream will be done after the event.

Parameters **event** (cupy.cuda.Event) – CUDA event.

class cupy.cuda.**Event**(*block=False*, *disable_timing=False*, *interprocess=False*)
CUDA event, a synchronization point of CUDA streams.

This class handles the CUDA event handle in RAII way, i.e., when an Event instance is destroyed by the GC, its handle is also destroyed.

Parameters

- **block** (*bool*) – If True, the event blocks on the *synchronize()* method.
- **disable_timing** (*bool*) – If True, the event does not prepare the timing data.
- **interprocess** (*bool*) – If True, the event can be passed to other processes.

Variables **ptr** (*cupy.cuda.runtime.Stream*) – Raw stream handle. It can be passed to the CUDA Runtime API via ctypes.

done
True if the event is done.

record(*stream=None*)
Records the event to a stream.

Parameters **stream** (cupy.cuda.Stream) – CUDA stream to record event. The null stream is used by default.

See also:

cupy.cuda.Stream.record()

synchronize()
Synchronizes all device work to the event.

If the event is created as a blocking event, it also blocks the CPU thread until the event is done.

cupy.cuda.**get_elapsed_time**(*start_event*, *end_event*)
Gets the elapsed time between two events.

Parameters

- **start_event** (Event) – Earlier event.
- **end_event** (Event) – Later event.

Returns Elapsed time in milliseconds.

Return type float

4.7 Kernel binary memoization

cupy.**memoize**()
Makes a function memoizing the result for each argument and device.

This decorator provides automatic memoization of the function result.

Parameters **for_each_device** (*bool*) – If True, it memoizes the results for each device. Otherwise, it memoizes the results only based on the arguments.

cupy.**clear_memo**()
Clears the memoized results for all functions decorated by memoize.

4.8 User-Defined Kernels

CuPy provides easy ways to define two types of CUDA kernels: elementwise kernels and reduction kernels. We first describe how to define and call elementwise kernels, and then describe how to define and call reduction kernels.

4.8.1 Basics of elementwise kernels

An elementwise kernel can be defined by the *ElementwiseKernel* class. The instance of this class defines a CUDA kernel which can be invoked by the __call__ method of this instance.

A definition of an elementwise kernel consists of four parts: an input argument list, an output argument list, a loop body code, and the kernel name. For example, a kernel that computes a squared difference $f(x, y) = (x - y)^2$ is defined as follows:

```
>>> squared_diff = cupy.ElementwiseKernel(
...     'float32 x, float32 y',
...     'float32 z',
...     'z = (x - y) * (x - y)',
...     'squared_diff')
```

The argument lists consist of comma-separated argument definitions. Each argument definition consists of a *type specifier* and an *argument name*. Names of NumPy data types can be used as type specifiers.

Note: n, i, and names starting with an underscore _ are reserved for the internal use.

The above kernel can be called on either scalars or arrays with broadcasting:

```
>>> x = cupy.arange(10, dtype=np.float32).reshape(2, 5)
>>> y = cupy.arange(5, dtype=np.float32)
>>> squared_diff(x, y)
array([[  0.,   0.,   0.,   0.,   0.],
       [ 25.,  25.,  25.,  25.,  25.]], dtype=float32)
>>> squared_diff(x, 5)
```

```
array([[ 25.,  16.,   9.,   4.,   1.],
       [  0.,   1.,   4.,   9.,  16.]], dtype=float32)
```

Output arguments can be explicitly specified (next to the input arguments):

```
>>> z = cupy.empty((2, 5), dtype=np.float32)
>>> squared_diff(x, y, z)
array([[  0.,   0.,   0.,   0.,   0.],
       [ 25.,  25.,  25.,  25.,  25.]], dtype=float32)
```

4.8.2 Type-generic kernels

If a type specifier is one character, then it is treated as a **type placeholder**. It can be used to define a type-generic kernels. For example, the above `squared_diff` kernel can be made type-generic as follows:

```
>>> squared_diff_generic = cupy.ElementwiseKernel(
...     'T x, T y',
...     'T z',
...     'z = (x - y) * (x - y)',
...     'squared_diff_generic')
```

Type placeholders of a same character in the kernel definition indicate the same type. The actual type of these placeholders is determined by the actual argument type. The ElementwiseKernel class first checks the output arguments and then the input arguments to determine the actual type. If no output arguments are given on the kernel invocation, then only the input arguments are used to determine the type.

The type placeholder can be used in the loop body code:

```
>>> squared_diff_generic = cupy.ElementwiseKernel(
...     'T x, T y',
...     'T z',
...     '''
...         T diff = x - y;
...         z = diff * diff;
...     ''',
...     'squared_diff_generic')
```

More than one type placeholder can be used in a kernel definition. For example, the above kernel can be further made generic over multiple arguments:

```
>>> squared_diff_super_generic = cupy.ElementwiseKernel(
...     'X x, Y y',
...     'Z z',
...     'z = (x - y) * (x - y)',
...     'squared_diff_super_generic')
```

Note that this kernel requires the output argument explicitly specified, because the type `Z` cannot be automatically determined from the input arguments.

