Chainer Documentation

Release 1.3.2

Preferred Networks, inc. and Preferred Infrastructure, inc.

September 30, 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, utils
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 have 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([[ 1.01847613,  0.23103087,  0.56507462],
       [ 1.29378033,  1.07823515, -0.56423163]], 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([[ 3.1757617 ,  1.75755572],
       [ 8.61950684,  7.18090773]], dtype=float32)
```

Gradients of parameters are computed by *backward()* method. Note that gradients are **accumulated** by the method rather than overwritten. So first you must initialize gradients to zero to renew the computation. 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),
... )
>>> type(model.l1)
<class 'chainer.functions.connection.linear.Linear'>
>>> type(model.l2)
<class 'chainer.functions.connection.linear.Linear'>
```

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)
```

The method *setup()* prepares for the optimization given parameters and gradients based on passed `FunctionSet`.

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, we prepared `load_mnist_data` function at `examples/mnist/data.py`:

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

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

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

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

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
>>> datasize = 60000
>>> for epoch in range(20):
...     print('epoch %d' % epoch)
...     indexes = np.random.permutation(datasize)
...     for i in range(0, datasize, batchsize):
```

```
...         x_batch = x_train[indexes[i : i + batchsize]]
...         y_batch = y_train[indexes[i : i + batchsize]]
...
...         optimizer.zero_grads()
...         loss, accuracy = forward(x_batch, y_batch)
...         loss.backward()
...         optimizer.update()
epoch 0...
```

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 range(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)
```

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((1, 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
```

Note that the first dimension of `h` and `x_list` is always the mini-batch size. The mini-batch size is assumed to be `1` here. 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((1, 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 CuPy
- Basics of CuPy
- Single-GPU usage of Chainer
- Multi-GPU usage of model-parallel computing
- Multi-GPU usage of data-parallel computing

After reading this section, you will be able to:

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

1.3.1 Relationship between Chainer and CuPy

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

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

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

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

1.3.2 Basics of cupy.ndarray

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

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

The main difference of *cupy.ndarray* from `numpy.ndarray` is that the content is allocated on the device memory. The allocation takes place on the current device by default. The current device can be changed by *cupy.cuda.Device* object as follows:

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

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

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

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

It is equivalent to the following code using CuPy:

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

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

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

It is equivalent to the following code using CuPy:

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

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

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

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

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

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

Since it accepts NumPy arrays, we can write a function that accepts both NumPy and CuPy arrays with correct device switching:

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

The compatibility of CuPy with NumPy enables us to write CPU/GPU generic code. It can be made easy by the *chainer.cuda.get_array_module()* function. This function returns the `numpy` or *cupy* module based on arguments. A CPU/GPU generic function is defined using it like follows:

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

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)
```

Note that this method returns the *FunctionSet* 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
datasize = len(x_train)
for epoch in range(20):
    print('epoch %d' % epoch)
    indexes = np.random.permutation(datasize)
    for i in range(0, datasize, batchsize):
```

```
        x_batch = 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)
```

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
datasize = len(x_train)
for epoch in range(20):
    print('epoch %d' % epoch)
    indexes = np.random.permutation(datasize)
    for i in range(0, datasize, batchsize):
        x_batch = x_train[indexes[i : i + batchsize]]
        y_batch = y_train[indexes[i : i + batchsize]]

        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.accumulate_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

        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

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

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

You 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

Since the *`cupy.ndarray`* class implements many methods of `numpy.ndarray`, we can write these unified methods in most cases.

1.4.2 Unified forward/backward methods with NumPy/CuPy functions

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

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

    def backward_cpu(self, inputs, grad_outputs):
        x, y = inputs
        gz, = grad_outputs

        gx = gz * np.exp(x)
        gy = gz * np.exp(y)
        return gx, gy

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

    def backward_gpu(self, inputs, grad_outputs):
        cupy = cuda.cupy
        x, y = inputs
        gz, = grad_outputs

        gx = gz * cupy.exp(x)
        gy = gz * cupy.exp(y)
        return gx, gy
```

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

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

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

    def backward(self, inputs, grad_outputs):
        xp = cuda.get_array_module(*inputs)
        x, y = inputs
        gz, = grad_outputs
```

```
        gx = gz * xp.exp(x)
        gy = gz * xp.exp(y)
        return gx, gy
```

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

1.4.3 Write an Elementwise Kernel Function

Let's turn back to the MulAdd example.

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

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

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

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

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

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

        gx, gy = cuda.elementwise(
            'float32 x, float32 y, float32 gw',
            'float32 gx, float32 gy',
            '''
               gx = gy * gw;
               gy = gx * gw;
            ''',
            'muladd_bwd')(x, y, gw)

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

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

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

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

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

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

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

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

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

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

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

        gx, gy = cuda.elementwise(
            'T x, T y, T gw',
            'T gx, T gy',
            '''
               gx = gy * gw;
               gy = gx * gw;
            ''',
            'muladd_bwd')(x, y, gw)

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

The type placeholder `T` indicates an arbitrary data type that CuPy supports.

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

1.4.4 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.randn(4, 3).astype(np.float32))
x = Variable(np.random.randn(4, 3).astype(np.float32))
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
        y = self.w * x
        return y,

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

        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.

Be careful that it might increase the memory consumption during the whole forward-backward computation. If you want to train very large networks on a GPU with limited memory, it is not recommended to cache arrays between forward and backward. There is one exception for this: caching the output arrays do not change the memory consumption, because they are also held by the output Variable objects.

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

1.4.5 Testing Function

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

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

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

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

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

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

Here is a typical usage of gradient checking utilities. This is a test example of *functions.relu()* function

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

        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/chainer_tests/function_tests` directory.

1.5 Type check

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

- Basic usage of type check
- Detail of type information
- Internal mechanism of type check
- More complicated cases
- Call functions
- Typical type check example

After reading this section, you will be able to:

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

1.5.1 Basic usage of type check

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

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

`check_type_forward()` gets an argument `in_types`:

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

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

```
x_type, y_type = in_types
```

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

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

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

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

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

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

1.5.2 Detail of type information

You can access three information of `x_type`.

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

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

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

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

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

And an error is like this:

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

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

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

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

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

1.5.3 Internal mechanism of type check

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

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

If you want to evaluate a *utils.type_check.Expr* object, call `eval()` method:

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

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

1.5.4 More powerfull methods

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

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

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

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

To compare a member variable of your function, wrap a value with `utils.type_check.Variable` to show readable error message:

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

This code can check the equivalent condition below:

```
x_type.shape[0] == self.in_size
```

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

```
InvalidType: Expect: in_types[0].shape[0] == 4  # what does '4' mean?
Actual: 3 != 4
```

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

The former shows this message:

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

1.5.5 Call functions

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

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

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

The above example makes an error message like this:

```
Traceback (most recent call last):
...
InvalidType: Expect: sum(in_types[0].shape) == 10
Actual: 7 != 10
```

1.5.6 More complicated cases

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

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

```
    actual_msg = 'Shape is ...'
    raise utils.type_check.InvalidType(expect_msg, actual_msg)
```

Please make a readable error message. This code generates an error below:

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

1.5.7 Typical type check example

We show a typical type check for a function.

First check the number of arguments:

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

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

And then, get each type:

```
x_type, y_type = in_types
```

Don't get each value before check `in_types.size()`. When the number of argument is illegal, this process may fail. For example, this code doesn't work when the size of `in_types` is zero:

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

After that, check each type:

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

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

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 numpy.ndarray or *cupy.ndarray*.

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

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

data

Data array of type either numpy.ndarray or *cupy.ndarray*.

grad

Gradient array. It is None until backprop reaches this variable.

creator

The function who creates this variable. It is None if the variable is not created by any function.

volatile

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 complements 1.0 as the initial error. This is useful on starting backprop from some scalar loss value.

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

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

label
Short text that represents the function.

set_creator(*gen_func*)
Notifies the variable that the given function is its creator.

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

unchain_backward()
Deletes references between variables and functions backward.

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

2.1.2 Function

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

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

The main feature of this class is keeping track of function applications as a backward graph. When a function is applied to *Variable* objects, 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*.

type_check_enable
When it is `True`, the function checks types of input arguments. Set `CHAINER_TYPE_CHECK` environment variable `0` to disable type check, or set the variable directly in your own program.

__call__(**inputs*)
Applies forward propagation with chaining backward references.

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

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

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

Return type tuple

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

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

Parameters

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

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

Return type tuple

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

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

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

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

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

It delegates the procedure to *forward_cpu()* or *forward_gpu()* by default. Which it selects is determined by the type of input arrays. Implementations of *Function* must implement either cpu/gpu methods or this method.

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

Returns Tuple of output array(s).

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

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

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

Returns Tuple of numpy.ndarray object(s).

Return type tuple

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

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

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

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

Return type tuple

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

gradients
A tuple of gradient arrays.

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

label
Short text that represents the function.

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

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 *cupy.ndarray* onto CPU.

Returns self.

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

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

Parameters **device** (int or *cupy.cuda.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 this function itself from the 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.

__getitem__(*key*)
Returns the *Function* objects by name.

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

Returns Function object.

Return type *Function*

Example

```
>>> model = FunctionSet(l1=F.Linear(10, 10), l2=F.Linear(10, 10))
>>> l1 = model['l1']
```

collect_parameters()
Returns a tuple of parameters and gradients.

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

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

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

gradients
Tuple of gradient arrays of all registered functions.

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

parameters
Tuple of parameter arrays of all registered functions.

The order of parameters is consistent with *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 *cupy.cuda.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 the initial state for 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 a zero-dimensional array, i.e. `numpy.ndarray` with shape `()`.

init_state_cpu(*param*, *grad*)
Returns the initial state for given parameter and gradient on GPU.

Parameters

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

Returns Initial state value.

See also:

init_state(), *init_state_gpu()*

init_state_gpu(*param*, *grad*)
Returns the initial state for given parameter and gradient on CPU.

Parameters

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

Returns Initial state value.

See also:

init_state(), *init_state_gpu()*

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

Parameters **params_grads** – FunctionSet or tuple (pair) of two tuples. For tuple, the first element is a tuple of parameter arrays, and the second is a tuple of corresponding gradient arrays.

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 a parameter array and its state using given gradient.

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 a parameter array and its state using given gradient on CPU.

Parameters

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

See also:

update_one(), *update_one_gpu()*

update_one_gpu(*param*, *grad*, *state*)
Updates a parameter array and its state using given gradient on GPU.

Parameters

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

See also:

update_one(), *update_one_cpu()*

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

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 CuPy.

Chainer uses CuPy (with very thin wrapper) to exploit the speed of GPU computation. Following modules and classes are imported to `cuda` module for convenience (refer to this table when reading chainer's source codes).

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

Chainer replaces the default allocator of CuPy by its memory pool implementation. It enables us to reuse the device memory over multiple forward/backward computations, and temporary arrays for consecutive elementwise operations.

Devices

`chainer.cuda.`**`get_device`**`(`**args*`)`

Gets the device from an ID integer or an array object.

This is a convenient utility to select a correct device if the type of `arg` is unknown (i.e., one can use this function on arrays that may be on CPU or GPU). The returned device object supports the context management protocol of Python for the *with* statement.

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

Returns Device object specified by given `args`.

See also:

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

CuPy array allocation and copy

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

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

chainer.cuda.**copy**(*array*, *out=None*, *out_device=None*, *stream=None*)
Copies a cupy.ndarray object using the default stream.

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

Parameters

- **array** (cupy.ndarray) – Array to be copied.
- **out** (cupy.ndarray) – Destination array. If it is not None, then out_device argument is ignored.
- **out_device** – Destination device specifier. Actual device object is obtained by passing this value to *get_device()*.
- **stream** (cupy.cuda.Stream) – CUDA stream.

Returns

Copied array.

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

Return type *cupy.ndarray*

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

Parameters

- **array** – Array to be sent to GPU.
- **stream** (cupy.cuda.Stream) – CUDA stream.

Returns

Array on CPU.

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

Return type numpy.ndarray

chainer.cuda.**to_gpu**(*array*, *device=None*, *stream=None*)
Copies the given CPU array to specified device.

Parameters

- **array** – Array to be sent to GPU.
- **device** – Device specifier.
- **stream** (cupy.cuda.Stream) – CUDA stream.

Returns

Array on GPU.

If array is already on GPU, then this function just returns array without performing any copy. Note that this function does not copy cupy.ndarray into specified device.

Return type *cupy.ndarray*

Kernel definition utilities

chainer.cuda.**memoize**(*for_each_device=False*)
Makes a function memoizing the result for each argument and device.

This is a similar version of *cupy.memoize()*. The difference is that this function can be used in the global scope even if CUDA is not available. In such case, this function does nothing.

Note: This decorator acts as a dummy if CUDA is not available. It cannot be used for general purpose memoization even if `for_each_device` is set to False.

---

chainer.cuda.**elementwise**(**args*, ***kwargs*)
Creates an elementwise kernel function.

This function uses *memoize()* to cache the kernel object, i.e. the resulting kernel object is cached for each argument combination and CUDA device.

The arguments are the same as those for *cupy.ElementwiseKernel*, except that the `name` argument is mandatory.

chainer.cuda.**reduce**(**args*, ***kwargs*)
Creates a global reduction kernel function.

This function uses *memoize()* to cache the resulting kernel object, i.e. the resulting kernel object is cached for each argument combination and CUDA device.

The arguments are the same as those for *cupy.ReductionKernel*, except that the `name` argument is mandatory.

CPU/GPU generic code support

chainer.cuda.**get_array_module**(**args*)
Gets an appropriate one from `numpy` or *cupy*.

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

Parameters **args** – Values to determine whether NumPy or CuPy should be used.

Returns *cupy* or `numpy` is returned based on the types of the arguments.

Return type module

2.2.2 Common algorithms

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

This method generates a random sample from given probabilities $p_1, \ldots, p_n$ in $O(1)$ time. It is more efficient than `choice()`. This class works on both CPU and GPU.

Parameters **probs** (*float list*) – Probabilities of entries. They are normalized with *sum(probs)*.

See: Wikipedia article

sample(*shape*)
Generates a random sample based on given probabilities.

Parameters **shape** (*tuple of int*) – Shape of a return value.

Returns Returns a generated array with the given shape. If a sampler is in CPU mode the return value is a `numpy.ndarray` object, and if it is in GPU mode the return value is a *`cupy.ndarray`* object.

to_gpu()
Make a sampler GPU mode.

2.3 Assertion and Testing

Chainer provides some facilities to make debugging easy.

