Chainer Documentation

Release 1.0.1

Preferred Networks, inc. and Preferred Infrastructure, inc.

June 23, 2015

Contents

This is the Chainer documentation.

CHAPTER 1

Chainer Tutorial

1.1 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 parameterized functions and their gradient computation
- Management of a set of parameterized functions (a.k.a. "model" in most frameworks)
- Parameter optimization

After reading this section, you will be able to:

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

1.1.1 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 minibatches. 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
from chainer import cuda, Function, FunctionSet, gradient_check, Variable, optimizers
import chainer.functions as F
```

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

1.1.2 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 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.

1.1.3 Parameterized functions

In order to write neural networks, we have to use some *parameterized functions* and optimize their parameters. As noted above, functions are predefined in `functions` module, which also includes parameterized functions.

One of the most fundamental parameterized functions is the *Linear* function (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. A linear function from three-dimensional space to two-dimensional space is defined by:

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

Note: Most functions only accept minibatch input, where the first dimension of input arrays is considered as the *batch dimension*. In the above Linear function case, input must has shape of (N, 3), where N is the minibatch size.

The parameters of Linear function are stored in `W` and `b` attributes. By default, the matrix W is initialized randomly, while the vector b is initialized with zeros.

```
>>> f.W
array([[ 1.33545339, -0.01839679,  0.7662735 ],
       [-1.21562171, -0.44784674, -0.07128379]], dtype=float32)
>>> f.b
array([ 0.,  0.], dtype=float32)
```

Instances of a parameterized function class act like usual functions:

```
>>> x = Variable(np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float32))
>>> y = f(x)
>>> y.data
array([[ 3.5974803 , -2.3251667 ],
       [ 9.84747124, -7.52942371]], 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. Gradients of Linear function are stored in `gW` and `gb` attributes:

```
>>> f.gW.fill(0)
>>> f.gb.fill(0)
```

Note: This procedure is simplified by FunctionSet and Optimizer, which we will see in the next seciton.

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.gW
array([[ 5.,  7.,  9.],
       [ 5.,  7.,  9.]], dtype=float32)
```

```
>>> f.gb
array([ 2.,  2.], dtype=float32)
```

1.1.4 FunctionSet

Most neural network architectures contain multiple parameterized functions. *FunctionSet* makes it easy to manage them. This class acts like a simple object, with attributes initialized by keyword arguments of the initializer:

```
>>> model = FunctionSet(
...     l1 = F.Linear(4, 3),
...     l2 = F.Linear(3, 2),
... )
>>> model.l1
<chainer.functions.linear.Linear object at 0x7f7f03e4f350>
>>> model.l2
<chainer.functions.linear.Linear object at 0x7f7f03e4f590>
```

You can also add additional functions later by setting attributes:

```
>>> model.l3 = F.Linear(2, 2)
```

Since the `model` is just an object with functions stored as its attributes, we can use these functions in forward computation:

```
>>> x = Variable(np.array([[1, 2, 3, 4], [5, 6, 7, 8]], dtype=np.float32))
>>> h1 = model.l1(x)
>>> h2 = model.l2(h1)
>>> h3 = model.l3(h2)
```

One of the features of FunctionSet is the ability to collect parameters and gradients. A tuple of all parameters and a tuple of all gradients are extracted by *FunctionSet.parameters* and *FunctionSet.gradients* properties, respectively.

1.1.5 Optimizer

Optimizer is the last core feature of Chainer described in this section. It runs a numerical optimization algorithm given tuples of parameters and gradients. Many algorithms are implemented in `optimizers` module. Here we use the simplest one, called Stochastic Gradient Descent:

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

The method *setup()* prepares for the optimization given parameters and gradients. The interface is designed to match the return values of the *FunctionSet.collect_parameters()* method.

Note: Since Optimizer does not know the functions that actually own the parameters and gradients, once parameters and gradients are given to Optimizer, functions must use same parameter and gradient array objects throughout all forward/backward computations.

In order to run optimization, you first have to compute gradients. Zeroing the initial gradient arrays are simply done by calling *zero_grads()* method:

```
>>> optimizer.zero_grads()
```

We have done the zeroing manually in the previous section. The line above is an equivalent and simpler way to initialize the gradients.

Then, after computing gradient of each parameter, *update()* method runs one iteration of optimization:

```
>>> (compute gradient)
>>> optimizer.update()
```

Optimizer also contains some features related to parameter and gradient manipulation, e.g. weight decay and gradient clipping.

1.1.6 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 the long-standing de-facto "hello world" of machine learning. This MNIST example is also found in `examples/mnist` directory of the official repository.

In order to use MNIST, `sklearn.datasets.fetch_mldata()` function of scikit-learn is useful:

```
>>> from sklearn.datasets import fetch_mldata
>>> mnist = fetch_mldata('MNIST original')
```

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. Before defining the forward routine, we have to prepare our parameterized functions:

```
>>> model = FunctionSet(
...     l1 = F.Linear(784, 100),
...     l2 = F.Linear(100, 100),
...     l3 = F.Linear(100,  10),
... )
>>> optimizer = optimizers.SGD()
>>> optimizer.setup(model.collect_parameters())
```

Note that `model.l3` is the final linear layer whose output corresponds to the ten digits. We also set up the optimizer here.

Now we can define the forward routine using these Linear functions. Typically it is defined as a simple python function given input arrays:

```
>>> def forward(x_data, y_data):
...     x = Variable(x_data)
...     t = Variable(y_data)
...     h1 = F.relu(model.l1(x))
...     h2 = F.relu(model.l2(h1))
...     y = model.l3(h2)
...     return F.softmax_cross_entropy(y, t), F.accuracy(y, t)
```

This function uses *functions.relu()* as an activation function. Since ReLU does not have parameters to optimize, it does not need to be included in *model*. *functions.softmax_cross_entropy()* computes the loss function of softmax regression. *functions.accuracy()* computes the classification accuracy of this minibatch.

Finally, we can write a learning loop as following:

```
>>> batchsize = 100
>>> for epoch in xrange(20):
...     print 'epoch', epoch
...     indexes = np.random.permutation(60000)
...     for i in xrange(0, 60000, batchsize):
...         x_batch = x_train[indexes[i : i + batchsize]]
...         y_batch = y_train[indexes[i : i + batchsize]]
...
...         optimizer.zero_grads()
...         loss, accuracy = forward(x_batch, y_batch)
...         loss.backward()
...         optimizer.update()
```

Only the last four lines are the code related to Chainer, which are already described above.

Here you 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 xrange(0, 10000, batchsize):
...     x_batch = x_test[i : i + batchsize]
...     y_batch = y_test[i : i + batchsize]
...     loss, accuracy = forward(x_batch, y_batch)
...     sum_loss      += loss.data * batchsize
...     sum_accuracy  += accuracy.data * batchsize
...
>>> mean_loss     = sum_loss / 10000
>>> mean_accuracy = sum_accuracy / 10000
```

The example code 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).