4.8.3 Raw argument specifiers

The ElementwiseKernel class does the indexing with broadcasting automatically, which is useful to define most elementwise computations. On the other hand, we sometimes want to write a kernel with manual indexing for some arguments. We can tell the ElementwiseKernel class to use manual indexing by adding the `raw` keyword preceding the type specifier.

We can use the special variable `i` and method `_ind.size()` for the manual indexing. `i` indicates the index within the loop. `_ind.size()` indicates total number of elements to apply the elementwise operation. Note that it represents the size **after** broadcast operation.

For example, a kernel that adds two vectors with reversing one of them can be written as follows:

```
>>> add_reverse = cupy.ElementwiseKernel(
...     'T x, raw T y', 'T z',
...     'z = x + y[_ind.size() - i - 1]',
...     'add_reverse')
```

(Note that this is an artificial example and you can write such operation just by `z = x + y[::-1]` without defining a new kernel). A raw argument can be used like an array. The indexing operator `y[_ind.size() - i - 1]` involves an indexing computation on `y`, so `y` can be arbitrarily shaped and strode.

Note that raw arguments are not involved in the broadcasting. If you want to mark all arguments as `raw`, you must specify the `size` argument on invocation, which defines the value of `_ind.size()`.

4.8.4 Reduction kernels

Reduction kernels can be defined by the *ReductionKernel* class. We can use it by defining four parts of the kernel code:

1. Identity value: This value is used for the initial value of reduction.
2. Mapping expression: It is used for the pre-processing of each element to be reduced.
3. Reduction expression: It is an operator to reduce the multiple mapped values. The special variables `a` and `b` are used for its operands.
4. Post mapping expression: It is used to transform the resulting reduced values. The special variable `a` is used as its input. Output should be written to the output parameter.

ReductionKernel class automatically inserts other code fragments that are required for an efficient and flexible reduction implementation.

For example, L2 norm along specified axes can be written as follows:

```
>>> l2norm_kernel = cupy.ReductionKernel(
...     'T x',  # input params
...     'T y',  # output params
...     'x * x',  # map
...     'a + b',  # reduce
...     'y = sqrt(a)',  # post-reduction map
...     '0',  # identity value
...     'l2norm'  # kernel name
... )
>>> x = cupy.arange(10, dtype='f').reshape(2, 5)
>>> l2norm_kernel(x, axis=1)
array([  5.47722578,  15.96871948], dtype=float32)
```

Note: `raw` specifier is restricted for usages that the axes to be reduced are put at the head of the shape. It means, if you want to use `raw` specifier for at least one argument, the `axis` argument must be `0` or a contiguous increasing sequence of integers starting from `0`, like `(0, 1)`, `(0, 1, 2)`, etc.

4.8.5 Reference

class cupy.**ElementwiseKernel**
User-defined elementwise kernel.

This class can be used to define an elementwise kernel with or without broadcasting.

The kernel is compiled at an invocation of the *__call__()* method, which is cached for each device. The compiled binary is also cached into a file under the $HOME/.cupy/kernel_cache/ directory with a hashed file name. The cached binary is reused by other processes.

Parameters

- **in_params** (*str*) – Input argument list.
- **out_params** (*str*) – Output argument list.
- **operation** (*str*) – The body in the loop written in CUDA-C/C++.
- **name** (*str*) – Name of the kernel function. It should be set for readability of the performance profiling.
- **reduce_dims** (*bool*) – If False, the shapes of array arguments are kept within the kernel invocation. The shapes are reduced (i.e., the arrays are reshaped without copy to the minimum dimension) by default. It may make the kernel fast by reducing the index calculations.
- **options** (*list*) – Options passed to the nvcc command.
- **preamble** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the cu file.
- **loop_prep** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the kernel function definition and above the for loop.
- **after_loop** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the bottom of the kernel function definition.

__call__
Compiles and invokes the elementwise kernel.

The compilation runs only if the kernel is not cached. Note that the kernels with different argument dtypes or dimensions are not compatible. It means that single ElementwiseKernel object may be compiled into multiple kernel binaries.

Parameters

- **args** – Arguments of the kernel.
- **size** (*int*) – Range size of the indices. If specified, the variable n is set to this value. Otherwise, the result of broadcasting is used to determine the value of n.

Returns Arrays are returned according to the out_params argument of the __init__ method.

class cupy.**ReductionKernel**
User-defined reduction kernel.

This class can be used to define a reduction kernel with or without broadcasting.

The kernel is compiled at an invocation of the *__call__()* method, which is cached for each device. The compiled binary is also cached into a file under the $HOME/.cupy/kernel_cache/ directory with a hashed file name. The cached binary is reused by other processes.