`Function` uses a systematic type checking of the *`chainer.utils.type_check`* module. It enables users to easily find bugs of forward and backward implementations. You can find examples of type checking in some function implementations.

Most function implementations are numerically tested by *gradient checking*. This method computes numerical gradients of forward routines and compares their results with the corresponding backward routines. It enables us to make the source of issues clear when we hit an error of gradient computations. The *`chainer.gradient_check`* module makes it easy to implement the gradient checking.

2.3.1 Type checking utilites

class chainer.utils.type_check.**Expr**(*priority*)
Abstract syntax tree of an expression.

It represents an abstract syntax tree, and isn't a value. You can get its actual value with *`eval()`* function, and get syntax representation with the `__str__()` method. Each comparison operator (e.g. ==) generates a new *`Expr`* object which represents the result of comparison between two expressions.

Example

Let `x` and `y` be instances of *`Expr`*, then

```
>>> c = (x == y)
```

is also an instance of *`Expr`*. To evaluate and get its value, call *`eval()`* method:

```
>>> c.eval()
True   # when x.eval() == y.eval()
```

Call `str` function to get a representation of the original equaltion:

```
>>> str(c)
'x + y'   # when str(x) == 'x' and str(y) == 'y'
```

You can actually compare an expression with a value:

```
>>> (x == 1).eval()
```

Note that you can't use boolean operators such as `and`, as they try to cast expressions to boolean values:

```
>>> x == y and y == z  # raises an error
```

---

eval()
Evaluates the tree to get actual value.

Behavior of this function depends on an implementation class. For example, a binary operator + calls the __add__ function with the two results of *eval()* funciton.

chainer.utils.type_check.**expect**(**bool_exprs*)
Evaluates and tests all given expressions.

This function evaluates given boolean expressions in order. When at least one expression is evaluated as *False*, that means the given condition is not satisfied. You can check conditions with this function.

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

class chainer.utils.type_check.**TypeInfo**(*shape*, *dtype*)
Type information of an input/gradient array.

It contains type information of an array, such as the shape of array and the number of dimensions. This information is independent of CPU or GPU array.

class chainer.utils.type_check.**TypeInfoTuple**
Type information of input/gradient tuples.

It is a sub-class of tuple containing *TypeInfo*. The i-th element of this object contains type information of the i-th input/gradinent data. As each element is *Expr*, you can easily check its validity.

size()
Returns an expression representing its length.

Returns An expression object representig length of the tuple.

Return type *Expr*

2.3.2 Gradient checking utilities

chainer.gradient_check.**assert_allclose**(*x*, *y*, *atol=1e-05*, *rtol=0.0001*, *verbose=True*)
Asserts if some corresponding element of x and y differs too much.

This function can handle both CPU and GPU arrays simultaneously.

Parameters

- **x** – Left-hand-side array.
- **y** – Right-hand-side array.
- **atol** (*float*) – Absolute tolerance.
- **rtol** (*float*) – Relative tolerance.
- **verbose** (*bool*) – If True, it outputs verbose messages on error.

chainer.gradient_check.**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

2.4 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.

2.4.1 Learnable connections

class `chainer.functions.`**Bilinear**(*left_size*, *right_size*, *out_size*, *nobias=False*, *initialW=None*, *initial_bias=None*)

Bilinear function, an extension of Linear function.

`Bilinear` function takes two input vectors and outputs one vector. If one of the input vectors is fixed, this function works as an affine transform of the other input vector.

`Bilinear` function is a building block of Neural Tensor Network (See the reference paper below).

To be precise, `Bilinear` function has four parameters, $W \in R^{J \cdot K \cdot L}$, $V^1 \in R^{J \cdot L}$, $V^2 \in R^{K \cdot L}$, and $b \in R^L$. In this document, we call V^1, V^2, and b linear parameters.

Given two inputs (in a mini-batch manner) $e^1 \in R^{I \cdot J}$ and $e^2 \in R^{I \cdot K}$ where I is mini-batch size, the output of forward propagation is calculated as

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

If `nobias` option is set `True`, `Bilinear` function does not have linear parameters, that is, the last three term is omitted and only W works as the parameter.

Note: `Bilinear` 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.

Note: In the original paper, J and K must be equal and the author denotes $[V^1 V^2]$ (concatenation of matrices) by V.

Parameters

- **left_size** (*int*) – Dimension of input vector e^1 (J)
- **right_size** (*int*) – Dimension of input vector e^2 (K)
- **out_size** (*int*) – Dimension of output vector y (L)
- **nobias** (*bool*) – If `True`, linear parameters are omitted.
- **initialW** (*3-D Array*) – Initial value of W. Shape of this argument must be `(left_size, right_size, out_size)`. If `None`, W is initialized by centered Gaussian distribution properly scaled according to the dimension of inputs and outputs.

- **initial_bias** (*tuple*) – Intial values of V^1, V^2 and b. The length this argument must be 3. Each element of this tuple must have the shapes of `(left_size, output_size)`, `(right_size, output_size)`, and `(output_size,)`, respectively. If `None`, V^1 and V^2 is initialized by scaled centered Gaussian distributions and b is set to 0.

See: Reasoning With Neural Tensor Networks for Knowledge Base Completion [Socher+, NIPS2013].

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*, *initialW=None*, *initial_bias=None*, *dtype=<type 'numpy.float32'>*)

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.
- **initialW** (*4-D array*) – Initial weight value. If `None`, then this function uses to initialize `wscale`.
- **initial_bias** (*1-D array*) – Initial bias value. If `None`, then this function uses to initialize `bias`.
- **dtype** (*numpy.dtype*) – Type to use in computing.

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*, *initialW=None*, *initial_bias=None*)

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.
- **initialW** (*2-D array*) – Initial weight value. If `None`, then this function uses to initialize `wscale`.
- **initial_bias** (*1-D array*) – Initial bias value. If `None`, then this function uses to initialize `bias`.

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.`**NegativeSampling**(*in_size*, *counts*, *sample_size*, *power=0.75*)

Implementation of negative sampling.

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

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

The objective function is below:

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

where $\sigma(\cdot)$ is a sigmoid function, w_i is the weight vector for the word i, and p is a positive example. It is approximeted with k examples N sampled from probability $P(i)$, like this:

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

Each sample of N is drawn from the word distribution $P(w)$. This is calculated as $P(w) = \frac{1}{Z} c(w)^\alpha$, where $c(w)$ is the unigram count of the word w, α is a hyper-parameter, and Z is the normalization constant.

Parameters
- **in_size** (*int*) – Dimension of input vectors.
- **counts** (*int list*) – Number of each identifiers.
- **sample_size** (*int*) – Number of negative samples.
- **power** (*float*) – Power factor α.

See: Distributed Representations of Words and Phrases and their Compositionality

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.

2.4.2 Array commputation functions

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

Two-dimensional convolution function.

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

- **W** (Variable) – Weight variable.
- **b** (Variable) – Bias variable (optional).
- **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.
- **use_cudnn** (*bool*) – If True, then this function uses CuDNN if available.

Returns Output variable.

Return type *Variable*

See also:

Convolution2D

`chainer.functions.linear`(*x*, *W*, *b=None*)

Nonparameterized linear function.

Parameters

- **x** (Variable) – Input variable.
- **W** (Variable) – Weight variable.
- **b** (Variable) – Bias variable (optional).

Returns Output variable.

Return type *Variable*

See also:

Linear

`chainer.functions.matmul`(*a*, *b*, *transa=False*, *transb=False*)

Computes the matrix multiplication of two arrays.

Parameters

- **a** (Variable) – The left operand of the matrix multiplication. A 1-D array of shape (N,) is considered as an Nx1 matrix. A 2-D array of shape (M, N) is considered as an MxN matrix.
- **b** (Variable) – The right operand of the matrix multiplication. Its array is treated as a matrix in the same way as `a`'s array.
- **transa** (*bool*) – If true, transpose a.
- **transb** (*bool*) – If true, transpose b.

Returns The result of the matrix multiplication as a 2-D array.

Return type *Variable*

`chainer.functions.batch_matmul`(*a*, *b*, *transa=False*, *transb=False*)

Computes the batch matrix multiplications of two sets of arrays.

Parameters

- **a** (Variable) – The left operand of the batch matrix multiplications. A 2-D array of shape (B, N,) is considered as B Nx1 matrices. A 3-D array of shape (B, M, N) is considered as B MxN matrices.

- **b** (Variable) – The right operand of the batch matrix multiplications. Its array is treated as matrices in the same way as `a`'s array.
- **transa** (*bool*) – If true, transpose each matrix in a.
- **transb** (*bool*) – If true, transpose each matrix in b.

Returns The result of the batch matrix multiplications as a 3-D array.

Return type *Variable*

2.4.3 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.`**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*

`chainer.functions.`**split_axis**(*x*, *indices_or_sections*, *axis*)
Splits given variables along an axis.

Parameters

- **x** (*tuple of Variables*) – Variables to be split.

- **indices_or_sections** (*int or 1-D array*) – If this argument is an integer, N, the array will be divided into N equal arrays along axis. If it is a 1-D array of sorted integers, it indicates the positions where the array is split.
- **axis** (*int*) – Axis that the input array is split along.

Returns Tuple of `Variable` objects if the number of outputs is more than 1 or `Variable` otherwise.

Return type `tuple` or `Variable`

Note: This function raises `ValueError` if at least one of the outputs is splitted to zero-size (i.e. *axis*-th value of its shape is zero).

2.4.4 Activation functions

`chainer.functions.`**`clipped_relu`**(*x*, *z=20.0*)
Clipped Rectifier Unit function.

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

Parameters

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

Returns Output variable.

Return type *Variable*

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

`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.

First, the incoming signal x is split into four arrays a, i, f, o of the same shapes along the second axis. It means that x 's second axis must have 4 times the length of c_{prev}.

The splitted input signals are corresponding to:

- a : sources of cell input
- i : sources of input gate
- f : sources of forget gate
- o : sources of output gate

Second, it computes outputs as:

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

These are returned as a tuple of two variables.

Parameters

- **c_prev** (Variable) – Variable that holds the previous cell state. The cell state should be a zero array or the output of the previous call of LSTM.
- **x** (Variable) – Variable that holds the incoming signal. It must have the second dimension four times of that of the cell state,

Returns Two *Variable* objects `c` and `h`. `c` is the updated cell state. `h` indicates the outgoing signal.

Return type tuple

See the original paper proposing LSTM with forget gates: Long Short-Term Memory in Recurrent Neural Networks.

Example

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

```
>>> model = FunctionSet(w=F.Linear(n_units, 4 * n_units),
...                     v=F.Linear(n_units, 4 * n_units),
...                     ...)
>>> x = model.w(y) + model.v(h)
>>> c, h = F.lstm(c, x)
```

It corresponds to calculate the input sources a, i, f, o from the current input `y` and the previous output `h`. Different parameters are used for different kind of input sources.

---

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

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

This function computes its softmax along the second axis. Let $x = (x_1, x_2, \ldots, x_d)^\top$ be the d dimensional index array and $f(x)$ be the d dimensional input array. For each index x of the input array $f(x)$, it computes the probability $p(x)$ defined as $p(x) = \frac{\exp(f(x))}{\sum_{x_2} \exp(f(x))}$.

Parameters

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

Returns Output variable.

Return type *Variable*

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

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

Parameters

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

Returns Output variable.

Return type *Variable*

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*

2.4.5 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. `stride=s` and `stride=(s, s)` are equivalent. If None is specified, then it uses same stride as the pooling window size.
- **pad** (*int or (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. `stride=s` and `stride=(s, s)` are equivalent. If None is specified, then it uses same stride as the pooling window size.
- **pad** (*int or (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*

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

Spatial pyramid pooling function.

It outputs a fixed-length vector regardless of input feature map size.

It performs pooling operation to the input 4D-array `x` with different kernel sizes and padding sizes, and then flattens all dimensions except first dimension of all pooling results, and finally concatenates them along 2nd dimension.

At i-th pyramid level, the kernel size $(k_h^{(i)}, k_w^{(i)})$ and padding size $(p_h^{(i)}, p_w^{(i)})$ of pooling operation are calculated as below:

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

where $\lceil \cdot \rceil$ denotes the ceiling function, and b_h, b_w are height and width of input variable `x`, respectively. Note that index of pyramid level i is zero-based.

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

Parameters

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

Returns Ouptut variable. The shape of the output variable will be (batchsize, $c \sum_{h=0}^{H-1} 2^{2h}$, 1, 1), where c is the number of channels of input variable `x` and H is the number of pyramid levels.

Return type *Variable*

Note: This function uses some pooling classes as components to perform spatial pyramid pooling. Now it supports only `MaxPooling2D` as elemental pooling operator so far.

2.4.6 Normalization functions

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

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.
- **dtype** (*numpy.dtype*) – Type to use in computing.

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

2.4.7 Noise injecting functions

`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.**gaussian**(*mean*, *ln_var*)
Gaussian sampling function.

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

Parameters

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

Returns Output variable.

Return type *Variable*

2.4.8 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.**sigmoid_cross_entropy**(*x*, *t*, *use_cudnn=True*)
Computes cross entropy loss for sigmoid activations.

Parameters

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

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

Return type *Variable*

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

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

Computes cross entropy loss for pre-softmax activations.

Parameters

- **`x`** (Variable) – Variable holding a multidimensional array whose element indicates unnormalized log probability: the first axis of the variable represents the number of samples, and the second axis represents the number of classes. While this function computes a usual softmax cross entropy if the number of dimensions is equal to 2, it computes a cross entropy of the replicated softmax if the number of dimensions is greater than 2.
- **`t`** (Variable) – Variable holding an int32 vector of groundtruth labels.
- **`normalize`** (Variable) – Variable holding a boolean value which determines the normalization constant. If true, this function normalizes the cross entropy loss across all instances. If else, it only normalizes along a batch size.

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*, *axis=None*)

Sum of array elements over a given axis.

Parameters

- **`x`** (Variable) – Elements to sum.
- **`axis`** (*None or int*) – Axis which a sum is performed. The default (axis = None) is perform a sum over all the dimensions of the input array.

Returns Output variable.

Return type *Variable*

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

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

Parameters

- **`y`** (Variable) – Variable holding a matrix where the first dimension corresponds to the batches
- **`z`** (Variable) – Variable holding a matrix where the first dimension corresponds to the batches

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

Return type *Variable*

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

2.4.9 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.

---

class chainer.functions.**InceptionBN**(*in_channels*, *out1*, *proj3*, *out3*, *proj33*, *out33*, *pooltype*, *proj_pool=None*, *stride=1*)
Inception module of the new GoogLeNet with BatchNormalization.