1.2 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

1.2.1 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 that there are 1,000 different word types, and that we use 100 dimensional real vectors to represent each word (a.k.a. word embedding).

Before writing the forward computation, we have to define parameterized functions:

```
model = FunctionSet(
    embed  = F.EmbedID(1000, 100),
    x_to_h = F.Linear(100,   50),
    h_to_h = F.Linear( 50,   50),
    h_to_y = F.Linear( 50, 1000),
)
optimizer = optimizers.SGD()
optimizer.setup(model.collect_parameters())
```

Here *EmbedID* is a parameterized function class for word embedding. It converts input integers into corresponding fixed-dimensional embedding vectors. Other Linear layers represent the transformation as their names indicate. Here we use 50 hidden units.

Then, we can write down the forward computation. Suppose that the input word sequence is given as a list of integer arrays. The forward computation is simply written with a for loop:

```
def forward_one_step(h, cur_word, next_word, volatile=False):
    word = Variable(cur_word, volatile=volatile)
    t    = Variable(next_word, volatile=volatile)
    x    = F.tanh(model.embed(word))
    h    = F.tanh(model.x_to_h(x) + model.h_to_h(h))
    y    = model.h_to_y(h)
    loss = F.softmax_cross_entropy(y, t)
    return h, loss

def forward(x_list, volatile=False):
    h = Variable(np.zeros((50,), dtype=np.float32), volatile=volatile)
    loss = 0
    for cur_word, next_word in zip(x_list, x_list[1:]):
        h, new_loss = forward_one_step(h, cur_word, next_word, volatile=volatile)
        loss += new_loss
    return loss
```

We implemented the one-step-forward computation as a separate function, which is a best practice of writing recurrent nets for higher extensibility. Ignore the argument `volatile` for now, we will review it in the next subsection. The `forward` function is very simple and no special care needs to be taken with respect to the length of the input sequence. This code actually handles variable length input sequences without any tricks.

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:

```
optimizer.zero_grads()
loss = forward(x_list)
loss.backward()
optimizer.update()
```

Do not forget to call *Optimizer.zero_grads()* before the backward computation!

1.2.2 Truncate the Graph by Unchaining

Learning from very long sequences is also a typical use case of recurrent nets. Suppose that 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 that 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 `forward_one_step` function that we wrote above.

```
h = Variable(np.zeros((50,), dtype=np.float32))
loss   = 0
count  = 0
seqlen = len(x_list[1:])

for cur_word, next_word in zip(x_list, x_list[1:]):
    h, new_loss = forward_one_step(h, cur_word, next_word)
    loss  += new_loss
    count += 1
    if count % 30 == 0 or count == seqlen:
        optimizer.zero_grads()
        loss.backward()
        loss.unchain_backward()
        optimizer.update()
```

State is updated at `foward_one_step`, 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 latest state `h` itself is not lost, since above code 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.

1.2.3 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*.

> **Warning:** It is not allowed to mix volatile and non-volatile variables as arguments to same function.

Remember that our `forward` function accepts `volatile` argument. So we can enable volatile forward computation by just passing `volatile=True` to this function:

```
loss = forward(x_list, volatile=True)
```

Volatile variables are also useful to evaluate feed-forward networks.

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 that we want to train a feed-forward network `predictor_func`, which is located on top of another fixed pretrained 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=True)
feat = fixed_func(x)
feat.volatile = False
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.

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.

1.3 Using GPU(s) in Chainer

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

- Relationship between Chainer and PyCUDA
- Basics of GPUArray
- 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

1.3.1 Relationship between Chainer and PyCUDA

Chainer uses PyCUDA as its backend for GPU computation and the `pycuda.gpuarray.GPUArray` class as the GPU array implementation. GPUArray has far less features compared to `numpy.ndarray`, though it is still enough to implement the required features for Chainer.

Note: *chainer.cuda* module imports many important symbols from PyCUDA. For example, the GPUArray class is referred as `cuda.GPUArray` in the Chainer code.

Chainer provides wrappers of many PyCUDA functions and classes, mainly in order to support customized default allocation mechanism. 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. `cuMemAlloc` and `cuMemFree` functions) synchronize CPU and GPU computations, which hurts performance. In order to avoid memory allocation and deallocation during the computation, Chainer uses PyCUDA's memory pool utilities as the standard memory allocator. Since memory pool is not the default allocator in PyCUDA, Chainer provides many wrapper functions and classes to use memory pools in a simple way. At the same time, Chainer's wrapper functions and classes make it easy to handle multiple GPUs.

Note: Chainer also uses scikits.cuda for a wrapper of CUBLAS, and some functions use CuDNN v2 if available. We omit their usage in this tutorial.

Note: We also do not touch the detail of PyCUDA. See PyCUDA's documentation instead.

1.3.2 Basics of GPUArray in Chainer

In order to use GPU in Chainer, we must initialize *chainer.cuda* module before any GPU-related operations:

```
cuda.init()
```

The *cuda.init()* function initializes global state and PyCUDA. This function accepts an optional argument `device`, which indicates the GPU device ID to select initially.

Warning: If you are using `multiprocessing`, the initialization must take place for each process *after* the fork. The main process is no exception, i.e., *cuda.init()* should not be called before all the children that use GPU have been forked.

Then we can create a GPUArray object using functions of the *cuda* module. Chainer provides many constructor functions resembling the ones of NumPy: *empty()*, *empty_like()*, *full()*, *full_like()*, *zeros()*, *zeros_like()*, *ones()*, *ones_like()*.

Another useful function to create a GPUArray object is *to_gpu()*. This function copies a `numpy.ndarray` object to a newly allocated GPUArray object. For example, the following code

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

generates the same `x_gpu` as the following code:

```
x_gpu = cuda.ones((5, 4, 3))
```

Note: Allocation functions of the *cuda* module use `numpy.float32` as the default element type.