Parameters

- **in_params** (*str*) – Input argument list.
- **out_params** (*str*) – Output argument list.
- **map_expr** (*str*) – Mapping expression for input values.
- **reduce_expr** (*str*) – Reduction expression.
- **post_map_expr** (*str*) – Mapping expression for reduced values.
- **identity** (*str*) – Identity value for starting the reduction.
- **name** (*str*) – Name of the kernel function. It should be set for readability of the performance profiling.
- **reduce_type** (*str*) – Type of values to be used for reduction. This type is used to store the special variables `a`.
- **reduce_dims** (*bool*) – If `True`, input arrays are reshaped without copy to smaller dimensions for efficiency.
- **preamble** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the cu file.
- **options** (*tuple of str*) – Additional compilation options.

__call__()

Compiles and invokes the reduction kernel.

The compilation runs only if the kernel is not cached. Note that the kernels with different argument dtypes, ndims, or axis are not compatible. It means that single ReductionKernel object may be compiled into multiple kernel binaries.

Parameters **args** – Arguments of the kernel.

Returns Arrays are returned according to the `out_params` argument of the `__init__` method.

4.9 Testing Modules

CuPy offers testing utilities to support unit testing. They are under namespace *cupy.testing*.

4.9.1 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*.

cupy.testing.**assert_allclose**(*actual*, *desired*, *rtol=1e-07*, *atol=0*, *err_msg=''*, *verbose=True*)

Raises an AssertionError if objects are not equal up to desired tolerance.

Parameters

- **actual** (*numpy.ndarray or cupy.ndarray*) – The actual object to check.
- **desired** (*numpy.ndarray or cupy.ndarray*) – The desired, expected object.
- **rtol** (*float*) – Relative tolerance.
- **atol** (*float*) – Absolute tolerance.
- **err_msg** (*str*) – The error message to be printed in case of failure.

- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.

See also:

`numpy.testing.assert_allclose()`

cupy.testing.**assert_array_almost_equal**(*x*, *y*, *decimal=6*, *err_msg=''*, *verbose=True*)
Raises an AssertionError if objects are not equal up to desired precision.

Parameters

- **x** (*numpy.ndarray or cupy.ndarray*) – The actual object to check.
- **y** (*numpy.ndarray or cupy.ndarray*) – The desired, expected object.
- **decimal** (*int*) – Desired precision.
- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.

See also:

`numpy.testing.assert_array_almost_equal()`

cupy.testing.**assert_array_almost_equal_nulp**(*x*, *y*, *nulp=1*)
Compare two arrays relatively to their spacing.

Parameters

- **x** (*numpy.ndarray or cupy.ndarray*) – The actual object to check.
- **y** (*numpy.ndarray or cupy.ndarray*) – The desired, expected object.
- **nulp** (*int*) – The maximum number of unit in the last place for tolerance.

See also:

`numpy.testing.assert_array_almost_equal_nulp()`

cupy.testing.**assert_array_max_ulp**(*a*, *b*, *maxulp=1*, *dtype=None*)
Check that all items of arrays differ in at most N Units in the Last Place.

Parameters

- **a** (*numpy.ndarray or cupy.ndarray*) – The actual object to check.
- **b** (*numpy.ndarray or cupy.ndarray*) – The desired, expected object.
- **maxulp** (*int*) – The maximum number of units in the last place that elements of `a` and `b` can differ.
- **dtype** (*numpy.dtype*) – Data-type to convert `a` and `b` to if given.

See also:

`numpy.testing.assert_array_max_ulp()`

cupy.testing.**assert_array_equal**(*x*, *y*, *err_msg=''*, *verbose=True*)
Raises an AssertionError if two array_like objects are not equal.

Parameters

- **x** (*numpy.ndarray or cupy.ndarray*) – The actual object to check.
- **y** (*numpy.ndarray or cupy.ndarray*) – The desired, expected object.
- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.

See also:

numpy.testing.assert_array_equal()

cupy.testing.**assert_array_list_equal**(*xlist*, *ylist*, *err_msg=''*, *verbose=True*)
Compares lists of arrays pairwise with assert_array_equal.

Parameters

- **x** (*array_like*) – Array of the actual objects.
- **y** (*array_like*) – Array of the desired, expected objects.
- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If True, the conflicting values are appended to the error message.

Each element of x and y must be either numpy.ndarray or *cupy.ndarray*. x and y must have same length. Otherwise, this function raises AssertionError. It compares elements of x and y pairwise with *assert_array_equal()* and raises error if at least one pair is not equal.

See also:

numpy.testing.assert_array_equal()

cupy.testing.**assert_array_less**(*x*, *y*, *err_msg=''*, *verbose=True*)
Raises an AssertionError if array_like objects are not ordered by less than.

Parameters

- **x** (*numpy.ndarray or cupy.ndarray*) – The smaller object to check.
- **y** (*numpy.ndarray or cupy.ndarray*) – The larger object to compare.
- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If True, the conflicting values are appended to the error message.

