PYTORCH TUTORIAL

Xiao Zhang

Advantage:

1. Super flexible, if you know numpy, you can write Pytorch code

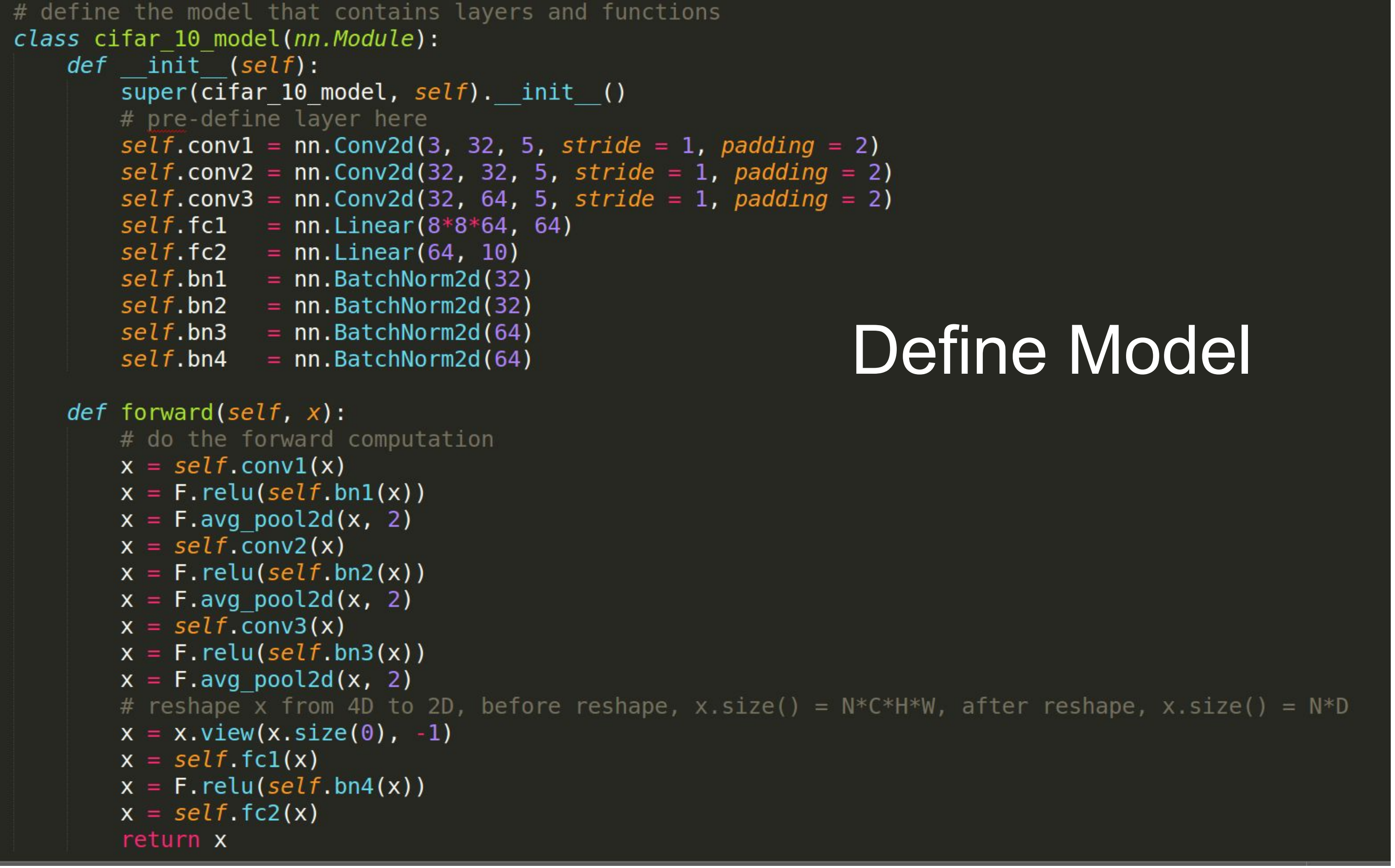
2. Dynamic Graph, you can set breakpoint or using PDB to debug and immediately print out the intermediate result just like numpy (No need to run additional session)