The *cuda* module also has *to_cpu()* function to copy a GPUArray object to an ndarray object:

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

All GPUArray constructors allocate memory on the current device. In order to allocate memory on a different device, we can use device switching utilities. *cuda.use_device()* function changes the current device:

```
cuda.use_device(1)
x_gpu1 = cuda.empty((4, 3))
```

There are many situations in which we want to temporarily switch the device, where the *cuda.using_device()* function is useful. It returns an resource object that can be combinated with the `with` statement:

```
with cuda.using_device(1):
    x_gpu1 = cuda.empty((4, 3))
```

These device switching utilities also accepts a GPUArray object as a device specifier. In this case, Chainer switches the current device to one that the array is allocated on:

```
with cuda.using_device(x_gpu1):
    y_gpu1 = x_gpu1 + 1
```

Warning: An array that is not allocated by Chainer's allocator cannot be used as a device specifier.

A GPUArray object allocated by Chainer can be copied between GPUs by *cuda.copy()* function:

```
cuda.use_device(0)
x0 = cuda.ones((4, 3))
x1 = cuda.copy(x0, out_device=1)
```

1.3.3 Run Neural Networks on a Single GPU

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

A *FunctionSet* object can be transferred to the specified GPU using the *to_gpu()* method. Make sure to give parameters and gradients of the GPU version to the optimizer.

```
model = FunctionSet(
    l1 = F.Linear(784, 100),
    l2 = F.Linear(100, 100),
    l3 = F.Linear(100,  10),
).to_gpu()

optimizer = optimizers.SGD()
optimizer.setup(model.collect_parameters())
```

Note that this method returns the function set itself. The device specifier can be omitted, in which case it uses the current device.

Then, all we have to do is transferring each minibatch to the GPU:

```
batchsize = 100
for epoch in xrange(20):
    print 'epoch', epoch
    indexes = np.random.permutation(60000)
    for i in xrange(0, 60000, batchsize):
        x_batch = cuda.to_gpu(x_train[indexes[i : i + batchsize]])
        y_batch = cuda.to_gpu(y_train[indexes[i : i + batchsize]])

        optimizer.zero_grads()
        loss, accuracy = forward(x_batch, y_batch)
        loss.backward()
        optimizer.update()
```

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.

1.3.4 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 first have to define a *FunctionSet*. Be careful that parameters that will be used on a device must reside on that device. Here is a simple example of the model definition:

```
model = FunctionSet(
    gpu0 = FunctionSet(
        l1=F.Linear( 784, 1000),
        l2=F.Linear(1000, 1000),
        l3=F.Linear(1000, 2000),
        l4=F.Linear(2000, 1000),
        l5=F.Linear(1000, 1000),
        l6=F.Linear(1000,   10)
    ).to_gpu(0),
    gpu1 = FunctionSet(
        l1=F.Linear( 784, 1000),
        l2=F.Linear(1000, 1000),
        l3=F.Linear(1000, 2000),
        l4=F.Linear(2000, 1000),
        l5=F.Linear(1000, 1000),
        l6=F.Linear(1000,   10)
    ).to_gpu(1)
)
```

Recall that *FunctionSet.to_gpu()* returns the FunctionSet object itself. Note that FunctionSet can be nested as above.

Now we can define the network architecture that we have shown in the diagram:

```
def forward(x_data, y_data):
    x_0 = Variable(cuda.to_gpu(x_data, 0))
    x_1 = Variable(cuda.to_gpu(x_data, 1))
    t   = Variable(cuda.to_gpu(y_data, 0))

    h1_0 = F.relu(model.gpu0.l1(x_0))
    h1_1 = F.relu(model.gpu1.l1(x_1))

    h2_0 = F.relu(model.gpu0.l2(h1_0))
    h2_1 = F.relu(model.gpu1.l2(h1_1))

    h3_0 = F.relu(model.gpu0.l3(h2_0))
    h3_1 = F.relu(model.gpu1.l3(h2_1))

    # Synchronize
    h3_0 += F.copy(h3_1, 0)
    h3_1  = F.copy(h3_0, 1)
```

```
    h4_0 = F.relu(model.gpu0.l4(h3_0))
    h4_1 = F.relu(model.gpu1.l4(h3_1))

    h5_0 = F.relu(model.gpu0.l5(h4_0))
    h5_1 = F.relu(model.gpu1.l5(h4_1))

    h6_0 = F.relu(model.gpu0.l6(h5_0))
    h6_1 = F.relu(model.gpu1.l6(h5_1))

    # Synchronize
    y = h6_0 + F.copy(h6_1, 0)
    return F.softmax_cross_entropy(y, t), F.accuracy(y, t)
```

First, recall that *cuda.to_gpu()* accepts an optional argument to specify the device identifier. We use this to transfer the input minibatch to both the 0th and the 1st devices. Then, we can write this model-parallel example employing the *functions.copy()* function. This function transfers an input array to another device. Since it is a function on *Variable*, the operation supports backprop, which 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 most of the GPU computation is asynchronous to the host CPU.

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

1.3.5 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 the model:

```
model = FunctionSet(
    l1 = F.Linear(784, 100),
    l2 = F.Linear(100, 100),
    l3 = F.Linear(100,  10),
)
```

We have to copy this model into two different devices. This is done by using `copy.deepcopy()` and *FunctionSet.to_gpu()* method:

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

Then, set up optimizer as:

```
optimizer = optimizers.SGD()
optimizer.setup(model_0.collect_parameters())
```

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`.