See also:

numpy.testing.assert_array_less()

4.9.2 NumPy-CuPy Consistency Check

The following decorators are for testing consistency between CuPy's functions and corresponding NumPy's ones.

cupy.testing.**numpy_cupy_allclose**(*rtol=1e-07*, *atol=0*, *err_msg=''*, *verbose=True*, *name='xp'*, *type_check=True*, *accept_error=True*)
Decorator that checks NumPy results and CuPy ones are close.

Parameters

- **rtol** (*float*) – Relative tolerance.
- **atol** (*float*) – Absolute tolerance
- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If True, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either numpy or cupy module.
- **type_check** (*bool*) – If True, consistency of dtype is also checked.
- **accept_error** (*bool*) – If True, errors are not raised as long as the errors occurred are identical between NumPy and CuPy.

Decorated test fixture is required to return the arrays whose values are close between `numpy` case and `cupy` case. For example, this test case checks `numpy.zeros` and `cupy.zeros` should return same value.

```
>>> from cupy import testing
... @testing.gpu
... class TestFoo(unittest.TestCase):
...
...     @testing.numpy_cupy_allclose()
...     def test_foo(self, xp):
...         # ...
...         # Prepare data with xp
...         # ...
...
...         xp_result = xp.zeros(10)
...         return xp_result
```

See also:

cupy.testing.assert_allclose()

cupy.testing.**numpy_cupy_array_almost_equal**(*decimal=6*, *err_msg=''*, *verbose=True*, *name='xp'*, *type_check=True*, *accept_error=True*)

Decorator that checks NumPy results and CuPy ones are almost equal.

Parameters

- **decimal** (*int*) – Desired precision.
- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept_error** (*bool*) – If `True`, errors are not raised as long as the errors occurred are identical between NumPy and CuPy.

Decorated test fixture is required to return the same arrays in the sense of *cupy.testing.assert_array_almost_equal()* (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

cupy.testing.assert_array_almost_equal()

cupy.testing.**numpy_cupy_array_almost_equal_nulp**(*nulp=1*, *name='xp'*, *type_check=True*, *accept_error=True*)

Decorator that checks results of NumPy and CuPy are equal w.r.t. spacing.

Parameters

- **nulp** (*int*) – The maximum number of unit in the last place for tolerance.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept_error** (*bool*) – If `True`, errors are not raised as long as the errors occurred are identical between NumPy and CuPy.

Decorated test fixture is required to return the same arrays in the sense of *cupy.testing.assert_array_almost_equal_nulp()* (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

cupy.testing.assert_array_almost_equal_nulp()

cupy.testing.**numpy_cupy_array_max_ulp**(*maxulp=1*, *dtype=None*, *name='xp'*, *type_check=True*, *accept_error=True*)

Decorator that checks results of NumPy and CuPy ones are equal w.r.t. ulp.

Parameters

- **maxulp** (*int*) – The maximum number of units in the last place that elements of resulting two arrays can differ.
- **dtype** (*numpy.dtype*) – Data-type to convert the resulting two array to if given.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept_error** (*bool*) – If `True`, errors are not raised as long as the errors occurred are identical between NumPy and CuPy.

Decorated test fixture is required to return the same arrays in the sense of *assert_array_max_ulp()* (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

cupy.testing.assert_array_max_ulp()

cupy.testing.**numpy_cupy_array_equal**(*err_msg=''*, *verbose=True*, *name='xp'*, *type_check=True*, *accept_error=True*)

Decorator that checks NumPy results and CuPy ones are equal.

Parameters

- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept_error** (*bool*) – If `True`, errors are not raised as long as the errors occurred are identical between NumPy and CuPy.

Decorated test fixture is required to return the same arrays in the sense of *numpy_cupy_array_equal()* (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

cupy.testing.assert_array_equal()

cupy.testing.**numpy_cupy_array_list_equal**(*err_msg=''*, *verbose=True*, *name='xp'*)

Decorator that checks the resulting lists of NumPy and CuPy's one are equal.

Parameters

- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.

Decorated test fixture is required to return the same list of arrays (except the type of array module) even if `xp` is `numpy` or `cupy`.

See also:

cupy.testing.assert_array_list_equal()

cupy.testing.**numpy_cupy_array_less**(*err_msg=''*, *verbose=True*, *name='xp'*, *type_check=True*, *accept_error=True*)

Decorator that checks the CuPy result is less than NumPy result.

Parameters

- **err_msg** (*str*) – The error message to be printed in case of failure.
- **verbose** (*bool*) – If `True`, the conflicting values are appended to the error message.
- **name** (*str*) – Argument name whose value is either `numpy` or `cupy` module.
- **type_check** (*bool*) – If `True`, consistency of dtype is also checked.
- **accept_error** (*bool*) – If `True`, errors are not raised as long as the errors occurred are identical between NumPy and CuPy.

Decorated test fixture is required to return the smaller array when `xp` is `cupy` than the one when `xp` is `numpy`.