This class acts like *Inception*, while InceptionBN uses the *BatchNormalization* on top of each convolution, the 5x5 convolution path is replaced by two consecutive 3x3 convolution applications, and the pooling method is configurable.

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

Parameters

- **in_channels** (*int*) – Number of channels of input arrays.
- **out1** (*int*) – Output size of the 1x1 convolution path.
- **proj3** (*int*) – Projection size of the single 3x3 convolution path.
- **out3** (*int*) – Output size of the single 3x3 convolution path.
- **proj33** (*int*) – Projection size of the double 3x3 convolutions path.
- **out33** (*int*) – Output size of the double 3x3 convolutions path.
- **pooltype** (*str*) – Pooling type. It must be either `'max'` or `'avg'`.
- **proj_pool** (*bool*) – If True, do projection in the pooling path.

- **stride** (*int*) – Stride parameter of the last convolution of each path.

See also:

Inception

2.4.10 Variational Auto-Encoder (VAE)

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

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

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

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

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

Parameters

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

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

Return type *Variable*

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

Computes the negative log-likelihood of a Bernoulli distribution.

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

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

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

Note: As this funtion uses a sigmoid function, you can pass a result of fully-connected layer (that means *Linear*) to this function directly.

Parameters

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

Returns A variable representing negative log-likelihood.

Return type *Variable*

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

Computes the negative log-likelihood of a Gaussian distribution.

Given two variable `mean` representing μ and `ln_var` representing $\log(\sigma^2)$, this function returns the negative log-likelihood of x on a Gaussian distribution $N(\mu, S)$,

$$-\log N(x; \mu, \sigma^2) = \log\left(\sqrt{(2\pi)^D |S|}\right) + \frac{1}{2}(x - \mu)^\top S^{-1}(x - \mu),$$

where D is a dimension of x and S is a diagonal matrix where $S_{ii} = \sigma_i^2$.

Parameters

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

Returns A variable representing the negative log-likelihood.

Return type *Variable*

2.5 Optimizers

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

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

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

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

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

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

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

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

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

2.6 Caffe Reference Model Support

Caffe is a popular framework maintained by BVLC at UC Berkeley. It is widely used by computer vision communities, and aims at fast computation and easy usage without any programming. The BVLC team provides trained reference models in their Model Zoo, one of the reason why this framework gets popular.

Chainer can import the reference models and emulate the network by *Function* implementations. This functionality is provided by the *chainer.functions.caffe.CaffeFunction* class.

class chainer.functions.caffe.**CaffeFunction**(*model_path*)
Function using the model file of Caffe.

Given a binary protobuf file of a Caffe model, this function loads and emulates it on *Variable* objects. It supports the official reference models provided by BVLC.

Note: This function only supports Python 2.7, since the compiled module for protocol buffers only supports Python 2. The __init__ function raises an exception in Python 3.

Note: CaffeFunction ignores the following layers:

•Layers that CaffeFunction does not support (including data layers)

•Layers that have no top blobs

•Layers whose bottom blobs are incomplete (i.e., some or all of them are not given nor computed)

Warning: It does not support full compatibility against Caffe. Some layers and configurations are not implemented in Chainer yet, though the reference models provided by the BVLC team are supported except data layers.

Example

Consider we want to extract the (unnormalized) log class probability of given images using BVLC reference CaffeNet. The model can be downloaded from:

http://dl.caffe.berkeleyvision.org/bvlc_reference_caffenet.caffemodel

We want to compute the `fc8` blob from the `data` blob. It is simply written as follows:

```
# Load the model
func = CaffeFunction('path/to/bvlc_reference_caffenet.caffemodel')

# Minibatch of size 10
x_data = numpy.ndarray((10, 3, 227, 227), dtype=numpy.float32)
...  # (Fill the minibatch here)

# Forward the pretrained net
x = Variable(x_data)
y, = func(inputs={'data': x}, outputs=['fc8'])
```

The result `y` contains the Variable corresponding to the `fc8` blob. The computational graph is memorized as a usual forward computation in Chainer, so we can run backprop through this pretrained net.

Parameters **model_path** (*str*) – Path to the binary-proto model file of Caffe.

fs

FunctionSet

A set of functions corresponding to parameterized layers of Caffe. The names of its attributes are same as the layer names of the given network.

forwards

dict

A mapping from layer names to corresponding functions.

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

Executes a subnetwork of the network.

This function acts as an interpreter of the network definition for Caffe. On execution, it interprets each layer one by one, and if the bottom blobs are already computed, then emulates the layer and stores output blobs as *Variable* objects.

Parameters

- **inputs** (*dict*) – A dictionary whose key-value pairs indicate initial correspondences between blob names and *Variable* objects.

- **outputs** (*Iterable*) – A list of blob names whose corresponding *Variable* objects are returned.
- **disable** (*Iterable*) – A list of layer names that will be ignored during the forward computation.
- **train** (*bool*) – If True, this function emulates the TRAIN phase of the Caffe layers. Otherwise, it emulates the TEST phase.

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

Return type tuple

2.7 Visualization of Computational Graph

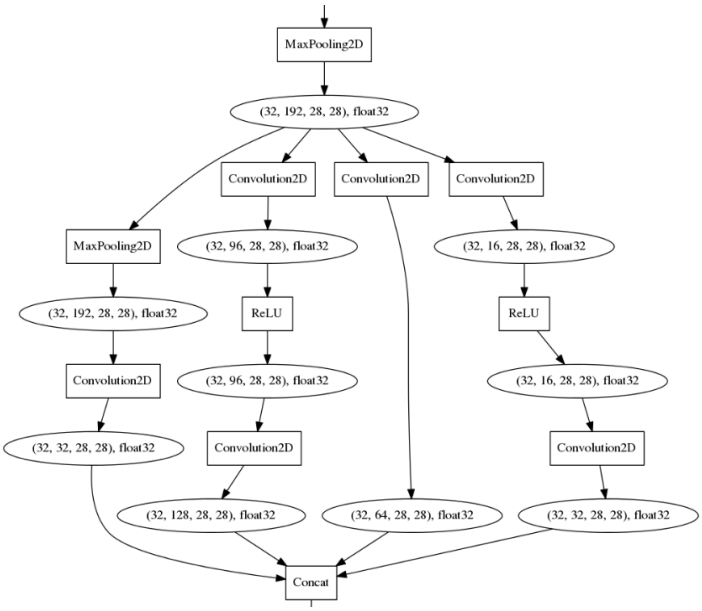
As neural networks get larger and complicated, it gets much harder to confirm if their architectures are constructed properly. Chainer supports visualization of computational graphs. Users can generate computational graphs by invoking *build_computational_graph()*. Generated computational graphs are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

Basic usage is as follows:

```
import chainer.computational_graph as c
...
g = c.build_computational_graph(vs)
with open('path/to/output/file', 'w') as o:
    o.write(g.dump())
```

where `vs` is list of `Variable` instances and `g` is an instance of *ComputationalGraph*. This code generates the computational graph that are backward-reachable (i.e. reachable by repetition of steps backward) from at least one of `vs`.

Here is an example of (a part of) the generated graph (inception(3a) in GoogLeNet). This example is from `example/imagenet`.



chainer.computational_graph.**build_computational_graph**(*outputs*, *remove_split=True*)
Builds a graph of functions and variables backward-reachable from outputs.

Parameters

- **outputs** (*list*) – nodes from which the graph is constructed. Each element of outputs must be either `Variable` object or `Function` object.
- **remove_split** (*bool*) – If it is `True`, this function hides `Split` functions and related variables from the graph.

Returns

A graph consisting of nodes and edges that are backward-reachable from at least one of `outputs`.

If `unchain_backward` was called in some variable in the computational graph before this function, backward step is stopped at this variable.

For example, suppose that computational graph is as follows:

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

Let `outputs = [y, z]`. If `remove_split` is `False`, this method generates the graph itself. On the other hand, if `remove_split` is `True`, `splitter`, `x'` and `x''` are removed from the graph and `x` is directly connected to `f` and `g`. Resulting graph will be:

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

Next, let `outputs = [y]`. Note that `z`, `g`, and `x''` are not backward-reachable from `y`. If `remove_split` is `False`, this function removes these unreachable nodes to get:

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

If `remove_split` is `True`, we further remove splitter and `x'` to get:

```
x ---> f ---> y
```

See `TestGraphBuilder` for details.

Return type *ComputationalGraph*

class chainer.computational_graph.**ComputationalGraph**(*nodes*, *edges*)
Class that represents computational graph.

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

dump(*format='dot'*)
Dumps graph as a text.

Parameters

- **format** (*str*) – The graph language name of the output.
- **it must be 'dot'.** (*Currently,*) –

Returns The graph in specified format.

Return type str

CHAPTER 3

CuPy Reference Manual

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

3.1 CuPy Overview

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

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

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

CuPy also includes following features for performance:

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

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

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

Memory layout

base ctypes itemsize flags nbytes shape size strides

Data type

dtype

Other attributes

T

Array conversion

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

Shape manipulation

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

Item selection and manipulation

take() diagonal()

Calculation

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

Arithmetic and comparison operations

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

Special methods

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

Memory transfer

get() set()

3.1.2 A list of supported routines of cupy module

Array creation routines

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

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

arange() linspace()

diag() diagflat()

Array manipulation routines

copyto()

reshape() ravel()

rollaxis() swapaxes() transpose()

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

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

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

Binary operations

bitwise_and bitwise_or bitwise_xor invert left_shift right_shift

Indexing routines

take() diagonal()

Input and output

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

array_repr() array_str()

Linear algebra

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

trace()

Logic functions

isfinite isinf isnan

logical_and logical_or logical_not logical_xor

greater greater_equal less less_equal equal not_equal

Mathematical functions

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

sinh cosh tanh arcsinh arccosh arctanh

rint floor ceil trunc

sum() prod()

exp expm1 exp2 log log10 log2 log1p logaddexp logaddexp2

signbit copysign ldexp frexp nextafter

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

clip() sqrt square absolute sign maximum minimum fmax fmin

Sorting, searching, and counting

argmax() argmin()

Statistics

amin() amax()

mean() var() std()

Other

asnumpy()

3.2 Multi-Dimensional Array (ndarray)

class cupy.**ndarray**(*shape*, *dtype=<type 'float'>*, *memptr=None*, *strides=None*)

Multi-dimensional array on a CUDA device.

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

Parameters

- **shape** (*tuple of ints*) – Length of axes.
- **dtype** – Data type. It must be an argument of numpy.dtype.
- **memptr** (cupy.cuda.MemoryPointer) – Pointer to the array content head.
- **strides** (*tuple of ints*) – The strides for axes.

data

cupy.cuda.MemoryPointer

Pointer to the array content head.

base
None or cupy.ndarray

Base array from which this array is created as a view.

T
Shape-reversed view of the array.

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

argmax(*axis=None*, *out=None*, *dtype=None*, *keepdims=False*)
Returns the indices of the maximum along a given axis.

See also:

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

argmin(*axis=None*, *out=None*, *dtype=None*, *keepdims=False*)
Returns the indices of the minimum along a given axis.

See also:

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

astype(*dtype*, *copy=True*)
Casts the array to given data type.

Parameters

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

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

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

See also:

numpy.ndarray.astype()

clip(*a_min*, *a_max*, *out=None*)
Returns an array with values limited to [a_min, a_max].

See also:

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

copy()
Returns a copy of the array.

See also:

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

ctypes
C representation of the array.

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

Note: The returned value does not have compatibility with numpy.ndarray.ctypes.

device
CUDA device on which this array resides.

diagonal(*offset=0*, *axis1=0*, *axis2=1*)
Returns a view of the specified diagonals.

See also:

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

dot(*b*, *out=None*)
Returns the dot product with given array.

See also:

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

dtype
Dtype object of element type.

See also:

Data type objects (dtype)

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

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

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

fill(*value*)
Fills the array with a scalar value.

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

See also:

numpy.ndarray.fill()

flags
Object containing memory-layout information.

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

See also:

numpy.ndarray.flags

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

It currently supports C-order only.

Returns A copy of the array with one dimension.

Return type *cupy.ndarray*

See also:

numpy.ndarray.flatten()

get(*stream=None*)
Returns a copy of the array on host memory.

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

Returns Copy of the array on host memory.

Return type numpy.ndarray

itemsize
Size of each element in bytes.

See also:

`numpy.ndarray.itemsize`

max(*axis=None*, *out=None*, *dtype=None*, *keepdims=False*)
Returns the maximum along a given axis.

See also:

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

mean(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the mean along a given axis.

See also:

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

min(*axis=None*, *out=None*, *dtype=None*, *keepdims=False*)
Returns the minimum along a given axis.

See also:

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

nbytes
Size of whole elements in bytes.

It does not count skips between elements.

See also:

`numpy.ndarray.nbytes`

ndim
Number of dimensions.

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

See also:

`numpy.ndarray.ndim`

prod(*axis=None*, *dtype=None*, *out=None*, *keepdims=None*)
Returns the product along a given axis.

See also:

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

ravel()
Returns an array flattend into one dimension.

See also:

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

reduced_view(*dtype=None*)
Returns a view of the array with minimum number of dimensions.

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

Returns A view of the array with reduced dimensions.

Return type *cupy.ndarray*

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

See also:

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

set(*arr*, *stream=None*)
Copies an array on the host memory to cuda.ndarray.

Parameters

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

shape
Lengths of axes.

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

size
Number of elements this array holds.

This is equivalent to product over the shape tuple.

See also:

numpy.ndarray.size

squeeze(*axis=None*)
Returns a view with size-one axes removed.

See also:

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

std(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the standard deviation along a given axis.

See also:

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

strides
Strides of axes in bytes.

See also:

numpy.ndarray.strides

sum(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the sum along a given axis.

See also:

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

swapaxes(*axis1*, *axis2*)
Returns a view of the array with two axes swapped.

See also:

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

take(*indices*, *axis=None*, *out=None*)
Returns an array of elements at given indices along the axis.

See also:

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

tofile(*fid*, *sep=''*, *format='%s'*)
Writes the array to a file.

See also:

numpy.ndarray.tolist()

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

Returns The possibly nested Python list of array elements.

Return type list

See also:

numpy.ndarray.tolist()

trace(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)
Returns the sum along diagonals of the array.

See also:

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

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

See also:

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

var(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the variance along a given axis.

See also:

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

view(*dtype=None*)
Returns a view of the array.

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

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

Return type *cupy.ndarray*

See also:

numpy.ndarray.view()

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

Parameters

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

Returns Converted array on the host memory.