Forward function is almost same as the original example:

```
def forward(x_data, y_data, model):
    x = Variable(x_data)
    t = Variable(y_data)
    h1 = F.relu(model.l1(x))
    h2 = F.relu(model.l2(h1))
    y = model.l3(h2)
    return F.softmax_cross_entropy(y, t), F.accuracy(y, t)
```

The only difference is that `forward` accepts `model` as an argument. We can feed it with a model and arrays on an appropriate device. Then, we can write a data-parallel learning loop as follows:

```
batchsize = 100
for epoch in xrange(20):
    print 'epoch', epoch
    indexes = np.random.permutation(60000)
    for i in xrange(0, 60000, batchsize):
        x_batch = x_train[indexes[i : i + batchsize]]
        y_batch = y_train[indexes[i : i + batchsize]]

        optimizer.zero_grads()

        loss_0, accuracy_0 = forward(
            cuda.to_gpu(x_batch[:batchsize/2], 0),
            cuda.to_gpu(y_batch[:batchsize/2], 0),
            model_0)
        loss_0.backward()

        loss_1, accuracy_1 = forward(
            cuda.to_gpu(x_batch[batchsize/2:], 1),
            cuda.to_gpu(y_batch[batchsize/2:], 1),
            model_1)
        loss_1.backward()

        optimizer.acumulate_grads(model_1.gradients)
        optimizer.update()

        model_1.copy_parameters_from(model_0.parameters)
```

One half of the minibatch is forwarded to GPU 0, the other half to GPU 1. Then the gradients are accumulated by the *`Optimizer.accumulate_grads()`* method. 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 *`FunctionSet.copy_parameters_from()`* 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.

1.4 Define your own function

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

- How to define a non-parameterized function
- Useful tools to write a function using a GPU

- How to define a parameterized function
- How to test the function definition

After reading this section, you will be able to:

- Write your own non-parameterized function
- Define simple kernels in the function definition
- Write your own parameterized function

1.4.1 Non-parameterized 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.

Since they are simpler, we first show how to define non-parameterized 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 non-parameterized 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[0]

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

As per the warning above, `forward_cpu` function 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.).

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[0]

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

In GPU methods, arrays are of type `pycuda.gpuarray.GPUArray` We use arithmetic operators defined for GPUArray. These operators implement the basic elementwise arithmetics.

You maybe 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[0]

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

Note that this is a very rare case, since GPUArray does not implement most features of `numpy.ndarray`.

1.4.2 Write an Elementwise Kernel Function

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, which may hurt performance. We can reduce the number of invocations by defining our own kernel.

Most functions only require elementwise operations like MulAdd. PyCUDA provides a useful tool to define elementwise kernels, the `pycuda.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):
        x, y, z = inputs
        w = cuda.empty_like(x)
        cuda.elementwise(
            'float* w, const float* x, const float* y, const float* z',
            'w[i] = x[i] * y[i] + z[i]',
            'muladd_fwd')(w, x, y, z)
        return w,

    def backward_gpu(self, inputs, grad_outputs):
        x, y, z = inputs
        gw      = grad_outputs[0]

        gx = cuda.empty_like(x)
        gy = cuda.empty_like(y)
        cuda.elementwise(
            '''
               float* gx, float* gy,
               const float* x, const float* y, const float* gw
            ''', '''
               gx[i] = gy[i] * gw[i];
               gy[i] = gx[i] * gw[i];
            ''', 'muladd_bwd')(gx, gy, x, y, gw)

        gz = gw  # no copy
        return gx, gy, gz
```

`cuda.elementwise()` function accepts the essential implentation of the kernel function, and returns a kernel invokation function (actually, it returns `ElementwiseKernel` object, which is callable). In typical usage, we pass three arguments to this function. The first is an argument list of the kernel function. The second is a body of *parallel loop*, where the variable `i` indicates the index in the loop. Note that `i` runs through all indexes of the first array argument by default. The third is the name of the kernel function, which is shown in debugger 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 `/tmp` 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 PyCUDA.

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 curent context, and if it is called with

the same arguments and the same context, it reuses the previously uploaded kernel code.

1.4.3 Parameterized Functions

Next we show how to define a parameterized function. At this time, suppose that we want to implement elementwise product function between the input array and the parameter array.

Note: Note that the elementwise product between a variable and parameters can be simply implemented by *functions.Parameter* function:

```
p = F.Parameter(np.random.rand((4, 3), dtype=np.float32))
x = Variable(...)
y = p() * x
```

The Parameter function takes no arguments and just returns a variable holding the parameter array. The example in this subsection may be slightly more efficient with respect to memory consumption, though.

There are two differences between parameterized functions and non-parameterized functions:

- Parameterized functions have parameter arrays and corresponding gradient arrays. They are typically stored as attributes of the function class, where the function should provide *parameter_names* and *gradient_names* attributes (or properties). Otherwise, the function must override *parameters* and *gradients* properties directly.
- Parameterized functions must accumulate gradients on backward.

Note that gradient arrays are automatically zeroed by an optimizer, so function implementation only need to initialize their shapes. Then, the implementation of elementwise product may be as following:

```
class EltwiseParamProduct(Function):
    parameter_names = 'w',
    gradient_names  = 'gw',

    def __init__(self, shape):
        self.w  = np.random.randn(shape).astype(np.float32)
        self.gw = np.empty_like(self.w)

    def forward(self, inputs):
        x = inputs[0]
        y = self.w * x
        return y,

    def backward(self, inputs, grad_outputs):
        x  = inputs[0]
        gy = grad_outputs[0]

        self.gw += gy * x
        gx       = gy * self.w

        return gx,
```

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. It does not make any trouble even if the function object is used more than once in the same network, since *Function.__call__()* operator copies itself before the forward computation.

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.

1.4.4 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:

```
class TestReLU(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()

        func = y.creator
        f = lambda: func.forward((x.data,))
        gx, = gradient_check.numerical_grad(f, (x.data,), (y.grad,))

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

We used *Variable.creator* to extract creator function object of a variable. The first four lines of the test code are simple forward and backward computation of ReLU function. The next three lines compute numerical gradient using the same forward function without backward routine. And at last, we compare these two results elementwise. Note that 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/function_tests` directory.

Chainer Reference Manual

2.1 Core functionalities

2.1.1 Variable

class chainer.**Variable**(*data*, *volatile=False*)
Array with a structure to keep track of computation

Every variable holds a data array of type either ndarray or GPUArray.

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.

data
Data array of type either ndarray or GPUArray.

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
Boolean flag. If True, the variable does not keep track of any function applications.

__len__()
Returns the number of elements of the data array.

Returns the number of elements of the data array.

Return type int

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 complement 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.

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.

unchain_backward()

Deletes backward references of variables and functions backward, a.k.a. *backward unchaining*.

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.

2.1.2 Function

class chainer.**Function**

Function of variable(s) to variable(s) that leaves footprint to the output variables on application.

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, the function is copied, and 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: Strictly speaking, when a function is applied to some variable, a special *Function* object called *splitter* is inserted between the variable and the function. The splitter is used to manipulate multiple function applications on the same variable, where gradients from different backward paths are accumulated at the variable.