See also:

cupy.testing.assert_array_less()

cupy.testing.**numpy_cupy_raises**(*name='xp'*)

Decorator that checks the NumPy and CuPy throw same errors.

Parameters

- **name** (*str*) – Argument name whose value is either
- **or cupy module.** (*numpy*) –

Decorated test fixture is required throw same errors even if `xp` is `numpy` or `cupy`.

See also:

cupy.testing.assert_array_less()

4.9.3 Parameterized dtype Test

The following decorators offers the standard way for parameterized test with respect to single or the combination of dtype(s).

cupy.testing.**for_dtypes**(*dtypes*, *name='dtype'*)

Decorator for parameterized dtype test.

Parameters

- **dtypes** (*list of dtypes*) – dtypes to be tested.
- **name** (*str*) – Argument name to which specified dtypes are passed.

This decorator adds a keyword argument specified by `name` to the test fixture. Then, it runs the fixtures in parallel by passing the each element of `dtypes` to the named argument.

cupy.testing.**for_all_dtypes**(*name='dtype'*, *no_float16=False*, *no_bool=False*)

Decorator that checks the fixture with all dtypes.

Parameters

- **name** (*str*) – Argument name to which specified dtypes are passed.
- **no_float16** (*bool*) – If, True, `numpy.float16` is omitted from candidate dtypes.
- **no_bool** (*bool*) – If, True, `numpy.bool_` is omitted from candidate dtypes.

dtypes to be tested: `numpy.float16` (optional), `numpy.float32`, `numpy.float64`, `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, `numpy.dtype('q')`, `numpy.dtype('B')`, `numpy.dtype('H')`, `numpy.dtype('I')`, `numpy.dtype('L')`, `numpy.dtype('Q')`, and `numpy.bool_` (optional).

The usage is as follows. This test fixture checks if `cPickle` successfully reconstructs *cupy.ndarray* for various dtypes. `dtype` is an argument inserted by the decorator.

```
>>> from cupy import testing
>>> @testing.gpu
... class TestNpz(unittest.TestCase):
...
...     @testing.for_all_dtypes()
...     def test_pickle(self, dtype):
...         a = testing.shaped_arange((2, 3, 4), dtype=dtype)
...         s = six.moves.cPickle.dumps(a)
...         b = six.moves.cPickle.loads(s)
...         testing.assert_array_equal(a, b)
```

Typically, we use this decorator in combination with decorators that check consistency between NumPy and CuPy like *cupy.testing.numpy_cupy_allclose()*. The following is such an example.

```
>>> from cupy import testing
>>> @testing.gpu
... class TestMean(unittest.TestCase):
...
...     @testing.for_all_dtypes()
...     @testing.numpy_cupy_allclose()
...     def test_mean_all(self, xp, dtype):
...         a = testing.shaped_arange((2, 3), xp, dtype)
...         return a.mean()
```

See also:

cupy.testing.for_dtypes()

cupy.testing.**for_float_dtypes**(*name='dtype'*, *no_float16=False*)

Decorator that checks the fixture with all float dtypes.

Parameters

- **name** (*str*) – Argument name to which specified dtypes are passed.
- **no_float16** (*bool*) – If, True, `numpy.float16` is omitted from candidate dtypes.

dtypes to be tested are `numpy.float16` (optional), `numpy.float32`, and `numpy.float64`.

See also:

cupy.testing.for_dtypes(), *cupy.testing.for_all_dtypes()*

cupy.testing.**for_signed_dtypes**(*name='dtype'*)

Decorator that checks the fixture with signed dtypes.

Parameters **name** (*str*) – Argument name to which specified dtypes are passed.

dtypes to be tested are `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, and `numpy.dtype('q')`.

See also:

cupy.testing.for_dtypes(), *cupy.testing.for_all_dtypes()*

cupy.testing.**for_unsigned_dtypes**(*name='dtype'*)
Decorator that checks the fixture with all dtypes.

Parameters **name** (*str*) – Argument name to which specified dtypes are passed.

dtypes to be tested are `numpy.dtype('B')`, `numpy.dtype('H')`,

`numpy.dtype('I')`, `numpy.dtype('L')`, and `numpy.dtype('Q')`.

See also:

cupy.testing.for_dtypes(), *cupy.testing.for_all_dtypes()*

cupy.testing.**for_int_dtypes**(*name='dtype'*, *no_bool=False*)
Decorator that checks the fixture with integer and optionally bool dtypes.

Parameters

- **name** (*str*) – Argument name to which specified dtypes are passed.
- **no_bool** (*bool*) – If `True`, `numpy.bool_` is omitted from candidate dtypes.

dtypes to be tested are `numpy.dtype('b')`, `numpy.dtype('h')`, `numpy.dtype('i')`, `numpy.dtype('l')`, `numpy.dtype('q')`, `numpy.dtype('B')`, `numpy.dtype('H')`, `numpy.dtype('I')`, `numpy.dtype('L')`, `numpy.dtype('Q')`, and `numpy.bool_` (optional).