Return type numpy.ndarray

3.3 Universal Functions (ufunc)

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

- Broadcasting
- Output type determination
- Casting rules

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

3.3.1 Ufunc class

class cupy.**ufunc**(*name*, *nin*, *nout*, *ops*, *preamble=''*, *doc=''*)
Universal function.

name
str

The name of the universal function.

nin
int

Number of input arguments.

nout
int

Number of output arguments.

nargs
int

Number of all arguments.

__call__(**args*, ***kwargs*)
Applies the universal function to arguments elementwise.

Parameters

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

- **dtype** – Data type specifier.

Returns Output array or a tuple of output arrays.

types
A list of type signatures.

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

3.3.2 Available ufuncs

Math operations

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

Trigonometric functions

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

Bit-twiddling functions

bitwise_and bitwise_or bitwise_xor invert left_shift right_shift

Comparison functions

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

Floating point values

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

3.4 Routines

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

3.4.1 Array Creation Routines

Basic creation routines

cupy.**empty**(*shape*, *dtype=<type 'float'>*)
Returns an array without initializing the elements.

This function currently does not support `order` option.

Parameters

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

Returns A new array with elements not initialized.

Return type *cupy.ndarray*

See also:

`numpy.empty()`

`cupy.`**`empty_like`**(*a*, *dtype=None*)
Returns a new array with same shape and dtype of a given array.

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

Parameters

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

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

Return type *cupy.ndarray*

See also:

`numpy.empty_like()`

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

Parameters

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

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

Return type *cupy.ndarray*

See also:

`numpy.eye()`

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

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

Parameters

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

Returns A 2-D identity array.

Return type *cupy.ndarray*

See also:

`numpy.identity()`

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

This function currently does not support `order` option.

Parameters

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

Returns An array filled with ones.

Return type *cupy.ndarray*

See also:

`numpy.ones()`

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

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

Parameters

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

Returns An array filled with ones.

Return type *cupy.ndarray*

See also:

`numpy.ones_like()`

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

This function currently does not support `order` option.

Parameters

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

Returns An array filled with ones.

Return type *cupy.ndarray*

See also:

`numpy.zeros()`

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

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

Parameters

- **a** (cupy.ndarray) – Base array.

- **dtype** – Data type specifier. The dtype of `a` is used by default.

Returns An array filled with ones.

Return type *cupy.ndarray*

See also:

`numpy.zeros_like()`

`cupy.full`(*shape*, *fill_value*, *dtype=None*)
Returns a new array of given shape and dtype, filled with a given value.

This function currently does not support `order` option.

Parameters

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

Returns An array filled with `fill_value`.

Return type *cupy.ndarray*

See also:

`numpy.full()`

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

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

Parameters

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

Returns An array filled with `fill_value`.

Return type *cupy.ndarray*

See also:

`numpy.full_like()`

Creation from other data

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

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

Parameters

- **obj** – cupy.ndarray object or any other object that can be passed to `numpy.array()`.
- **dtype** – Data type specifier.
- **copy** (*bool*) – If False, this function returns `obj` if possible. Otherwise this function always returns a new array.

- **ndmin** (*int*) – Minimum number of dimensions. Ones are inserted to the head of the shape if needed.

Returns An array on the current device.

Return type *cupy.ndarray*

See also:

`numpy.array()`

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

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

Parameters

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

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

Return type *cupy.ndarray*

See also:

`numpy.asarray()`

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

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

See also:

`cupy.asarray()`, `numpy.asanyarray()`

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

Parameters

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

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

Return type *cupy.ndarray*

See also:

`numpy.ascontiguousarray()`

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

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

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

Returns The copy of `a` on the current device.

Return type *cupy.ndarray*

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

Numerical ranges

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

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

Parameters

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

Returns The 1-D array of range values.

Return type *cupy.ndarray*

See also:

numpy.arange()

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

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

Parameters

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

Returns The 1-D array of ranged values.

Return type *cupy.ndarray*

Matrix creation

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

Parameters

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

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

Return type *cupy.ndarray*

See also:

`numpy.diag()`

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

Parameters

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

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

Return type *cupy.ndarray*

3.4.2 Array Manipulation Routines

Basic manipulations

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

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

Parameters

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

See also:

`numpy.copyto()`

Shape manipulation

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

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

This function currently does not support `order` option.

Parameters

- **`a`** (cupy.ndarray) – Array to be reshaped.

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

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

Return type *cupy.ndarray*

See also:

`numpy.reshape()`

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

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

This function currently does not support `order` option.

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

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

Return type *cupy.ndarray*

See also:

`numpy.ravel()`

Transposition

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

Parameters

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

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

Return type *cupy.ndarray*

See also:

`numpy.rollaxis()`

`cupy.swapaxes(a, axis1, axis2)`
Swaps the two axes.

Parameters

- **a** (cupy.ndarray) – Array to swap the axes.
- **axis1** (*int*) – The first axis to swap.
- **axis2** (*int*) – The second axis to swap.

Returns A view of `a` that the two axes are swapped.

Return type *cupy.ndarray*

See also:

numpy.swapaxes()

cupy.**transpose**(*a*, *axes=None*)
Permutes the dimensions of an array.

Parameters

- **a** (cupy.ndarray) – Array to permute the dimensions.
- **axes** (*tuple of ints*) – Permutation of the dimensions. This function reverses the shape by default.

Returns A view of a that the dimensions are permuted.

Return type *cupy.ndarray*

See also:

numpy.transpose()

Edit dimensionalities

cupy.**atleast_1d**(**arys*)
Converts arrays to arrays with dimensions >= 1.

Parameters **arys** (*tuple of arrays*) – Arrays to be converted. All arguments must be cupy.ndarray objects. Only zero-dimensional array is affected.

Returns If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

See also:

numpy.atleast_1d()

cupy.**atleast_2d**(**arys*)
Converts arrays to arrays with dimensions >= 2.

If an input array has dimensions less than two, then this function inserts new axes at the head of dimensions to make it have two dimensions.

Parameters **arys** (*tuple of arrays*) – Arrays to be converted. All arguments must be cupy.ndarray objects.

Returns If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

See also:

numpy.atleast_2d()

cupy.**atleast_3d**(**arys*)
Converts arrays to arrays with dimensions >= 3.

If an input array has dimensions less than three, then this function inserts new axes to make it have three dimensions. The place of the new axes are following:

•If its shape is `()`, then the shape of output is `(1, 1, 1)`.

•If its shape is `(N,)`, then the shape of output is `(1, N, 1)`.

•If its shape is `(M, N)`, then the shape of output is `(M, N, 1)`.

•Otherwise, the output is the input array itself.

Parameters **arys** (*tuple of arrays*) – Arrays to be converted. All arguments must be cupy.ndarray objects.

Returns If there are only one input, then it returns its converted version. Otherwise, it returns a list of converted arrays.

See also:

numpy.atleast_3d()

class cupy.**broadcast**(**arrays*)
Object that performs broadcasting.

CuPy actually uses this class to support broadcasting in various operations. Note that this class does not provide an iterator.

Parameters **arrays** (*tuple of arrays*) – Arrays to be broadcasted.

shape
tuple of ints

The broadcasted shape.

nd
int

Number of dimensions of the broadcasted shape.

size
int

Total size of the broadcasted shape.

values
list of arrays

The broadcasted arrays.

See also:

numpy.broadcast

cupy.**broadcast_arrays**(**args*)
Broadcasts given arrays.

Parameters **args** (*tuple of arrays*) – Arrays to broadcast for each other.

Returns A list of broadcasted arrays.

Return type list

See also:

numpy.broadcast_arrays()

cupy.**expand_dims**(*a*, *axis*)
Expands given arrays.

Parameters

- **a** (cupy.ndarray) – Array to be expanded.
- **axis** (*int*) – Position where new axis is to be inserted.

Returns The number of dimensions is one greater than that of the input array.

Return type *cupy.ndarray*

See also:

numpy.expand_dims()

cupy.**squeeze**(*a*, *axis=None*)
Removes size-one axes from the shape of an array.

Parameters

- **a** (cupy.ndarray) – Array to be reshaped.
- **axis** (*int or tuple of ints*) – Axes to be removed. This function removes all size-one axes by default. If one of the specified axes is not of size one, an exception is raised.

Returns An array without (specified) size-one axes.

Return type *cupy.ndarray*

See also:

numpy.squeeze()

Joining arrays along axis

cupy.**column_stack**(*tup*)
Stacks 1-D and 2-D arrays as columns into a 2-D array.

A 1-D array is first converted to a 2-D column array. Then, the 2-D arrays are concatenated along the second axis.

Parameters **tup** (*sequence of arrays*) – 1-D or 2-D arrays to be stacked.

Returns A new 2-D array of stacked columns.

Return type *cupy.ndarray*

See also:

numpy.column_stack()

cupy.**concatenate**(*tup*, *axis=0*)
Joins arrays along an axis.

Parameters

- **tup** (*sequence of arrays*) – Arrays to be joined. All of these should have same dimensionalities except the specified axis.
- **axis** (*int*) – The axis to join arrays along.

Returns Joined array.

Return type *cupy.ndarray*

See also:

numpy.concatenate()

cupy.**vstack**(*tup*)
Stacks arrays vertically.

If an input array has one dimension, then the array is treated as a horizontal vector and stacked along the additional axis at the head. Otherwise, the array is stacked along the first axis.

Parameters **tup** (*sequence of arrays*) – Arrays to be stacked. Each array is converted by *cupy.atleast_2d()* before stacking.

Returns Stacked array.

Return type *cupy.ndarray*

See also:

`numpy.dstack()`

`cupy.`**`hstack`**(*tup*)
Stacks arrays horizontally.

If an input array has one dimension, then the array is treated as a horizontal vector and stacked along the first axis. Otherwise, the array is stacked along the second axis.

Parameters **`tup`** (*sequence of arrays*) – Arrays to be stacked.

Returns Stacked array.

Return type *cupy.ndarray*

See also:

`numpy.hstack()`

`cupy.`**`dstack`**(*tup*)
Stacks arrays along the third axis.

Parameters **`tup`** (*sequence of arrays*) – Arrays to be stacked. Each array is converted by *`cupy.atleast_3d()`* before stacking.

Returns Stacked array.

Return type *cupy.ndarray*

See also:

`numpy.dstack()`

Splitting arrays along axis

`cupy.`**`array_split`**(*ary*, *indices_or_sections*, *axis=0*)
Splits an array into multiple sub arrays along a given axis.

This function is almost equivalent to *`cupy.split()`*. The only difference is that this function allows an integer sections that does not evenly divide the axis.

See also:

`cupy.split()` for more detail, `numpy.array_split()`

`cupy.`**`split`**(*ary*, *indices_or_sections*, *axis=0*)
Splits an array into multiple sub arrays along a given axis.

Parameters

- **`ary`** (cupy.ndarray) – Array to split.
- **`indices_or_sections`** (*int or sequence of ints*) – A value indicating how to divide the axis. If it is an integer, then is treated as the number of sections, and the axis is evenly divided. Otherwise, the integers indicate indices to split at. Note that the sequence on the device memory is not allowed.
- **`axis`** (*int*) – Axis along which the array is split.

Returns A list of sub arrays. Eacy array is a view of the corresponding input array.

See also:

numpy.split()

cupy.**vsplit**(*ary*, *indices_or_sections*)
Splits an array into multiple sub arrays along the first axis.

This is equivalent to split with axis=0.

See also:

cupy.split() for more detail, numpy.dsplit()

cupy.**hsplit**(*ary*, *indices_or_sections*)
Splits an array into multiple sub arrays horizontally.

This is equivalent to split with axis=0 if ary has one dimension, and otherwise that with axis=1.

See also:

cupy.split() for more detail, numpy.hsplit()

cupy.**dsplit**(*ary*, *indices_or_sections*)
Splits an array into multiple sub arrays along the third axis.

This is equivalent to split with axis=2.

See also:

cupy.split() for more detail, numpy.dsplit()

3.4.3 Binary Operations

Elementwise bit operations

cupy.**bitwise_and = <ufunc 'cupy_bitwise_and'>**
Computes the bitwise AND of two arrays elementwise.

Only integer and boolean arrays are handled.

See also:

numpy.bitwise_and

cupy.**bitwise_or = <ufunc 'cupy_bitwise_or'>**
Computes the bitwise OR of two arrays elementwise.

Only integer and boolean arrays are handled.

See also:

numpy.bitwise_or

cupy.**bitwise_xor = <ufunc 'cupy_bitwise_xor'>**
Computes the bitwise XOR of two arrays elementwise.

Only integer and boolean arrays are handled.

See also:

numpy.bitwise_xor

cupy.**invert = <ufunc 'cupy_invert'>**
Computes the bitwise NOT of an array elementwise.

Only integer and boolean arrays are handled.

See also:

numpy.invert

cupy.**left_shift** = <ufunc 'cupy_left_shift'>
Shifts the bits of each integer element to the left.

Only integer arrays are handled.

See also:

numpy.left_shift

cupy.**right_shift** = <ufunc 'cupy_right_shift'>
Shifts the bits of each integer element to the right.

Only integer arrays are handled

See also:

numpy.right_shift

3.4.4 Indexing Routines

cupy.**take**(*a*, *indices*, *axis=None*, *out=None*)
Takes elements of an array at specified indices along an axis.

This is an implementation of "fancy indexing" at single axis.

This function does not support `mode` option.

Parameters

- **a** (cupy.ndarray) – Array to extract elements.
- **indices** (*int or array-like*) – Indices of elements that this function takes.
- **axis** (*int*) – The axis along which to select indices. The flattened input is used by default.
- **out** (cupy.ndarray) – Output array. If provided, it should be of appropriate shape and dtype.

Returns The result of fancy indexing.

Return type *cupy.ndarray*

See also:

numpy.take()

cupy.**diagonal**(*a*, *offset=0*, *axis1=0*, *axis2=1*)
Returns specified diagonals.

This function extracts the diagonals along two specified axes. The other axes are not changed. This function returns a writable view of this array as NumPy 1.10 will do.

Parameters

- **a** (cupy.ndarray) – Array from which the diagonals are taken.
- **offset** (*int*) – Index of the diagonals. Zero indicates the main diagonals, a positive value upper diagonals, and a negative value lower diagonals.
- **axis1** (*int*) – The first axis to take diagonals from.
- **axis2** (*int*) – The second axis to take diagonals from.

Returns A view of the diagonals of `a`.

Return type *cupy.ndarray*

See also:

`numpy.diagonal()`

3.4.5 Input and Output

NPZ files

`cupy.`**`load`**(*file*, *mmap_mode=None*)

Loads arrays or pickled objects from `.npy`, `.npz` or pickled file.