Note: *__call__()* copies the function instance before the forward computation and chaining. This enables us to reuse one function object for multiple function applications, where the different calls must use different references to the function object. Note that the copy is shallow, so implementations of *Function* must take care of any member attributes shared accross forward and backward computations.

Example

Let x an instance of *Variable* and f an instance of *Function* taking only one argument. Then a line

```
>>> y = f(x)
```

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

```
x <--- (splitter) <--- x' <--- f' <--- y
```

where prime "'" indicates a copy of the original object. If another application the function occurs as

```
>>> z = f(x)
```

then the splitter acts like a branch as the following new diagram:

```
                      |--- x'  <--- f'  <--- y
x <--- (splitter) <-+
                      |--- x'' <--- f'' <--- z
```

Note that the splitter is implicitly inserted and user does not need to take any special care of it; just remember that such branching is correctly managed by chainer.

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.

Function implementations are classified into two types: parameterized ones and non-parameterized ones. A parameterized function holds parameter arrays and coresponding gradient arrays. Implementation can choose any way to keep these arrays, but it is recommended to keep them as attributes to easily migrate between CPU and GPU. Parameterized function must provide accessors to these arrays called *parameters()* and *gradients()*.

inputs
A tuple or list of input variables.

outputs
A tuple or list of output variables.

parameter_names
A tuple or list of names of parameter attributes. It is set to an empty tuple by default. This attribute is used by the default implementation of *parameters()* property to gather the collection of parameter arrays. Implementation of parameterized function should override this field as an attribute or a property, or otherwise it should override *parameters()* property.

gradient_names
A tuple or list of names of gradient attributes. The detail is same as *parameter_names*.

__call__(**inputs*)
Applies forward propagation on input variables with chaining backward reference.

Basic behavior is also expressed in documentation of *Function* class. This function first copies itself to avoid conflict over multiple invokations.

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

Parameters **inputs** – Tuple of input *Variable* objects. All input variables must have same volatile flag.

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

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

Returns Tuple of input gradient 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 GPUArray object(s).
- **grad_outputs** – Tuple of output gradient GPUArray object(s).

Returns Tuple of input gradient GPUArray 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.

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 ndarray object(s).

Returns Tuple of 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 GPUArray object(s).

Returns Tuple of GPUArray 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.

gradients
A tuple of gradient arrays.

Default implementation collects gradient arrays based on *gradient_names* attribute.

parameters
A tuple of parameter arrays.

Default implementation collects parameter arrays based on *parameter_names* attribute.

to_cpu()
Migrates the function to CPU and returns self.

The default implementation moves all fields of type pycuda.gpuarray.GPUArray onto CPU.

Returns self.

to_gpu(*device=None*)
Migrates the function to GPU and returns self.

The default implementation moves all fields of type ndarray onto GPU.

Parameters **device** (int or pycuda.driver.Device or None) – Device ID of GPU that the function will be migrated on. If this is None, the current device is used.

Returns self.

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

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

2.1.3 FunctionSet

class chainer.**FunctionSet**(***functions*)
Set of objects with parameters and gradients properties.

FunctionSet is useful to collect parameters and gradients of multiple parameterized *Function* objects. *FunctionSet* itself also implements *parameters* and *gradients*, so it can be nested in another *FunctionSet* object.

Function registration is done by just adding an attribute to *FunctionSet* object.

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 *gradients()* property.

to_cpu()
Migrates all parameters and gradients onto CPU.

This method calls to_cpu method of each registered object.

Returns self

to_gpu(*device=None*)
Migrates all parameters and gradients onto GPU.

This method calls to_gpu method of each registered object.

Parameters **device** (int or pycuda.driver.Device or None) – Device ID of GPU. If None is given, it uses the current device.

Returns self

2.1.4 Optimizer

class chainer.**Optimizer**
Base class of all numerical optimizers.

Optimizer is set up with references to parameters and gradients, and then on every call of *update()*, it updates parameters based on corresponding gradients. Optimizer implementations must override *update_one()* method, which updates one parameter array using the corresponding gradient array.

Optimizer can optionally use state for each parameter/gradient pair. It is initialized by *init_state()* method at set up.

t
int

Number of update steps. It can be used in *update_one()* implementation, where *t* is incremented beforehand.

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.

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

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

See also:

compute_grads_norm() It uses this method to compute the gradient norm to be clipped.

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.

init_state(*param*, *grad*)
Returns initial state corresponding to the given parameter and gradient.

Default implementation delegates the procedure to *init_state_cpu()* or *init_state_gpu()* depending on the type of `param`.

Parameters

- **param** – Parameter array.
- **grad** – Gradient array corresponding to `param`.

Returns

Initial state value.

Warning: Note that, on every call of *update_one()*, the state value is passed by value and then the method updates its content, so the state must be a reference. Especiallly, one cannot use a value of built-in numeric type. If the state is one scalar value, it is recommended to use scalar array, i.e. `ndarray` with shape `()`.

init_state_cpu(*param*, *grad*)
Returns initial state corresponding to the given CPU parameter and gradient.

Parameters

- **param** (*ndarray*) – Parameter array.
- **grad** (*ndarray*) – Gradient array.

Returns Initial state value.

See also:

init_state(), *init_state_gpu()*

init_state_gpu(*param*, *grad*)
Returns initial state corresponding to the given GPU parameter and gradient.

Parameters

- **param** (*GPUArray*) – Parameter array.
- **grad** (*GPUArray*) – Gradient array.

Returns Initial state value.

See also:

init_state(), *init_state_gpu()*

setup(*params_grads*)
Prepares parameter/gradient/state tuples for all given parameter/gradient pairs.

Parameters **params_grads** – Tuple (pair) of two tuples. The first element is a tuple of parameter arrays, and the second is a tuple of corresponding gradient arrays. Return value of *FunctionSet.collect_parameters()* method can be used.

update()
Updates all parameters and states using corresponding gradients.

This method iteratively calls *update_one()* for each parameter/ gradient/state tuple. Beforehand, *t* attribute is incremented.

update_one(*param*, *grad*, *state*)
Updates one parameter array and its state using the corresponding gradient array.

The default implementation delegates the procedure to *update_one_cpu()* or *update_one_gpu()* depending on the type of the parameter array. Optimizer implmentation must override these type-specific methods or this *update_one()* method directly.