See also:

cupy.testing.for_dtypes(), *cupy.testing.for_all_dtypes()*

cupy.testing.**for_dtypes_combination**(*types*, *names=['dtype']*, *full=None*)
Decorator that checks the fixture with a product set of dtypes.

Parameters

- **types** (*list of dtypes*) – dtypes to be tested.
- **names** (*list of str*) – Argument names to which dtypes are passed.
- **full** (*bool*) – If `True`, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see the description below).

Decorator adds the keyword arguments specified by `names` to the test fixture. Then, it runs the fixtures in parallel with passing (possibly a subset of) the product set of dtypes. The range of dtypes is specified by `types`.

The combination of dtypes to be tested changes depending on the option `full`. If `full` is `True`, all combinations of `types` are tested. Sometimes, such an exhaustive test can be costly. So, if `full` is `False`, only the subset of possible combinations is tested. Specifically, at first, the shuffled lists of `types` are made for each argument name in `names`. Let the lists be `D1`, `D2`, ..., `Dn` where n is the number of arguments. Then, the combinations to be tested will be `zip(D1, ..., Dn)`. If `full` is `None`, the behavior is switched by setting the environment variable `CUPY_TEST_FULL_COMBINATION=1`.

For example, let `types` be `[float16, float32, float64]` and `names` be `['a_type', 'b_type']`. If `full` is `True`, then the decorated test fixture is executed with all 2^3 patterns. On the other hand, if `full` is `False`, shuffled lists are made for `a_type` and `b_type`. Suppose the lists are `(16, 64, 32)` for `a_type` and `(32, 64, 16)` for `b_type` (prefixes are removed for short). Then the combinations of `(a_type, b_type)` to be tested are `(16, 32)`, `(64, 64)` and `(32, 16)`.

cupy.testing.**for_all_dtypes_combination**(*names=['dtyes']*, *no_float16=False*, *no_bool=False*, *full=None*)
Decorator that checks the fixture with a product set of all dtypes.

Parameters

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **no_float16** (*bool*) – If `True`, `numpy.float16` is omitted from candidate dtypes.
- **no_bool** (*bool*) – If `True`, `numpy.bool_` is omitted from candidate dtypes.
- **full** (*bool*) – If `True`, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in *cupy.testing.for_dtypes_combination()*).

See also:

cupy.testing.for_dtypes_combination()

cupy.testing.**for_signed_dtypes_combination**(*names=['dtype'], full=None*)
Decorator for parameterized test w.r.t. the product set of signed dtypes.

Parameters

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **full** (*bool*) – If `True`, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in *cupy.testing.for_dtypes_combination()*).

See also:

cupy.testing.for_dtypes_combination()

cupy.testing.**for_unsigned_dtypes_combination**(*names=['dtype'], full=None*)
Decorator for parameterized test w.r.t. the product set of unsigned dtypes.

Parameters

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **full** (*bool*) – If `True`, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in *cupy.testing.for_dtypes_combination()*).

See also:

cupy.testing.for_dtypes_combination()

cupy.testing.**for_int_dtypes_combination**(*names=['dtype'], no_bool=False, full=None*)
Decorator for parameterized test w.r.t. the product set of int and boolean.

Parameters

- **names** (*list of str*) – Argument names to which dtypes are passed.
- **no_bool** (*bool*) – If `True`, `numpy.bool_` is omitted from candidate dtypes.
- **full** (*bool*) – If `True`, then all combinations of dtypes will be tested. Otherwise, the subset of combinations will be tested (see description in *cupy.testing.for_dtypes_combination()*).

See also:

cupy.testing.for_dtypes_combination()

CHAPTER 5

Chainer 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.

5.1 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 to 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.

5.2 Release and Milestone

We are using GitHub Flow as our basic working process. In particular, we are using the master branch for our development, and releases are made as tags.

Releases are classified into three groups: major, minor, and revision. This classification is based on following criteria:

- **Major update** contains disruptive changes that break the backward compatibility.
- **Minor update** contains additions and extensions to the APIs keeping the supported backward compatibility.
- **Revision update** contains improvements on the API implementations without changing any API specification.

The release classification is reflected into the version number x.y.z, where x, y, and z corresponds to major, minor, and revision updates, respectively.

We sets milestones for some future releases. A milestone for a revision release is set right after the last release. On the other hand, a milestone for a minor or major release is set four weeks prior to its due.

See also API Compatibility Policy.

5.3 Issues and PRs

Issues and PRs are classified into following categories:

- **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
- **Other**: other issues and PRs

Issues and PRs are labeled by these categories. This classification is often reflected into its corresponding release category: Feature issues/PRs are contained into minor/major releases and NoCompat issues/PRs are contained into major releases, while other issues/PRs can be contained into any releases including revision ones.