This function just calls `numpy.load` and then sends the arrays to the current device. NPZ file is converted to NpzFile object, which defers the transfer to the time of accessing the items.

Parameters

- **`file`** (*file-like object or string*) – The file to read.
- **`mmap_mode`** (*None, 'r+', 'r', 'w+', 'c'*) – If not None, memory-map the file to construct an intermediate `numpy.ndarray` object and transfer it to the current device.

Returns CuPy array or NpzFile object depending on the type of the file. NpzFile object is a dictionary-like object with the context manager protocol (which enables us to use *with* statement on it).

See also:

`numpy.load()`

`cupy.`**`save`**(*file*, *arr*)

Saves an array to a binary file in `.npy` format.

Parameters

- **`file`** (*file or str*) – File or filename to save.
- **`arr`** (*array_like*) – Array to save. It should be able to feed to *`cupy.asnumpy()`*.

See also:

`numpy.save()`

`cupy.`**`savez`**(*file*, **args*, ***kwds*)

Saves one or more arrays into a file in uncompressed `.npz` format.

Arguments without keys are treated as arguments with automatic keys named `arr_0`, `arr_1`, etc. corresponding to the positions in the argument list. The keys of arguments are used as keys in the `.npz` file, which are used for accessing NpzFile object when the file is read by *`cupy.load()`* function.

Parameters

- **`file`** (*file or str*) – File or filename to save.
- **`*args`** – Arrays with implicit keys.
- **`**kwds`** – Arrays with explicit keys.

See also:

`numpy.savez()`

cupy.**savez_compressed**(*file*, **args*, ***kwds*)
Saves one or more arrays into a file in compressed `.npz` format.

It is equivalent to *cupy.savez()* function except the outptu file is compressed.

See also:

cupy.savez() for more detail, numpy.savez_compressed()

String formatting

cupy.**array_repr**(*arr*, *max_line_width=None*, *precision=None*, *suppress_small=None*)
Returns the string representation of an array.

Parameters

- **arr** (*array_like*) – Input array. It should be able to feed to *cupy.asnumpy()*.
- **max_line_width** (*int*) – The maximum number of line lengths.
- **precision** (*int*) – Floating point precision. It uses the current printing precision of NumPy.
- **suppress_small** (*bool*) – If True, very small numbers are printed as zeros

Returns The string representation of `arr`.

Return type str

See also:

numpy.array_repr()

cupy.**array_str**(*arr*, *max_line_width=None*, *precision=None*, *suppress_small=None*)
Returns the string representation of the content of an array.

Parameters

- **arr** (*array_like*) – Input array. It should be able to feed to *cupy.asnumpy()*.
- **max_line_width** (*int*) – The maximum number of line lengths.
- **precision** (*int*) – Floating point precision. It uses the current printing precision of NumPy.
- **suppress_small** (*bool*) – If True, very small number are printed as zeros.

See also:

numpy.array_str()

3.4.6 Linear Algebra

Matrix and vector products

cupy.**dot**(*a*, *b*, *out=None*)
Returns a dot product of two arrays.

For arrays with more than one axis, it computes the dot product along the last axis of `a` and the second-to-last axis of `b`. This is just a matrix product if the both arrays are 2-D. For 1-D arrays, it uses their unique axis as an axis to take dot product over.

Parameters

- **a** (cupy.ndarray) – The left argument.
- **b** (cupy.ndarray) – The right argument.
- **out** (cupy.ndarray) – Output array.

Returns The dot product of `a` and `b`.

Return type *cupy.ndarray*

See also:

`numpy.dot()`

`cupy.`**vdot**(*a*, *b*)

Returns the dot product of two vectors.

The input arrays are flattened into 1-D vectors and then it performs inner product of these vectors.

Parameters

- **a** (cupy.ndarray) – The first argument.
- **b** (cupy.ndarray) – The second argument.

Returns Zero-dimensional array of the dot product result.

Return type *cupy.ndarray*

See also:

`numpy.vdot()`

`cupy.`**inner**(*a*, *b*)

Returns the inner product of two arrays.

It uses the last axis of each argument to take sum product.

Parameters

- **a** (cupy.ndarray) – The first argument.
- **b** (cupy.ndarray) – The second argument.

Returns The inner product of `a` and `b`.

Return type *cupy.ndarray*

See also:

`numpy.inner()`

`cupy.`**outer**(*a*, *b*, *out=None*)

Returns the outer product of two vectors.

The input arrays are flattened into 1-D vectors and then it performs outer product of these vectors.

Parameters

- **a** (cupy.ndarray) – The first argument.
- **b** (cupy.ndarray) – The second argument.
- **out** (cupy.ndarray) – Output array.

Returns 2-D array of the outer product of `a` and `b`.

Return type *cupy.ndarray*

See also:

numpy.outer()

cupy.**tensordot**(*a*, *b*, *axes=2*)

Returns the tensor dot product of two arrays along specified axes.

This is equivalent to compute dot product along the specified axes which are treated as one axis by reshaping.

Parameters

- **a** (cupy.ndarray) – The first argument.
- **b** (cupy.ndarray) – The second argument.
- **axes** –
  - If it is an integer, then `axes` axes at the last of `a` and the first of `b` are used.
  - If it is a pair of sequences of integers, then these two sequences specify the list of axes for `a` and `b`. The corresponding axes are paired for sum-product.
- **out** (cupy.ndarray) – Output array.

Returns The tensor dot product of `a` and `b` along the axes specified by `axes`.

Return type *cupy.ndarray*

See also:

numpy.tensordot()

Norms etc.

cupy.**trace**(*a*, *offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Returns the sum along the diagonals of an array.

It computes the sum along the diagonals at `axis1` and `axis2`.

Parameters

- **a** (cupy.ndarray) – Array to take trace.
- **offset** (*int*) – Index of diagonals. Zero indicates the main diagonal, a positive value an upper diagonal, and a negative value a lower diagonal.
- **axis1** (*int*) – The first axis along which the trace is taken.
- **axis2** (*int*) – The second axis along which the trace is taken.
- **dtype** – Data type specifier of the output.
- **out** (cupy.ndarray) – Output array.

Returns The trace of `a` along axes `(axis1, axis2)`.

Return type *cupy.ndarray*

See also:

numpy.trace()

3.4.7 Logic Functions

Infinities and NaNs

cupy.**isfinite** = <ufunc 'cupy_isfinite'>
Tests finiteness elementwise.

Each element of returned array is True only if the corresponding element of the input is finite (i.e. not an infinity nor NaN).

See also:

numpy.isfinite

cupy.**isinf** = <ufunc 'cupy_isinf'>
Tests if each element is the positive or negative infinity.

See also:

numpy.isinf

cupy.**isnan** = <ufunc 'cupy_isnan'>
Tests if each element is a NaN.

See also:

numpy.isnan

Logic operations

cupy.**logical_and** = <ufunc 'cupy_logical_and'>
Computes the logical AND of two arrays.

See also:

numpy.logical_and

cupy.**logical_or** = <ufunc 'cupy_logical_or'>
Computes the logical OR of two arrays.

See also:

numpy.logical_or

cupy.**logical_not** = <ufunc 'cupy_logical_not'>
Computes the logical NOT of an array.

See also:

numpy.logical_not

cupy.**logical_xor** = <ufunc 'cupy_logical_xor'>
Computes the logical XOR of two arrays.

See also:

numpy.logical_xor

Comparison operations

cupy.**greater** = <ufunc 'cupy_greater'>
Tests elementwise if `x1 > x2`.

See also:

`numpy.greater`

cupy.**greater_equal** = <ufunc 'cupy_greater_equal'>
Tests elementwise if `x1 >= x2`.

See also:

`numpy.greater_equal`

cupy.**less** = <ufunc 'cupy_less'>
Tests elementwise if `x1 < x2`.

See also:

`numpy.less`

cupy.**less_equal** = <ufunc 'cupy_less_equal'>
Tests elementwise if `x1 <= x2`.

See also:

`numpy.less_equal`

cupy.**equal** = <ufunc 'cupy_equal'>
Tests elementwise if `x1 == x2`.

See also:

`numpy.equal`

cupy.**not_equal** = <ufunc 'cupy_not_equal'>
Tests elementwise if `x1 != x2`.

See also:

`numpy.equal`

3.4.8 Mathematical Functions

Trigonometric functions

cupy.**sin** = <ufunc 'cupy_sin'>
Elementwise sine function.

See also:

`numpy.sin`

cupy.**cos** = <ufunc 'cupy_cos'>
Elementwise cosine function.

See also:

`numpy.cos`

cupy.**tan = <ufunc 'cupy_tan'>**
Elementwise tangent function.

See also:

numpy.tan

cupy.**arcsin = <ufunc 'cupy_arcsin'>**
Elementwise inverse-sine function (a.k.a. arcsine function).

See also:

numpy.arcsin

cupy.**arccos = <ufunc 'cupy_arccos'>**
Elementwise inverse-cosine function (a.k.a. arccosine function).

See also:

numpy.arccos

cupy.**arctan = <ufunc 'cupy_arctan'>**
Elementwise inverse-tangent function (a.k.a. arctangent function).

See also:

numpy.arctan

cupy.**hypot = <ufunc 'cupy_hypot'>**
Computes the hypoteneous of orthogonal vectors of given length.

This is equivalent to `sqrt(x1 **2 + x2 ** 2)`, while this function is more efficient.

See also:

numpy.hypot

cupy.**arctan2 = <ufunc 'cupy_arctan2'>**
Elementwise inverse-tangent of the ratio of two arrays.

See also:

numpy.arctan2

cupy.**deg2rad = <ufunc 'cupy_deg2rad'>**
Converts angles from degrees to radians elementwise.

See also:

numpy.deg2rad, numpy.radians

cupy.**rad2deg = <ufunc 'cupy_rad2deg'>**
Converts angles from radians to degrees elementwise.

See also:

numpy.rad2deg, numpy.degrees

cupy.**degrees = <ufunc 'cupy_rad2deg'>**
Converts angles from radians to degrees elementwise.

See also:

numpy.rad2deg, numpy.degrees

cupy.**radians = <ufunc 'cupy_deg2rad'>**
Converts angles from degrees to radians elementwise.

See also:

numpy.deg2rad, numpy.radians

Hyperbolic functions

cupy.**sinh = <ufunc 'cupy_sinh'>**
Elementwise hypoerbolic sine function.

See also:

numpy.sinh

cupy.**cosh = <ufunc 'cupy_cosh'>**
Elementwise hypoerbolic cosine function.

See also:

numpy.cosh

cupy.**tanh = <ufunc 'cupy_tanh'>**
Elementwise hyperbolic tangent function.

See also:

numpy.tanh

cupy.**arcsinh = <ufunc 'cupy_arcsinh'>**
Elementwise inverse of hyperbolic sine function.

See also:

numpy.arcsinh

cupy.**arccosh = <ufunc 'cupy_arccosh'>**
Elementwise inverse of hyperbolic cosine function.

See also:

numpy.arccosh

cupy.**arctanh = <ufunc 'cupy_arctanh'>**
Elementwise inverse of hyperbolic tangent function.

See also:

numpy.arctanh

Rounding

cupy.**rint = <ufunc 'cupy_rint'>**
Rounds eacy element of an array to the nearest integer.

See also:

numpy.rint

cupy.**floor = <ufunc 'cupy_floor'>**
Rounds each element of an array to its floow integer.

See also:

numpy.floor

cupy.**ceil** = <ufunc 'cupy_ceil'>
Rounds each element of an array to its ceil integer.

See also:

numpy.ceil

cupy.**trunc** = <ufunc 'cupy_trunc'>
Rounds each element of an array towards zero.

See also:

numpy.trunc

Sums and products

cupy.**sum**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the sum of an array along given axes.

Parameters

- **a** (cupy.ndarray) – Array to take sum.
- **axis** (*int or sequence of ints*) – Axes along which the sum is taken.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

Returns The result array.

Return type *cupy.ndarray*

See also:

numpy.sum()

cupy.**prod**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the product of an array along given axes.

Parameters

- **a** (cupy.ndarray) – Array to take product.
- **axis** (*int or sequence of ints*) – Axes along which the product is taken.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the specified axes are remained as axes of length one.

Returns The result array.

Return type *cupy.ndarray*

See also:

numpy.prod()

Exponential and logarithm functions

cupy.**exp = <ufunc 'cupy_exp'>**
Elementwise exponential function.

See also:

numpy.exp

cupy.**expm1 = <ufunc 'cupy_expm1'>**
Computes `exp(x) - 1` elementwise.

See also:

numpy.expm1

cupy.**exp2 = <ufunc 'cupy_exp2'>**
Elementwise exponentiation with base 2.

See also:

numpy.exp2

cupy.**log = <ufunc 'cupy_log'>**
Elementwise natural logarithm function.

See also:

numpy.log

cupy.**log10 = <ufunc 'cupy_log10'>**
Elementwise common logarithm function.

See also:

numpy.log10

cupy.**log2 = <ufunc 'cupy_log2'>**
Elementwise binary logarithm function.

See also:

numpy.log2

cupy.**log1p = <ufunc 'cupy_log1p'>**
Computes `log(1 + x)` elementwise.

See also:

numpy.log1p

cupy.**logaddexp = <ufunc 'cupy_logaddexp'>**
Computes `log(exp(x1) + exp(x2))` elementwise.

See also:

numpy.logaddexp

cupy.**logaddexp2 = <ufunc 'cupy_logaddexp2'>**
Computes `log2(exp2(x1) + exp2(x2))` elementwise.

See also:

numpy.logaddexp2

Floating point manipulations

cupy.**signbit = <ufunc 'cupy_signbit'>**
Tests elementwise if the sign bit is set (i.e. less than zero).

See also:

numpy.signbit

cupy.**copysign = <ufunc 'cupy_copysign'>**
Returns the first arugment with the sign bit of the second elementwise.

See also:

numpy.copysign

cupy.**ldexp = <ufunc 'cupy_ldexp'>**
Computes `x1 * 2 ** x2` elementwise.

See also:

numpy.ldexp

cupy.**frexp = <ufunc 'cupy_frexp'>**
Decomposes each element to mantissa and two's exponent.

This ufunc outputs two arrays of the input dtype and the `int` dtype.

See also:

numpy.frexp

cupy.**nextafter = <ufunc 'cupy_nextafter'>**
Computes the nearest neighbor float values towards the second argument.

See also:

numpy.nextafter

Arithmetic operations

cupy.**negative = <ufunc 'cupy_negative'>**
Takes numerical negative elementwise.