Parameters

- **param** – Parameter array.
- **grad** – Gradient array.
- **state** – State value.

See also:

update_one_cpu(), *update_one_gpu()*

update_one_cpu(*param*, *grad*, *state*)
Updates one parameter array and its state using the corresponding gradient array on CPU.

Parameters

- **param** (*ndarray*) – Parameter array.
- **grad** (*ndarray*) – Gradient array.
- **state** – State value.

See also:

update_one(), *update_one_gpu()*

update_one_gpu(*param*, *grad*, *state*)
Updates one parameter array and its state using the corresponding gradient array on GPU.

Parameters

- **param** (*GPUArray*) – Parameter array.
- **grad** (*GPUArray*) – Gradient array.
- **state** – State value.

See also:

update_one(), *update_one_cpu()*

weight_decay(*decay*)
Applies weight decay (a.k.a. L2 or Tikonov regularization) of given scale to the current gradients.

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

zero_grads()
Fills all gradient arrays by zeros.

This method should be call before backprop takes place, since gradients are accumulated on backprop.

2.2 Utilities

2.2.1 CUDA utilities

Device, context and memory management on PyCUDA and scikits.cuda.

Chainer uses PyCUDA facilities (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.cublas`	`scikits.cuda.cublas`
`chainer.cuda.cumath`	`pycuda.cumath`
`chainer.cuda.curandom`	`pycuda.curandom`
`chainer.cuda.culinalg`	`scikits.cuda.linalg`
`chainer.cuda.cumisc`	`scikits.cuda.misc`
`chainer.cuda.gpuarray`	`pycuda.gpuarray`
`chainer.cuda.Context`	`pycuda.driver.Context`
`chainer.cuda.Device`	`pycuda.driver.Device`
`chainer.cuda.Event`	`pycuda.driver.Event`
`chainer.cuda.GPUArray`	`pycuda.gpuarray.GPUArray`
`chainer.cuda.Stream`	`pycuda.driver.Stream`

Chainer provides thin wrappers of GPUArray allocation routines, which use *mem_alloc()* as the allocator. This allocator uses device-wise instance of `DeviceMemoryPool`, which enables the reuse of device memory over multiple forward/backward computations. *mem_alloc()* also inserts an additional attribute to the allocated memory called `device`, which indicates the device that the memory is allocated on. Functions of `cuda` uses this attribute to select appropriate device on each manipulation routine.

Initialization and global states

`chainer.cuda.`**init**(*device=None*)
Initializes CUDA global state.

Chainer maintains CUDA context, CUBLAS context, random number generator and device memory pool for each GPU device and for each process (the main process or a process forked by `multiprocessing`) as global states. When called for the first time on the process, this function initializes these global states.

Warning: This function also initializes PyCUDA and scikits.cuda. Since these packages do not support forking after initialization, do not call this function before forking the process.

This function also registers *`shutdown()`* to `atexit` slot.

It also initializes random number generator. User can set fixed seed with `CHAINER_SEED` environment variable.

Parameters **`device`** (`int` or `Device` or `None`) – Device ID to initialize on.

`chainer.cuda.`**`shutdown`**`()`
Finalizes CUDA global state.

This function is automatically called by `atexit`. Multiple calls are allowed, so user can manually call this function if necessary.

`chainer.cuda.`**`mem_alloc`**`(`*nbytes*`)`
Allocates device memory of given size from memory pool.

This function chooses memory pool corresponding to the current device.

Parameters **`nbytes`** (*int*) – The size of memory in bytes.

Returns Allocated memory with additional `device` attribute. This attribute is used to determine on which GPU the memory resides.

Return type pycuda.tools.PooledDeviceAllocation

Devices and contexts

`chainer.cuda.`**`get_device`**`(`*arg=None*`)`
Gets the device from ID ''arg'' or given chainer's `GPUArray`.

Parameters **`arg`** – Value to specify a GPU device.

Returns

Device object specified by given `arg`.

The rule of device selection is following.

Type of `arg`	Return value
`None`	Current device
`int`	Device of ID `arg`
`Device`	`arg`
`GPUArray`	Device given array was allocated on
`ndarray`	`None`

`chainer.cuda.`**`use_device`**`(`*arg*, *pop=True*`)`
Switches the CUDA context to use given device.

Parameters

- **`arg`** – Argument of *`get_device()`*.
- **`pop`** (*bool*) – If True, pop the current context from context stack.

chainer.cuda.**using_device**(**args*)
Returns *DeviceUser* object of the first GPUArray argument.

If none of the arguments specifies a GPU device, then it returns a dummy *DeviceUser* object which is inactive.

Parameters ***args** – Objects based on which an appropriate device should be selected.

Returns Device user instance of selected argument.

Return type *DeviceUser*

Example

Suppose arrays is a list of arrays of type either ndarray or GPUArray. Then, the following code invokes do_something_on with an appropriate context:

```
with using_device(*arrays):
    do_something_on(arrays)
```

---

class chainer.cuda.**DeviceUser**(*arg*)
RAII-style CUDA context swithcer.

Parameters **arg** – Argument of *get_device()*.

device
~pycuda.driver.Device

Selected device.

chainer.cuda.**get_context**(*arg=None*)
Gets the context corresponding to the specified device.

Parameters **arg** – Argument of *get_device()*.

Returns Context object corresponding to the specified device.

Return type Context

chainer.cuda.**get_cublas_handle**()
Gets CUBLAS handle for the current device.

Returns CUBLAS handle.

chainer.cuda.**using_cumisc**(*handle=None*)
Temporarily use chainer's CUBLAS handle on scikits.cuda.misc.

The usage is similar to *using_device()*.

Parameters **handle** – CUBLAS handle. If None is specified, it uses CUBLAS handle for the current device.

Returns Misc user object.

Return type *CumiscUser*

class chainer.cuda.**CumiscUser**(*handle*)
RAII-style switcher of scikits.cuda.misc default CUBLAS handle.

GPUArray allocation and copy

chainer.cuda.**copy**(*array*, *out=None*, *out_device=None*)
Copies GPUArray using default stream.

This function can copy the device array to the destination array on another device.

Parameters

- **array** (*GPUArray*) – Array to be copied.
- **out** (*GPUArray*) – 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()*.

Returns

Copied array.

If out is not specified, then the array is allocated on the device specified by out_device argument.