On registering an issue, write precise explanations on what you want Chainer to be. Bug reports must include necessary and sufficient conditions to reproduce the bugs. Feature requests must include **what** you want to do (and **why** you want to do, if needed). You can contain your thoughts on **how** to realize it into the feature requests, though **what** part is most important for discussions.

> **Warning:** If you have a question on usages of Chainer, it is highly recommended to send a post to Chainer User Group instead of the issue tracker. The issue tracker is not a place to share knowledge on practices. We may redirect question issues to Chainer User Group.

If you can write code to fix an issue, send a PR to the master branch. Before writing your code for PRs, read through the *Coding Guidelines*. The description of any PR must contain a precise explanation of **what** and **how** you want to do; it is the first documentation of your code for developers, 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 need 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. Since this service is running on our internal server, we do not use it for automatic PR tests to keep the server secure.

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.

5.4 Coding Guidelines

We use PEP8 and a part of OpenStack Style Guidelines related to general coding style as our basic style guidelines.

To check your code, use `flake8` command installed by `hacking` package:

```
$ pip install hacking
$ flake8 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.

5.5 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:

```
$ pip install nose mock
```

In order to run unit tests at the repository root, you first have to build Cython files in place by running the following command:

```
$ python setup.py develop
```

Once the Cython modules are built, you can run unit tests simply by running `nosetests` command at the repository root:

```
$ nosetests
```

It requires CUDA by default. In order to run unit tests that do not require CUDA, pass `--attr='!gpu'` option to the `nosetests` command:

```
$ 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 `--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:

```
$ nosetests path/to/gpu/test.py --attr='gpu<2'
```

Tests are put into the `tests/chainer_tests`, `tests/cupy_tests` and `tests/install_tests` directories. These have the same structure as that of `chainer`, `cupy` and `install` directories, respectively. 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.

Following this naming convention, you can run all the tests by just typing `nosetests` at the repository root:

```
$ nosetests
```

Or you can also specify a root directory to search test scripts from:

```
$ nosetests tests/chainer_tests  # to just run tests of Chainer
$ nosetests tests/cupy_tests     # to just run tests of CuPy
$ nosetests tests/install_tests  # to just run tests of installation modules
```

If you modify the code related to existing unit tests, you must run appropriate commands.

Note: CuPy tests include type-exhaustive test functions which take long time to execute. If you are running tests on a multi-core machine, you can parallelize the tests by following options:

```
$ nosetests --processes=12 --process-timeout=1000 tests/cupy_tests
```

The magic numbers can be modified for your usage. Note that some tests require many CUDA compilations, which require a bit long time. Without the `process-timeout` option, the timeout is set shorter, causing timeout failures for many test cases.

There are many examples of unit tests under the `tests` directory. They simply use the `unittest` package of the standard library.

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 the `chainer.testing.attr.gpu` decorator (or `cupy.testing.attr.gpu` for testing CuPy APIs):

```
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 `nosetests` 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()` or `cupy.testing.attr.multi_gpu()` decorators instead:

```
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):
        ...
```

Once you send a pull request, your code is automatically tested by Travis-CI **with –attr='!gpu' 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.

CHAPTER 6

API Compatibility Policy

This document expresses the design policy on compatibilities of Chainer APIs. Development team should obey 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.

6.1 Targeted Versions

This policy is applied to Chainer of versions v1.5.1 and higher. Note that this policy is not applied to Chainer of lower versions.

6.2 Versioning and Backward Compatibilities

The updates of Chainer are classified into three levels: major, minor, and revision. These types have distinct levels of backward compatibilities.

- **Major update** contains disruptive changes that break the backward compatibility.
- **Minor update** contains addition and extension to the APIs keeping the supported backward compatibility.
- **Revision update** contains improvements on the API implementations without changing any API specifications.

Note that we do not support full backward compatibility, which is almost infeasible for Python-based APIs, since there is no way to completely hide the implementation details.

6.3 Processes to Break Backward Compatibilities

6.3.1 Deprecation, Dropping, and Its Preparation

Any APIs may be *deprecated* at some minor updates. In such a case, the deprecation note is added to the API documentation, and the API implementation is changed to fire deprecation warning (if possible). There should be another way to reimplement the same things previously written with the deprecated APIs.

Any APIs may be marked as *to be dropped in the future*. In such a case, the dropping is stated in the documentation with the major version number on which the API is planned to be dropped, and the API implementation is changed to fire the future warning (if possible).

The actual dropping should be done through the following steps:

- Make the API deprecated. At this point, users should not need the deprecated API in their new application codes.
- After that, mark the API as *to be dropped in the future*. It must be done in the minor update different from that of the deprecation.
- At the major version announced in the above update, drop the API.

Consequently, it takes at least two minor versions to drop any APIs after the first deprecation. Since each minor update is made for every six weeks, this dropping procedure takes at least 12 weeks (~ 3 months).

6.3.2 API Changes and Its Preparation

Any APIs may be marked as *to be changed in the future* for changes without backward compatibility. In such a case, the change is stated in the documentation with the version number on which the API is planned to be changed, and the API implementation is changed to fire the future warning on the certain usages.

The actual change should be done in the following steps:

- Announce that the API will be changed in the future. At this point, the actual version of change need not be accurate.
- After the announcement, mark the API as *to be changed in the future* with version number of planned changes. At this point, users should not use the marked API in their new application codes.
- At the major update announced in the above update, change the API.

6.4 Supported Backward Compatibility

This section defines backward compatibilities that minor updates must maintain.

6.4.1 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 minor/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.

6.4.2 Undocumented behaviors

Behaviors of Chainer implementation not stated in the documentation are undefined. Undocumented behaviors are not guaranteed to be stable between different minor/revision versions.

Minor update may contain changes to undocumented behaviors. For example, suppose an API X is added at the minor update. In the previous version, attempts to use X cause AttributeError. This behavior is not stated in the documentation, so this is undefined. Thus, adding the API X in minor version is permissible.

Revision update may also contain changes to undefined behaviors. Typical example 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.**

6.4.3 Documentation Error

Compatibility is basically determined based on the documentation, though 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 MUST NOT fix the documentation and implementation of the same functionality at the same time in revision updates as "bug fix". Such a change completely breaks the backward compatibility. If you want to fix the bugs in both sides, first fix the documentation to fit it into the implementation, and start the API changing procedure described above.

6.4.4 Object Attributes and Properties

Object attributes and properties are sometimes replaced by each other at minor updates. It does not break the user codes, except the codes depend on how the attributes and properties are implemented.

6.4.5 Functions and Methods

Methods may be replaced by callable attributes keeping the compatibility of parameters and return values in minor updates. It does not break the user codes, except the codes depend on how the methods and callable attributes are implemented.

6.4.6 Exceptions and Warnings

The specifications of raising exceptions are considered as a part of standard backward compatibilities. No exception is raised in the future versions with correct usages that the documentation allows, unless the API changing process is completed.

On the other hand, warnings may be added at any minor updates for any APIs. It means minor updates do not keep backward compatibility of warnings.

6.5 Model Format Compatibility

Objects serialized by official serializers that Chainer provides are correctly loaded with the higher (future) versions. They might not be correctly loaded with Chainer of the lower versions.

Note: Current serialization APIs do not support versioning (at least in v1.6.1). It prevents us from introducing changes in the layout of objects that support serialization. We are discussing about introducing versioning in serialization APIs.

6.6 Installation Compatibility

The installation process is another concern of compatibilities. We support environmental compatibilities in the following ways.

- Any changes of dependent libraries that force modifications on the existing environments must be done in 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)
- Supporting optional packages/libraries may be done in minor updates (e.g. supporting h5py in optional features).

Note: The installation compatibility does not guarantee that all the features of Chainer correctly run on supported environments. It may contain bugs that only occurs in certain environments. Such bugs should be fixed in some updates.

CHAPTER 7

Tips and FAQs

7.1 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.

7.2 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 OpneBLAS using Homebrew.

```
$ brew tap homebrew/science
$ brew install openblas
$ brew install numpy --with-openblas
```

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*.

```
>>> 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

8.1 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.

		Chainer	Theano-based	Torch7	Caffe
Specs	Scripting	Python	Python	LuaJIT	Python
	Net definition language	Python	Python	LuaJIT	Protocol Buffers
	Define-by-Run scheme	Y			
	CPU Array backend	NumPy	NumPy	Tensor	
	GPU Array backend	CuPy	CudaNdarray [1]	CudaTensor	
NNs	Reverse-mode AD	Y	Y	Y	Y
	Basic RNN support	Y	Y	Y (`nnx`)	#2033
	Variable-length loops	Y	Y (`scan`)		
	Stateful RNNs [2]	Y		Y [6]	
	Per-batch architectures	Y			
Perf	CUDA support	Y	Y	Y	Y
	cuDNN support	Y	Y	Y (`cudnn.torch`)	Y
	FFT-based convolution		Y	Y (`fbcunn`)	#544
	CPU/GPU generic coding [3]	Y	[4]	Y	
	Multi GPU (data parallel)	Y		Y (`fbcunn`)	Y
	Multi GPU (model parallel)	Y		Y (`fbcunn`)	
Misc	Type checking	Y	Y	Y	N/A
	Model serialization	Y	Y (`pickle`)	Y	Y
	Caffe reference model	Y	[5]	Y (`loadcaffe`)	Y

[1] They are also developing 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.

[6] Also available in the *Torch RNN package <https://github.com/Element-Research/rnn>*

[3] This row shows whether each array API supports unified codes for CPU and GPU.

[4] The array backend of Theano does not have compatible interface with NumPy, though most users write code on Theano variables, which is generic for CPU and GPU.

[5] Depending on the frameworks.

8.2 Benchmarks

We are preparing for the benchmarks.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[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

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Z