See also:

numpy.negative

cupy.**add = <ufunc 'cupy_add'>**
Adds two arrays elementwise.

See also:

numpy.add

cupy.**subtract = <ufunc 'cupy_subtract'>**
Subtracts arguments elementwise.

See also:

numpy.subtract

cupy.**multiply = <ufunc 'cupy_multiply'>**
Multiplies two arrays elementwise.

See also:

numpy.multiply

cupy.**divide** = <ufunc 'cupy_divide'>
Divides arguments elementwise.

See also:

numpy.divide

cupy.**true_divide** = <ufunc 'cupy_true_divide'>
Elementwise true division (i.e. division as floating values).

See also:

numpy.true_divide

cupy.**floor_divide** = <ufunc 'cupy_floor_divide'>
Elementwise floor division (i.e. integer quotient).

See also:

numpy.floor_divide

cupy.**power** = <ufunc 'cupy_power'>
Computes `x1 ** x2` elementwise.

See also:

numpy.power

cupy.**fmod** = <ufunc 'cupy_fmod'>
Computes the remainder of C division elementwise.

See also:

numpy.fmod

cupy.**mod** = <ufunc 'cupy_remainder'>
Computes the remainder of Python division elementwise.

See also:

numpy.remainder

cupy.**remainder** = <ufunc 'cupy_remainder'>
Computes the remainder of Python division elementwise.

See also:

numpy.remainder

cupy.**modf** = <ufunc 'cupy_modf'>
Extracts the fractional and integral parts of an array elementwise.

This ufunc returns two arrays.

See also:

numpy.modf

cupy.**reciprocal** = <ufunc 'cupy_reciprocal'>
Computes `1 / x` elementwise.

See also:

numpy.reciprocal

Miscellaneous

cupy.**clip**(*a*, *a_min*, *a_max*, *out=None*)
Clips the values of an array to a given interval.

This is equivalent to `maximum(minimum(a, a_max), a_min)`, while this function is more efficient.

Parameters

- **a** (cupy.ndarray) – The source array.
- **a_min** (*scalar or cupy.ndarray*) – The left side of the interval.
- **a_max** (*scalar or cupy.ndarray*) – The right side of the interval.
- **out** (cupy.ndarray) – Output array.

Returns Clipped array.

Return type *cupy.ndarray*

See also:

`numpy.clip()`

cupy.**sqrt = <ufunc 'cupy_sqrt'>**
Elementwise positive square-root function.

Note: This ufunc outputs float32 arrays for float16 arrays input by default as well as NumPy 1.9. If you want to override this behavior, specify the dtype argument explicitly, or use `cupy.math.misc.sqrt_fixed` instead.

See also:

`numpy.sqrt`

cupy.**square = <ufunc 'cupy_square'>**
Elementwise square function.

See also:

`numpy.square`

cupy.**absolute = <ufunc 'cupy_absolute'>**
Elementwise absolute value function.

See also:

`numpy.absolute`

cupy.**sign = <ufunc 'cupy_sign'>**
Elementwise sign function.

It returns -1, 0, or 1 depending on the sign of the input.

See also:

`numpy.sign`

cupy.**maximum = <ufunc 'cupy_maximum'>**
Takes the maximum of two arrays elementwise.

If NaN appears, it returns the NaN.

See also:

`numpy.maximum`

cupy.**minimum** = <ufunc 'cupy_minimum'>
Takes the minimum of two arrays elementwise.

If NaN appears, it returns the NaN.

See also:

numpy.minimum

cupy.**fmax** = <ufunc 'cupy_fmax'>
Takes the maximum of two arrays elementwise.

If NaN appears, it returns the other operand.

See also:

numpy.fmax

cupy.**fmin** = <ufunc 'cupy_fmin'>
Takes the minimum of two arrays elementwise.

If NaN apperas, it returns the other operand.

See also:

numpy.fmin

3.4.9 Random Sampling (cupy.random)

CuPy's random number generation routines are based on cuRAND. They cover a small fraction of numpy.random.

The big difference of *cupy.random* from numpy.random is that *cupy.random* supports dtype option for most functions. This option enables us to generate float32 values directly without any space overhead.

Sample random data

cupy.random.**rand**(**size*, ***kwarg*)
Returns an array of uniform random values over the interval [0, 1).

Each element of the array is uniformly distributed on the half-open interval [0, 1). All elements are identically and independently distributed (i.i.d.).

Parameters

- **size** (*tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only float32 and float64 types are allowed. The default is float64.

Returns A random array.

Return type *cupy.ndarray*

See also:

numpy.random.rand()

cupy.random.**randn**(**size*, ***kwarg*)
Returns an array of standand normal random values.

Each element of the array is normally distributed with zero mean and unit variance. All elements are identically and independently distributed (i.i.d.).

Parameters

- **size** (*tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only float32 and float64 types are allowed. The default is float64.

Returns An array of standanr normal random values.

Return type *cupy.ndarray*

See also:

`numpy.random.randn()`

cupy.random.**random_sample**(*size=None*, *dtype=<type 'float'>*)
Returns an array of random values over the interval `[0, 1)`.

This is a variant of *cupy.random.rand()*.

Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only float32 and float64 types are allowed.

Returns An array of uniformly distributed random values.

Return type *cupy.ndarray*

See also:

`numpy.random.random_sample()`

cupy.random.**random**(*size=None*, *dtype=<type 'float'>*)
Returns an array of random values over the interval `[0, 1)`.

This is a variant of *cupy.random.rand()*.

Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only float32 and float64 types are allowed.

Returns An array of uniformly distributed random values.

Return type *cupy.ndarray*

See also:

`numpy.random.random_sample()`

cupy.random.**ranf**(*size=None*, *dtype=<type 'float'>*)
Returns an array of random values over the interval `[0, 1)`.

This is a variant of *cupy.random.rand()*.

Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only float32 and float64 types are allowed.

Returns An array of uniformly distributed random values.

Return type *cupy.ndarray*

See also:

`numpy.random.random_sample()`

cupy.random.**sample**(*size=None*, *dtype=<type 'float'>*)
Returns an array of random values over the interval `[0, 1)`.

This is a variant of *cupy.random.rand()*.

Parameters

- **size** (*int or tuple of ints*) – The shape of the array.
- **dtype** – Data type specifier. Only float32 and float64 types are allowed.

Returns An array of uniformly distributed random values.

Return type *cupy.ndarray*

See also:

`numpy.random.random_sample()`

Distributions

cupy.random.**lognormal**(*mean=0.0*, *sigma=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of samples drawn from a log normal distribution.

The samples are natural log of samples drawn from a normal distribution with mean `mean` and deviation `sigma`.

Parameters

- **mean** (*float*) – Mean of the normal distribution.
- **sigma** (*float*) – Standard deviation of the normal distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If None, a zero- dimensional array is generated.
- **dtype** – Data type specifier. Only float32 and float64 types are allowed.

Returns Samples drawn from the log normal distribution.

Return type *cupy.ndarray*

See also:

`numpy.random.lognormal()`

cupy.random.**normal**(*loc=0.0*, *scale=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of normally distributed samples.

Parameters

- **loc** (*float*) – Mean of the normal distribution.
- **scale** (*float*) – Standard deviation of the normal distribution.
- **size** (*int or tuple of ints*) – The shape of the array. If None, a zero- dimensional array is generated.
- **dtype** – Data type specifier. Only float32 and float64 types are allowed.

Returns Normally distributed samples.

Return type *cupy.ndarray*

See also:

`numpy.random.normal()`

cupy.random.**standard_normal**(*size=None*, *dtype=<type 'float'>*)
Returns an array of samples drawn from the standard normal distribution.

This is a variant of *cupy.random.randn()*.

Parameters

- **size** (*int or tuple of ints*) – The shape of the array. If None, a zero- dimensional array is generated.
- **dtype** – Data type specifier.

Returns Samples drawn from the standard normal distribution.

Return type *cupy.ndarray*

See also:

numpy.randomm.standard_normal()

cupy.random.**uniform**(*low=0.0*, *high=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of uniformlly-distributed samples over an interval.

Samples are drawn from a uniform distribution over the half-open interaval `[low, high)`.

Parameters

- **low** (*float*) – Lower end of the interval.
- **high** (*float*) – Upper end of the interval.
- **size** (*int or tuple of ints*) – The shape of the array. If None, a zero- dimensional array is generated.
- **dtype** – Data type specifier.

Returns Samples drawn from the uniform distribution.

Return type *cupy.ndarray*

See also:

numpy.random.uniform()

Random number generator

cupy.random.**seed**(*seed=None*)
Resets the state of the random number generator with a seed.

This function resets the state of the global random number generator for the current device. Be careful that generators for other devices are not affected.

Parameters **seed** (*None or int*) – Seed for the random number generator. If None, it uses os.urandom() if available or time.clock() otherwise. Note that this function does not support seeding by an integer array.

cupy.random.**get_random_state**()
Gets the state of the random number generator for the current device.

If the state for the current device is not created yet, this function creates a new one, initializes it, and stores it as the state for the current device.

Returns The state of the random number generator for the device.

Return type *RandomState*

class cupy.random.**RandomState**(*seed=None*, *method=100*)
Portable container of a pseudo-random number generator.

An instance of this class holds the state of a random number generator. The state is available only on the device which has been current at the initialization of the instance.

Functions of *cupy.random* use global instances of this class. Different instances are used for different devices. The global state for the current device can be obtained by the *cupy.random.get_random_state()* function.

Parameters

- **seed** (*None or int*) – Seed of the random number generator. See the *seed()* method for detail.
- **method** (*int*) – Method of the random number generator. Following values are available:

```
cupy.cuda.curand.CURAND_RNG_PSEUDO_DEFAULT
cupy.cuda.curand.CURAND_RNG_XORWOW
cupy.cuda.curand.CURAND_RNG_MRG32K3A
cupy.cuda.curand.CURAND_RNG_MTGP32
cupy.cuda.curand.CURAND_RNG_MT19937
cupy.cuda.curand.CURAND_RNG_PHILOX4_32_10
```

lognormal(*mean=0.0*, *sigma=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of samples drawn from a log normal distribution.

See also:

cupy.random.lognormal() for full documentation, numpy.random.RandomState.lognormal()

normal(*loc=0.0*, *scale=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of normally distributed samples.

See also:

cupy.random.normal() for full documentation, numpy.random.RandomState.normal()

rand(**size*, ***kwarg*)
Returns uniform random values over the interval `[0, 1)`.

See also:

cupy.random.rand() for full documentation, numpy.random.RandomState.rand()

randn(**size*, ***kwarg*)
Returns an array of standand normal random values.

See also:

cupy.random.randn() for full documentation, numpy.random.RandomState.randn()

random_sample(*size=None*, *dtype=<type 'float'>*)
Returns an array of random values over the interval `[0, 1)`.

See also:

cupy.random.random_sample() for full documentation, numpy.random.RandomState.random_sample()

seed(*seed=None*)
Resets the state of the random number generator with a seed.

..seealso:: *cupy.random.seed()* for full documentation, numpy.random.RandomState.seed()

standard_normal(*size=None*, *dtype=<type 'float'>*)
Returns samples drawn from the standard normal distribution.

See also:

cupy.random.standard_normal() for full documentation, numpy.random.RandomState.standard_norm

uniform(*low=0.0*, *high=1.0*, *size=None*, *dtype=<type 'float'>*)
Returns an array of uniformlly-distributed samples over an interval.

See also:

cupy.random.uniform() for full documentation, numpy.random.RandomState.uniform()

3.4.10 Sorting, Searching, and Counting

cupy.**argmax**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the indices of the maximum along an axis.

Parameters

- **a** (cupy.ndarray) – Array to take argmax.
- **axis** (*int*) – Along which axis to find the maximum. a is flattened by default.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis axis is preserved as an axis of length one.

Returns The indices of the maximum of a along an axis.

Return type *cupy.ndarray*

See also:

numpy.argmax()

cupy.**argmin**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the indices of the minimum along an axis.

Parameters

- **a** (cupy.ndarray) – Array to take argmin.
- **axis** (*int*) – Along which axis to find the minimum. a is flattened by default.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis axis is preserved as an axis of length one.

Returns The indices of the minimum of a along an axis.

Return type *cupy.ndarray*

See also:

numpy.argmin()

3.4.11 Statistics

Order statistics

cupy.**amin**(*a*, *axis=None*, *out=None*, *keepdims=False*, *dtype=None*)
Returns the minimum of an array or the minimum along an axis.

Parameters

- **a** (cupy.ndarray) – Array to take the minimum.
- **axis** (*int*) – Along which axis to take the minimum. The flattened array is used by default.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.
- **dtype** – Data type specifier.

Returns The minimum of a, along the axis if specified.

Return type *cupy.ndarray*

See also:

numpy.amin()

cupy.**amax**(*a*, *axis=None*, *out=None*, *keepdims=False*, *dtype=None*)
Returns the maximum of an array or the maximum along an axis.

Parameters

- **a** (cupy.ndarray) – Array to take the maximum.
- **axis** (*int*) – Along which axis to take the maximum. The flattened array is used by default.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.
- **dtype** – Data type specifier.

Returns The maximum of a, along the axis if specified.

Return type *cupy.ndarray*

See also:

numpy.amax()

Means and variances

cupy.**mean**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)
Returns the arithmetic mean along an axis.

Parameters

- **a** (cupy.ndarray) – Array to compute mean.
- **axis** (*int*) – Along which axis to compute mean. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.

Returns The mean of the input array along the axis.

Return type *cupy.ndarray*

See also:

`numpy.mean()`

cupy.**var**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the variance along an axis.

Parameters

- **a** (cupy.ndarray) – Array to compute variance.
- **axis** (*int*) – Along which axis to compute variance. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.

Returns The variance of the input array along the axis.

Return type *cupy.ndarray*

See also:

`numpy.var()`

cupy.**std**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)
Returns the standard deviation along an axis.

Parameters

- **a** (cupy.ndarray) – Array to compute standard deviation.
- **axis** (*int*) – Along which axis to compute standard deviation. The flattened array is used by default.
- **dtype** – Data type specifier.
- **out** (cupy.ndarray) – Output array.
- **keepdims** (*bool*) – If True, the axis is remained as an axis of size one.

Returns The standard deviation of the input array along the axis.

Return type *cupy.ndarray*

See also:

`numpy.std()`

3.5 NumPy-CuPy Generic Code Support

cupy.**get_array_module**(**args*)
Returns the array module for arguments.

This function is used to implement CPU/GPU generic code. If at least one of the arguments is a *cupy.ndarray* object, the *cupy* module is returned.

Parameters **args** – Values to determine whether NumPy or CuPy should be used.