Return type GPUArray

chainer.cuda.**copy_async**(*array*, *out=None*, *out_device=None*, *stream=None*)
Copies GPUArray using given stream.

This function can copy the device array to the destination array on another device.

Parameters

- **array** (*GPUArray*) – Array to be copied.
- **out** (*GPUArray*) – 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** (*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 GPUArray

Warning: Currently, copy_async over different devices raises exception, since PyCUDA drops the definition of pycuda.driver.memcopy_peer_async().

chainer.cuda.**empty**(*shape*, *dtype=<type 'numpy.float32'>*)
Creates an uninitialized GPUArray.

Parameters

- **shape** (*tuple of ints*) – The shape of array.
- **dtype** (*numpy.dtype*) – Element type.

Returns Uninitialized GPU array allocated by memory pool.

Return type GPUArray

chainer.cuda.**empty_like**(*array*)
Alias to pycuda.gpuarray.empty_like().

chainer.cuda.**full**(*shape*, *fill_value*, *dtype=<type 'numpy.float32'>*, *stream=None*)
Creates a constant-filled GPUArray.

Parameters

- **shape** (*tuple of ints*) – The shape of array.
- **fill_value** – Constant to fill the array by.
- **dtype** (*numpy.dtype*) – Element type.
- **stream** (*Stream*) – CUDA stream.

Returns Constant-filled GPU array allocated by memory pool.

Return type GPUArray

chainer.cuda.**full_like**(*array*, *fill_value*, *stream=None*)
Creates a constant-filled GPUArray like given array.

Parameters

- **array** (*GPUArray*) – Base array.
- **fill_value** – Constant value to fill the array by.
- **stream** (*Stream*) – CUDA stream.

Returns Constant-filled array.

Return type GPUArray

chainer.cuda.**zeros**(*shape*, *dtype=<type 'numpy.float32'>*, *stream=None*)
Creates a zero-filled GPUArray.

This function is equivalent to `full(shape, 0, dtype, stream)`.

chainer.cuda.**zeros_like**(*array*, *stream=None*)
Creates a zero-filled GPUArray like given array.

This function is equivalent to `full_like(array, 0, stream)`.

chainer.cuda.**ones**(*shape*, *dtype=<type 'numpy.float32'>*, *stream=None*)
Creates a zero-filled GPUArray.

This function is equivalent to `full(shape, 1, dtype, stream)`.

chainer.cuda.**ones_like**(*array*, *stream=None*)
Creates a one-filled GPUArray like given array.

This function is equivalent to `full_like(array, 1, stream)`.

chainer.cuda.**to_cpu**(*array*)
Copies the given GPU array to host CPU.

Parameters **array** – Array to be sent to GPU.

Returns

Array on CPU.

If given `array` is already on CPU, then this function just returns `array` without performing any copy.

Return type ndarray

chainer.cuda.**to_cpu_async**(*array*, *stream=None*)
Copies the given GPU array asynchronously to host CPU.

Parameters

- **array** – Array to be sent to GPU.
- **stream** (*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 ndarray

`chainer.cuda.`**`to_gpu`**(*array*, *device=None*)

Copies the given CPU array to specified device.

Parameters

- **array** – Array to be sent to GPU.
- **device** – Device specifier.

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 GPUArray into specified device.

Return type GPUArray

`chainer.cuda.`**`to_gpu_async`**(*array*, *stream=None*)

Copies the given CPU array asynchronously to the current device.

Parameters

- **array** – Array to be sent to GPU. If it is `ndarray`, then its memory must be pagelocked.
- **stream** (*Stream*) – CUDA stream.

Returns

Array on GPU.

If given `array` is already on GPU, then this function just returns `array` without performing any copy.

Return type GPUArray

Random number generators

`chainer.cuda.`**`get_generator`**(*device=None*)

Gets the random number generator for the given device.

Parameters **device** – Device specifier (an arugment of *`get_device()`*)

Returns Random number generator.

Return type pycuda.curandom.XORWOWRandomNumberGenerator

`chainer.cuda.`**`seed`**(*s=None*, *device=None*)

Resets the random number generator of the specified device by the given seed.

Parameters

- **s** (*int or None*) – Seed value. If it is None, it initializes the generator without fixed seed.
- **device** – Device specifier (i.e. argument of *get_device()*).

Kernel definition utilities

chainer.cuda.**elementwise**(*arguments*, *operation*, *name*, *keep=False*, *options=None*, *preamble=''*, *loop_prep=''*, *after_loop=''*)

Creates an elementwise kernel function.

This function uses pycuda.tools.context_dependent_memoize() to cache the resulting kernel object, i.e. the resulting kernel object is cached for each arguments and CUDA context.

The arguments are the same as those for pycuda.elementwise.ElementwiseKernel(), except that name argument is mandatory.

chainer.cuda.**reduce**(*arguments*, *map_expr*, *reduce_expr*, *neutral*, *name*, *dtype_out=<type 'numpy.float32'>*, *keep=False*, *options=None*, *preamble=''*)

Creates a global reduction kernel function.

This function uses pycuda.tools.context_dependent_memoize() to cache the resulting kernel object, i.e. the resulting kernel object is cached for each argument and CUDA context.

The arguments are the same as those for pycuda.reduction.ReductionKernel(), except that their order is different and name argument is mandatory.

Interprocess communication on GPU

class chainer.cuda.**IPCEvent**

Event object for interprocess synchronization on GPU.

class chainer.cuda.**IPCArrayHandle**(*array*)

Converter between GPUArray and its Inter-Process Communication handle.

It holds IPC memory handle with shape and dtype information. The instance can be pickled, which means it can be passed through IPC path way, e.g. Pipe and Queue. The other process can extract shared GPUArray by calling get(). Also, the extracted array can be re-converted into another IPCArrayHandle.

2.2.2 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.**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

2.3 Standard Function implementations

Chainer provides basic *Function* implementations in the *chainer.functions* package.

Non-parameterized functions are provided as plain Python functions. These can be used directly in forward computation without explicit handling of *Function* objects. On the other hand, parameterized functions should be used with explicit handling of *Function* objects.

2.3.1 Learnable connections

class chainer.functions.**BinaryHierarchicalSoftmax**(*in_size*, *tree*)
Implementation of hierarchical softmax (HSM).

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:

$$\begin{aligned} P(x) &= \prod_{(e_i, b_i) \in \mathrm{path}(x)} P(b_i | e_i) \\ &= \prod_{(e_i, b_i) \in \mathrm{path}(x)} \sigma(b_i x^\top w_{e_i}), \end{aligned}$$

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)*.

See: Hierarchical Probabilistic Neural Network Language Model [Morin+, AISTAT2005].

class chainer.functions.**Convolution2D**(*in_channels*, *out_channels*, *ksize*, *stride=1*, *pad=0*, *wscale=1*, *bias=0*, *nobias=False*, *use_cudnn=True*)

Two-dimensional convolution function.

The details of this function are described below the arguments description.

Parameters

- **in_channels** (*int*) – Number of channels of input arrays.
- **out_channels** (*int*) – Number of channels of output arrays.
- **ksize** (*int or (int, int)*) – Size of filters (a.k.a. kernels). `ksize=k` and `ksize=(k, k)` are equivalent.
- **stride** (*int or (int, int)*) – Stride of filter applications. `stride=s` and `stride=(s, s)` are equivalent.
- **pad** (*int or (int, int)*) – 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.
- **use_cudnn** (*bool*) – If True, then this function uses CuDNN if available.

This function holds at most two parameter arrays: `W` and `b`, which indicate the filter weight and the bias vector, respectively.

The filter weight has four dimensions (c_O, c_I, k_H, k_W) which indicate the number of output channels, the number of input 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. Each element of it is initialized by `bias` argument. If `nobias` argument is set to True, then this function does not hold the bias parameter.

The two-dimensional convolution function is defined as follows. Let X be the input tensor of dimensions (n, c_I, h, w), where n is the batch size, and (h, w) is spatial size of the input image. 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:

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

class chainer.functions.**EmbedID**(*in_size*, *out_size*)

Efficient linear function for one-hot input.

This is a parameterized function to embed the given discrete identifier (e.g. word) into a continuous vector space. This function just holds embedding vectors for all identifiers as one large matrix `W`, which is learnable. The identifiers are directly used as indexes of the matrix `W`.

Parameters

- **in_size** (*int*) – Number of different identifiers (a.k.a. vocabulary size).
- **out_size** (*int*) – Size of embedding vector.

Note: This function is non-differentiable with respect to the input identifiers.

---

class chainer.functions.**Linear**(*in_size*, *out_size*, *wscale=1*, *bias=0*, *nobias=False*)

Implementation of a linear function (a.k.a. fully-connected layer or affine transformation).

This function holds a weight matrix `W` and a bias vector `b`.

The weight matrix `W` has shape `(out_size, in_size)`. This matrix 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.

Note: This function accepts an input variable 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.

---

class chainer.functions.**Parameter**(*array*)

Function that outputs its weight array.

This is a parameterized function that takes no input and returns a variable holding a shallow copy of the parameter array.

Parameters **array** – Initial parameter array.

2.3.2 Array manipulation functions

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*

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*

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.

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

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*

2.3.3 Activation functions

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

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*

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

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. Then, first the incoming signal x is split along the second dimension into four arrays a, i, f, o of the same shapes. Second, it computes outputs as:

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

This function outputs these two arrays 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.

class chainer.functions.**PReLU**(*shape=()*, *init=0.25*)

Parametric ReLU function.

PReLU function is written in elementwise equation as $PReLU(x) = \max(x, ax)$, where a is a parameter array.

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.

Parameters

- **shape** (*tuple of ints*) – Shape of the parameter array.
- **init** (*float*) – Initial parameter value.

See detail in paper: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.

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*

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*

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

This function only accepts a two dimensional input array, and computes its softmax along the second axis. For each index i, j of the input matrix x, it computes $f_{ij}(x) = \frac{\exp(x_{ij})}{\sum_j \exp(x_{ij})}$.

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*

`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*

2.3.4 Pooling functions

`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 (int, int)*) – Size of pooling window. `ksize=k` and `ksize=(k, k)` are equivalent.
- **`stride`** (*int or (int, int) or None*) – Stride of pooling applications. `ksize=k` and `ksize=(k, k)` are equivalent. If None is specified, then it uses same stride as the pooling window size.
- **`pad`** (*int or (int, int)*) – 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.

`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 (int, int)*) – Size of pooling window. `ksize=k` and `ksize=(k, k)` are equivalent.
- **`stride`** (*int or (int, int) or None*) – Stride of pooling applications. `ksize=k` and `ksize=(k, k)` are equivalent. If None is specified, then it uses same stride as the pooling window size.
- **`pad`** (*int or (int, int)*) – 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 Ouptut variable.

Return type *Variable*

2.3.5 Normalization functions

class `chainer.functions.`**`BatchNormalization`**(*size*, *decay=0.9*, *eps=1e-05*)

Batch normalization on outputs of linear or convolution functions.

Parameters

- **`size`** (*int or tuple of ints*) – Size (or shape) of channel dimensions.
- **`decay`** (*float*) – Decay rate of moving average.
- **`eps`** (*float*) – Epsilon value for numerical stability.

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

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

Invokes the forward propagation of BatchNormalization.

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

Parameters

- **`x`** (Variable) – An input variable.
- **`test`** (*bool*) – If `True`, BatchNormalization runs in testing mode; it normalizes the input using precomputed statistics.

- **finetune** (*bool*) – If `True`, BatchNormalization runs in finetuning 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.

`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: SSec. 3.3 of ImageNet Classification with Deep Convolutional Neural Networks

2.3.6 Loss, evaluation and aggregation

`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 groundtruth labels.

Returns A variable holding a scalar array of the accuracy.

Return type *Variable*

Note: This function is non-differentiable.

`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.

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

Computes cross entropy loss on softmax of the prediction using the groundtruth label vector.

Parameters

- **x** (Variable) – Variable holding a matrix whose (i, j)-th element indicates unnormalized log probability of the class j at the i-th example.
- **t** (Variable) – Variable holding an int32 vector of groundtruth labels.

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

Return type *Variable*

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

`chainer.functions.sum`(*x*)
Computes sum of all elements.

2.3.7 Reusable subnetwork of complex architectures

class `chainer.functions.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.

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*

Note: This function inserts the full computation graph of the Inception module behind the input array. This function itself is not inserted into the computation graph.

2.4 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*, *lam=0.99999999*, *eps=1e-08*)
Adam optimization algorithm.

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

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

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

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

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

C

Symbols

A

B

C

D

E

F

G

I