Returns *cupy* or `numpy` is returned based on the types of the arguments.

Return type module

Example

A NumPy/CuPy generic function can be written as follows:

```
def softplus(x):
    xp = cupy.get_array_module(x)
    return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

3.6 Low-Level CUDA Support

3.6.1 Device management

class cupy.cuda.**Device**(*device=None*)
Object that represents a CUDA device.

This class provides some basic manipulations on CUDA devices.

It supports the context protocol. For example, the following code is an example of temporarily switching the current device:

```
with Device(0):
    do_something_on_device_0()
```

After the *with* statement gets done, the current device is reset to the original one.

Parameters **device** (*int or cupy.cuda.Device*) – Index of the device to manipulate. Be careful that the device ID (a.k.a. GPU ID) is zero origin. If it is a Device object, then its ID is used. The current device is selected by default.

id
int

ID of this device.

__eq__(*other*)
Returns True if `other` refers to the same device.

__ne__(*other*)
Returns True if `other` refers to a different device.

compute_capability
Compute capability of this device.

The capability is represented by a string containing the major index and the minor index. For example, compute capability 3.5 is represented by the string '35'.

cublas_handle
The cuBLAS handle for this device.

The same handle is used for the same device even if the Device instance itself is different.

synchronize()
Synchronizes the current thread to the device.

use()
Makes this device current.

If you want to switch a device temporarily, use the *with* statement.

3.6.2 Memory management

class cupy.cuda.**Memory**(*size*)
Memory allocation on a CUDA device.

This class provides an RAII interface of the CUDA memory allocation.

Parameters **size** (*int*) – Size of the memory allocation in bytes.

__int__()
Returns the pointer value to the head of the allocation.

device
Device whose memory the pointer refers to.

class cupy.cuda.**MemoryPointer**(*mem*, *offset*)
Pointer to a point on a device memory.

An instance of this class holds a reference to the original memory buffer and a pointer to a place within this buffer.

Parameters

- **mem** (Memory) – The device memory buffer.
- **offset** (*int*) – An offset from the head of the buffer to the place this pointer refers.

mem
Memory

The device memory buffer.

ptr
ctypes.c_void_p

Pointer to the place within the buffer.

__add__(*offset*)
Adds an offset to the pointer.

__iadd__(*offset*)
Adds an offset to the pointer in place.

__int__()
Returns the pointer value.

__isub__(*offset*)
Subtracts an offset from the pointer in place.

__radd__(*offset*)
Adds an offset to the pointer.

__sub__(*offset*)
Subtracts an offset from the pointer.

copy_from(*mem*, *size*)
Copies a memory sequence from a (possibly different) device or host.

This function is a useful interface that selects appropriate one from *copy_from_device()* and *copy_from_host()*.

Parameters

- **mem** (*ctypes.c_void_p or cupy.cuda.MemoryPointer*) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

copy_from_async(*mem*, *size*, *stream*)
Copies a memory sequence from an arbitrary place asynchronously.

This function is a useful interface that selects appropriate one from *copy_from_device_async()* and *copy_from_host_async()*.

Parameters

- **mem** (*ctypes.c_void_p or cupy.cuda.MemoryPointer*) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (cupy.cuda.Stream) – CUDA stream.

copy_from_device(*src*, *size*)
Copies a memory sequence from a (possibly different) device.

Parameters

- **src** (cupy.cuda.MemoryPointer) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

copy_from_device_async(*src*, *size*, *stream*)
Copies a memory sequence from a (possibly different) device asynchronously.

Parameters

- **src** (cupy.cuda.MemoryPointer) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (cupy.cuda.Stream) – CUDA stream.

copy_from_host(*mem*, *size*)
Copies a memory sequence from the host memory.

Parameters

- **mem** (*ctypes.c_void_p*) – Source memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

copy_from_host_async(*mem*, *size*, *stream*)
Copies a memory sequence from the host memory asynchronously.

Parameters

- **src** (*ctypes.c_void_p*) – Source memory pointer. It must be a pinned memory.
- **size** (*int*) – Size of the sequence in bytes.

copy_to_host(*mem*, *size*)
Copies a memory sequence to the host memory.

Parameters

- **mem** (*ctypes.c_void_p*) – Target memory pointer.
- **size** (*int*) – Size of the sequence in bytes.

copy_to_host_async(*mem*, *size*, *stream*)
Copies a memory sequence to the host memory asynchronously.

Parameters

- **mem** (*ctypes.c_void_p*) – Target memory pointer. It must be a pinned memory.
- **size** (*int*) – Size of the sequence in bytes.

- **stream** (cupy.cuda.Stream) – CUDA stream.

device
Device whose memory the pointer refers to.

memset(*value*, *size*)
Fills a memory sequence by constant byte value.

Parameters

- **value** (*int*) – Value to fill.
- **size** (*int*) – Size of the sequence in bytes.

memset_async(*value*, *size*, *stream*)
Fills a memory sequence by constant byte value asynchronously.

Parameters

- **value** (*int*) – Value to fill.
- **size** (*int*) – Size of the sequence in bytes.
- **stream** (cupy.cuda.Stream) – CUDA stream.

cupy.cuda.**alloc**(*size*)
Calls the current allocator.

Use *set_allocator()* to change the current allocator.

Parameters **size** (*int*) – Size of the memory allocation.

Returns Pointer to the allocated buffer.

Return type *MemoryPointer*

cupy.cuda.**set_allocator**(*allocator=<function _malloc>*)
Sets the current allocator.

Parameters **allocator** (*function*) – CuPy memory allocator. It must have the same interface as the *cupy.cuda.alloc()* function, which takes the buffer size as an argument and returns the device buffer of that size.

class cupy.cuda.**MemoryPool**(*allocator=<function _malloc>*)
Memory pool for all devices on the machine.

A memory pool preserves any allocations even if they are freed by the user. Freed memory buffers are held by the memory pool as *free blocks*, and they are reused for further memory allocations of the same sizes. The allocated blocks are managed for each device, so one instance of this class can be used for multiple devices.

Note: When the allocation is skipped by reusing the pre-allocated block, it does not call cudaMalloc and therefore CPU-GPU synchronization does not occur. It makes interleaves of memory allocations and kernel invocations very fast.

Note: The memory pool holds allocated blocks without freeing as much as possible. It makes the program hold most of the device memory, which may make other CUDA programs running in parallel out-of-memory situation.

Parameters **allocator** (*function*) – The base CuPy memory allocator. It is used for allocating new blocks when the blocks of the required size are all in use.

malloc(*size*)
Allocates the memory, from the pool if possible.

This method can be used as a CuPy memory allocator. The simplest way to use a memory pool as the default allocator is the following code:

```
set_allocator(MemoryPool().malloc)
```

Parameters **size** (*int*) – Size of the memory buffer to allocate in bytes.

Returns Pointer to the allocated buffer.

Return type *MemoryPointer*

3.6.3 Streams and events

class cupy.cuda.**Stream**(*null=False*, *non_blocking=False*)
CUDA stream.

This class handles the CUDA stream handle in RAII way, i.e., when an Stream instance is destroyed by the GC, its handle is also destroyed.

Parameters

- **null** (*bool*) – If True, the stream is a null stream (i.e. the default stream that synchronizes with all streams). Otherwise, a plain new stream is created.
- **non_blocking** (*bool*) – If True, the stream does not synchronize with the NULL stream.

ptr
cupy.cuda.runtime.Stream

Raw stream handle. It can be passed to the CUDA Runtime API via ctypes.

add_callback(*callback*, *arg*)
Adds a callback that is called when all queued work is done.

Parameters

- **callback** (*function*) – Callback function. It must take three arguments (Stream object, int error status, and user data of type ctypes.c_void_p), and returns nothing.
- **arg** (*ctypes.c_void_p*) – Argument to the callback.

done
True if all work on this stream has been done.

record(*event=None*)
Records an event on the stream.

Parameters **event** (*None or cupy.cuda.Event*) – CUDA event. If None, then a new plain event is created and used.

Returns The recorded event.

Return type *cupy.cuda.Event*

See also:

cupy.cuda.Event.record()

synchronize()
Waits for the stream completing all queued work.

wait_event(*event*)
Makes the stream wait for an event.

The future work on this stream will be done after the event.

Parameters **event** (cupy.cuda.Event) – CUDA event.

class cupy.cuda.**Event**(*block=False*, *disable_timing=False*, *interprocess=False*)
CUDA event, a synchronization point of CUDA streams.

This class handles the CUDA event handle in RAII way, i.e., when an Event instance is destroyed by the GC, its handle is also destroyed.

Parameters

- **block** (*bool*) – If True, the event blocks on the *synchronize()* method.
- **disable_timing** (*bool*) – If True, the event does not prepare the timing data.
- **interprocess** (*bool*) – If True, the event can be passed to other processes.

ptr
cupy.cuda.runtime.Stream

Raw stream handle. It can be passed to the CUDA Runtime API via ctypes.

done
True if the event is done.

record(*stream=None*)
Records the event to a stream.

Parameters **stream** (cupy.cuda.Stream) – CUDA stream to record event. The null stream is used by default.

See also:

cupy.cuda.Stream.record()

synchronize()
Synchronizes all device work to the event.

If the event is created as a blocking event, it also blocks the CPU thread until the event is done.

cupy.cuda.**get_elapsed_time**(*start_event*, *end_event*)
Gets the elapsed time between two events.

Parameters

- **start_event** (Event) – Earlier event.
- **end_event** (Event) – Later event.

Returns Elapsed time in milliseconds.

Return type float

3.7 Kernel binary memoization

cupy.**memoize**(*for_each_device=False*)
Makes a function memoizing the result for each argument and device.

This decorator provides automatic memoization of the function result.

Parameters **for_each_device** (*bool*) – If True, it memoizes the results for each device. Otherwise, it memoizes the results only based on the arguments.

cupy.**clear_memo**()
Clears the memoized results for all functions decorated by memoize.

3.8 User-Defined Kernels

CuPy provides easy ways to define two types of CUDA kernels: elementwise kernels and reduction kernels. We first describe how to define and call elementwise kernels, and then describe how to define and call reduction kernels.

3.8.1 Basics of elementwise kernels

An elementwise kernel can be defined by the *ElementwiseKernel* class. The instance of this class defines a CUDA kernel which can be invoked by the __call__ method of this instance.

A definition of an elementwise kernel consists of four parts: an input argument list, an output argument list, a loop body code, and the kernel name. For example, a kernel that computes a squared difference $f(x, y) = (x - y)^2$ is defined as follows:

```
>>> squared_diff = cupy.ElementwiseKernel(
...     'float32 x, float32 y',
...     'float32 z',
...     'z = (x - y) * (x - y)',
...     'squared_diff')
```

The argument lists consist of comma-separated argument definitions. Each argument definition consists of a *type specifier* and an *argument name*. Names of NumPy data types can be used as type specifiers.

Note: n, i, and names starting with an underscore _ are reserved for the internal use.

The above kernel can be called on either scalars or arrays with broadcasting:

```
>>> x = cupy.arange(10, dtype=np.float32).reshape(2, 5)
>>> y = cupy.arange(5, dtype=np.float32)
>>> squared_diff(x, y)
array([[  0.,   0.,   0.,   0.,   0.],
       [ 25.,  25.,  25.,  25.,  25.]], dtype=float32)
>>> squared_diff(x, 5)
array([[ 25.,  16.,   9.,   4.,   1.],
       [  0.,   1.,   4.,   9.,  16.]], dtype=float32)
```

Output arguments can be explicitly specified (next to the input arguments):

```
>>> z = cupy.empty((2, 5), dtype=np.float32)
>>> squared_diff(x, y, z)
array([[  0.,   0.,   0.,   0.,   0.],
       [ 25.,  25.,  25.,  25.,  25.]], dtype=float32)
```

3.8.2 Type-generic kernels

If a type specifier is one character, then it is treated as a **type placeholder**. It can be used to define a type-generic kernels. For example, the above squared_diff kernel can be made type-generic as follows:

```
>>> squared_diff_generic = cupy.ElementwiseKernel(
...     'T x, T y',
...     'T z',
...     'z = (x - y) * (x - y)',
...     'squared_diff_generic')
```

Type placeholders of a same character in the kernel definition indicate the same type. The actual type of these placeholders is determined by the actual argument type. The ElementwiseKernel class first checks the output arguments and then the input arguments to determine the actual type. If no output arguments are given on the kernel invocation, then only the input arguments are used to determine the type.

The type placeholder can be used in the loop body code:

```
>>> squared_diff_generic = cupy.ElementwiseKernel(
...     'T x, T y',
...     'T z',
...     '''
...         T diff = x - y;
...         z = diff * diff;
...     ''',
...     'squared_diff_generic')
```

More than one type placeholder can be used in a kernel definition. For example, the above kernel can be further made generic over multiple arguments:

```
>>> squared_diff_super_generic = cupy.ElementwiseKernel(
...     'X x, Y y',
...     'Z z',
...     'z = (x - y) * (x - y)',
...     'squared_diff_super_generic')
```

Note that this kernel requires the output argument explicitly specified, because the type `Z` cannot be automatically determined from the input arguments.

3.8.3 Raw argument specifiers

The ElementwiseKernel class does the indexing with broadcasting automatically, which is useful to define most elementwise computations. On the other hand, we sometimes want to write a kernel with manual indexing for some arguments. We can tell the ElementwiseKernel class to use manual indexing by adding the `raw` keyword preceding the type specifier.

We can use the special variables `n` and `i` for the manual indexing. `n` indicates total number of elements to apply the elementwise operation. `i` indicates the index within the loop. For example, a kernel that adds two vectors with reversing one of them can be written as follows:

```
>>> add_reverse = cupy.ElementwiseKernel(
...     'T x, raw T y', 'T z',
...     'z = x + y[n - i - 1]',
...     'add_reverse')
```

(Note that this is an artificial example and you can write such operation just by `z = x + y[::-1]` without defining a new kernel). A raw argument can be used like an array. The indexing operator `y[n - i]` involves an indexing computation on `y`, so `y` can be arbitrarily shaped and strided.

Note that raw arguments are not involved in the broadcasting and the determination of `n`. If you want to mark all arguments as `raw`, you must specify the `size` argument on invocation, which defines the value of `n`.

3.8.4 Reduction kernels

Reduction kernels can be defined by the *ReductionKernel* class. We can use it by defining four parts of the kernel code:

1. Identity value: This value is used for the initial value of reduction.
2. Mapping expression: It is used for the preprocessing of each element to be reduced.
3. Reduction expression: It is an operator to reduce the multiple mapped values. The special variables `a` and `b` are used for its operands.
4. Post mapping expression: It is used to transform the resulting reduced values. The special variable `a` is used as its input. Output should be written to the output parameter.

ReductionKernel class automatically inserts other code fragments that are required for an efficient and flexible reduction implementation.

For example, L2 norm along specified axes can be written as follows:

```
>>> l2norm_kernel = cupy.ReductionKernel(
...     'T x',  # input params
...     'T y',  # output params
...     'x * x',  # map
...     'a + b',  # reduce
...     'y = sqrt(a)',  # post-reduction map
...     '0',  # identity value
...     'l2norm'  # kernel name
... )
>>> x = cupy.arange(10, dtype='f').reshape(2, 5)
>>> l2norm_kernel(x, axis=1)
array([  5.47722578,  15.96871948], dtype=float32)
```

Note: `raw` specifier is restricted for usages that the axes to be reduced are put at the head of the shape. It means, if you want to use `raw` specifier for at least one argument, the `axis` argument must be `0` or a contiguous increasing sequence of integers starting from `0`, like `(0, 1)`, `(0, 1, 2)`, etc.

3.8.5 Reference

class `cupy.`**`ElementwiseKernel`**(*in_params*, *out_params*, *operation*, *name='kernel'*, *reduce_dims=True*, *preamble=''*, ***kwargs*)

User-defined elementwise kernel.

This class can be used to define an elementwise kernel with or without broadcasting.

The kernel is compiled at an invocation of the *__call__()* method, which is cached for each device. The compiled binary is also cached into a file under the `$HOME/.cupy/kernel_cache/` directory with a hashed file name. The cached binary is reused by other processes.

Parameters

- **`in_params`** (*str*) – Input argument list.
- **`out_params`** (*str*) – Output argument list.
- **`operation`** (*str*) – The body in the loop written in CUDA-C/C++.
- **`name`** (*str*) – Name of the kernel function. It should be set for readability of the performance profiling.

- **reduce_dims** (*bool*) – If False, the shapes of array arguments are kept within the kernel invocation. The shapes are reduced (i.e., the arrays are reshaped without copy to the minimum ndims) by default. It may make the kernel fast by reducing the index calculations.
- **options** (*list*) – Options passed to the nvcc command.
- **preamble** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the cu file.
- **loop_prep** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the kernel function definition and above the `for` loop.
- **after_loop** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the bottom of the kernel function definition.

__call__(**args*, ***kwargs*)

Compiles and invokes the elementwise kernel.

The compilation runs only if the kernel is not cached. Note that the kernels with different argument dtypes or ndims are not compatible. It means that single ElementwiseKernel object may be compiled into multiple kernel binaries.

Parameters

- **args** – Argumens of the kernel.
- **size** (*int*) – Range size of the indices. If specified, the variable `n` is set to this value. Otherwise, the result of broadcasting is used to determine the value of `n`.

Returns Arrays are returned according to the `out_params` argument of the `__init__` method.

class cupy.**ReductionKernel**(*in_params*, *out_params*, *map_expr*, *reduce_expr*, *post_map_expr*, *identity*, *name='reduce_kernel'*, *reduce_type=None*, *reduce_dims=True*, *preamble=''*, *options=()*)

User-defined reduction kernel.

This class can be used to define a reduction kernel with or without broadcasting.

The kernel is compiled at an invocation of the *__call__()* method, which is cached for each device. The compiled binary is also cached into a file under the `$HOME/.cupy/kernel_cache/` directory with a hashed file name. The cached binary is resued by other processes.

Parameters

- **in_params** (*str*) – Input argument list.
- **out_params** (*str*) – Output argument list.
- **map_expr** (*str*) – Mapping expression for input values.
- **reduce_expr** (*str*) – Reduction expression.
- **post_map_expr** (*str*) – Mapping expression for reduced values.
- **identity** (*str*) – Identity value for starting the reduction.
- **name** (*str*) – Name of the kernel function. It should be set for readability of the performance profiling.
- **reduce_type** (*str*) – Type of values to be used for reduction. This type is used to store the special variables `a`.
- **reduce_dims** (*bool*) – If True, input arrays are reshaped without copy to smaller dimensions for efficiency.

- **preamble** (*str*) – Fragment of the CUDA-C/C++ code that is inserted at the top of the cu file.
- **options** (*tuple of str*) – Additional compilation options.

__call__ (**args*, ***kwargs*)

Compiles and invokes the reduction kernel.

The compilation runs only if the kernel is not cached. Note that the kernels with different argument dtypes, ndims, or axis are not compatible. It means that single ReductionKernel object may be compiled into multiple kernel binaries.

Parameters **args** – Arguments of the kernel.

Returns Arrays are returned according to the `out_params` argument of the `__init__` method.

CHAPTER 4

Chainer Contribution Guide

This is a guide for all contributions to Chainer. The development of Chainer is running on the official repository at GitHub. Anyone that wants to register an issue or to send a pull request should read through this document.

4.1 Classification of Contributions

There are several ways to contribute to Chainer community:

1. Registering an issue
2. Sending a pull request (PR)
3. Sending a question to Chainer User Group
4. Open-sourcing an external example
5. Writing a post about Chainer

This document mainly focuses on 1 and 2, though other contributions are also appreciated.

4.2 Release and Milestone

We are using GitHub Flow as our basic working process. In particular, we are using the master branch for our development, and releases are made as tags.

Releases are classified into three groups: major, minor, and revision. This classification is based on following criteria:

- A **major** release contains catastrophic changes on the interface that may break existing user codes.
- A **minor** release contains additions and modifications on the interface. It may break some existing user codes, though they must be fixed by small efforts.
- A **revision** release contains changes that does not affect the documented interface. It mainly consists of bug fixes, implementation improvements, and test/document/example updates.

The release classification is reflected into the version number x.y.z, where x, y, and z corresponds to major, minor, and revision updates, respectively.

We sets milestones for some future releases. A milestone for a revision release is set right after the last release. On the other hand, a milestone for a minor or major release is set four weeks prior to its due.

4.3 Issues and PRs

Issues and PRs are classified into following categories:

- **Bug**: bug reports (isseus) and bug fixes (PRs)
- **Enhancement**: implementation improvements without breaking the interface
- **Feature**: feature requests (issues) and their implementations (PRs)
- **Test**: test fixes and updates
- **Document**: document fixes and improvements
- **Example**: fixes and improvements on the examples
- **Other**: other issues and PRs

Issues and PRs are labeled by these categories. This classification is often reflected into its corresponding release category: Feature issues/PRs are contained into minor/major releases, while other issues/PRs can be contained into any releases including revision ones.

On registering an issue, write precise explanations on what you want Chainer to be. Bug reports must include necessary and sufficient conditions to reproduce the bugs. Feature requests must include **what** you want to do (and **why** you want to do, if needed). You can contain your thoughts on **how** to realize it into the feature requests, though **what** part is most important for discussions.

> **Warning:** If you have a question on usages of Chainer, it is highly recommended to send a post to Chainer User Group instead of the issue tracker. The issue tracker is not a place to share knowledge on practices. We may redirect question issues to Chainer User Group.

If you can write codes to fix an issue, send a PR to the master branch. Before writing your codes for PRs, read through the *Coding Guidelines*. The description of any PR must contain a precise explanation of **what** and **how** you want to do; it is the first documentation of your codes for developers, a very important part of your PR.

Once you send a PR, it is automatically tested on Travis CI. After the automatic test passes, some of the core developers will start reviewing your codes. Note that this automatic PR test only includes CPU tests.

Note: We are also running continuous integrations with GPU tests for the master branch. Since this service is running on our internal server, we do not use it for automatic PR tests to keep the server secure.

Even if your codes are not complete, you can send a pull request as a *work-in-progress PR* by putting the `[WIP]` prefix to the PR title. If you write a precise explanation about the PR, core developers and other contributors can join the discussion about how to proceed the PR.

4.4 Coding Guidelines

We use PEP8 and a part of OpenStack Style Guidelines related to general coding style as our basic style guidelines.

Before checking your code, you can use automatic formatter to set appropriate spacing, etc. We recommend you to install the `pyformat` and `isort` packages, and run the following commands:

```
$ pyformat -i path/to/your/code.py
$ isort path/to/your/code.py
```

Note that these formatters do not cover all part of the style guidelines.

To check your code, use `flake8` command installed by `hacking` package:

```
$ pip install hacking
$ flake8 path/to/your/code.py
```

The `flake8` command lets you know the part of your code not obeying our style guidelines. Before sending a pull request, be sure to check that your code passes the `flake8` checking.

Note that `flake8` command is not perfect. It does not check some of the style guidelines. Here is a (not-complete) list of the rules that `flake8` cannot check.

- Relative imports are prohibited. [H304]
- Importing non-module symbols is prohibited.
- Import statements must be organized into three parts: standard libraries, third-party libraries, and internal imports. [H306]

In addition, we restrict the usage of *shortcut symbols* in our code base. They are symbols imported by packages and subpackages of `chainer`. For example, `chainer.Variable` is a shortcut of `chainer.variable.Variable`. **It is not allowed to use such shortcuts in the ‘‘chainer‘‘ library implementation**. Note that you can still use them in `tests` and `examples` directories.

Once you send a pull request, your coding style is automatically checked by Travis-CI. The reviewing process starts after the check passes.

4.5 Testing Guidelines

Testing is one of the most important part of your code. You must test your code by unit tests following our testing guidelines.

We are using `nose` package to run unit tests. You can run unit tests simply by running `nosetests` command under the repository root. It requires CUDA by default. In order to run unit tests that do not require CUDA, pass `--attr='!gpu'` option to the nosetests command:

```
$ nosetests path/to/your/test.py --attr='!gpu'
```

Tests are put into the `tests` directory. This directory has the same structure as the `chainer` directory. In order to enable test runner to find test scripts correctly, we are using special naming convention for the test subdirectories and the test scripts.

- The name of each subdirectory of `tests` must end with the `_tests` suffix.
- The name of each test script must start with the `test_` prefix.

Following this naming convention, you can run all the tests by just typing `nosetests` at the repository root:

```
$ nosetests
```

If you modify the code related to existing unit tests, you must run this command.

There are many examples of unit tests under the `tests` directory. They simply use the `unittest` package of the standard library.

If your patch includes GPU-related code, your tests must run with and without GPU capability. Test functions that requires CUDA must be tagged by the `chainer.testing.attr.gpu` decorator:

```
import unittest
from chainer.testing import attr

class TestMyFunc(unittest.TestCase):
    ...
```

```
    @attr.gpu
    def test_my_gpu_func(self):
        ...
```

The functions tagged by the `chainer.testing.attr.gpu` decorator are skipped if `--attr='!gpu'` is given. We also have the `chainer.testing.attr.cudnn` decorator to let nosetests know that the test depends on CuDNN.

Once you send a pull request, your code is automatically tested by Travis-CI **with –attr='!gpu' option**. Since Travis-CI does not support CUDA, we cannot check you CUDA-related code automatically. The reviewing process starts after the test passes. Note that reviewers will test your code without the option to check CUDA-related code.

CHAPTER 5

Tips and FAQs

5.1 It takes too long time to compile a computational graph. Can I skip it?

Chainer does not compile computational graphs, so you cannot skip it, or, I mean, you have already skipped it :).

It seems you have actually seen on-the-fly compilations of CUDA kernels. CuPy compiles kernels on demand to make kernels optimized to the number of dimensions and element types of input arguments. Precompilation is not available, because we have to compile an exponential number of kernels to support all CuPy functionalities. This restriction is unavoidable because Python cannot call CUDA/C++ template functions in generic way. Note that every framework using CUDA require compilation at some point; the difference between other statically-compiled frameworks (such as cutorch) and Chainer is whether a kernel is compiled at installtion or at the first use.

These compilations should run only at the first use of the kernels. The compiled binaries are cached to the `$(HOME)/.cupy/kernel_cache` directory by default. If you see that compilations run everytime you run the same script, then the caching is failed. Please check that the directory is kept as is between multiple executions of the script. If your home directory is not suited to caching the kernels (e.g. in case that it uses NFS), change the kernel caching directory by setting the `CUPY_CACHE_DIR` environment variable to an appropriate path. See *CuPy Overview* for more details.

CHAPTER 6

Comparison with Other Frameworks

6.1 A table for quick comparison

This table compares Chainer with other popular deep learning frameworks. We hope it helps you to choose an appropriate framework for the demand.

Note: This chart may be out-dated, since the developers of Chainer do not perfectly follow the latest development status of each framework. Please report us if you find an out-dated cell. Requests for new comparison axes are also welcome.

		Chainer	Theano-based	Torch7	Caffe
Specs	Scripting	Python	Python	LuaJIT	Python
	Net definition language	Python	Python	LuaJIT	Protocol Buffers
	Define-by-Run scheme	Y			
	CPU Array backend	NumPy	NumPy	Tensor	
	GPU Array backend	PyCUDA [1]	CudaNdarray [2]	CudaTensor	
NNs	Reverse-mode AD	Y	Y	Y	Y
	Basic RNN support	Y	Y	Y (nnx)	#2033
	Variable-length loops	Y	Y (scan)		
	Stateful RNNs [3]	Y		Y [4]	
	Per-batch architectures	Y			
Perf	CUDA support	Y	Y	Y	Y
	cuDNN support	Y	Y	Y (cudnn.torch)	Y
	FFT-based convolution		Y	Y (fbcunn)	#544
	CPU/GPU generic coding [5]	[1]	[6]	Y	
	Multi GPU (data parallel)	Y		Y (fbcunn)	Y
	Multi GPU (model parallel)	Y		Y (fbcunn)	
Misc	Type checking	Y	Y	Y	N/A
	Model serialization	Y (pickle)	Y (pickle)	Y	Y
	Caffe reference model	Y	[7]	Y (loadcaffe)	Y

[1] We are preparing for changing the GPU array backend to CuPy. It enables us to write one code for both CPU and GPU arrays.
[2] They are also developing libgpuarray
[3] Stateful RNN is a type of RNN implementation that maintains states in the loops. It should enable us to use the states arbitrarily to update

6.2 Benchmarks

We are preparing for the benchmarks.

them.

[4] Also available in the *Torch RNN package <https://github.com/Element-Research/rnn>*

[5] This row shows whether each array API supports unified codes for CPU and GPU.

[6] The array backend of Theano does not have compatible interface with NumPy, though most users write code on theano variables, which is generic for CPU and GPU.

[7] Depending on the frameworks.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

C

Symbols

A

B

C

D

E

F

G

H

I

L

M

T

U

V

W

Z