3. Highly concise API. The API document is well organized and it’s intuitive to understand

4. Easy to build complex network structure

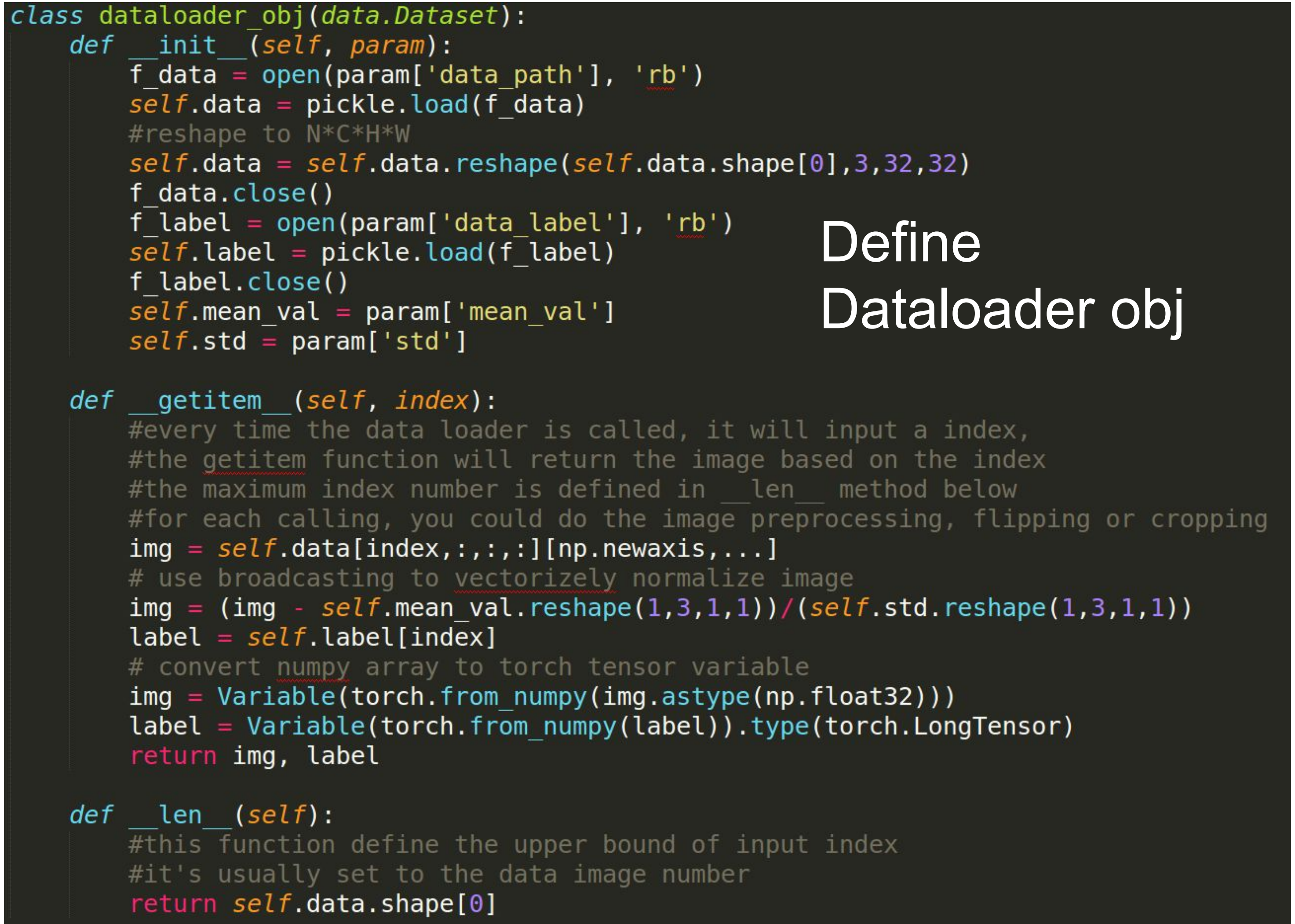
Writing Pytorch code, just needs 3 step:

1. Define model (initialized layer)
2. Define Dataloader obj(define how the data will be loaded)
3. Write training or testing code

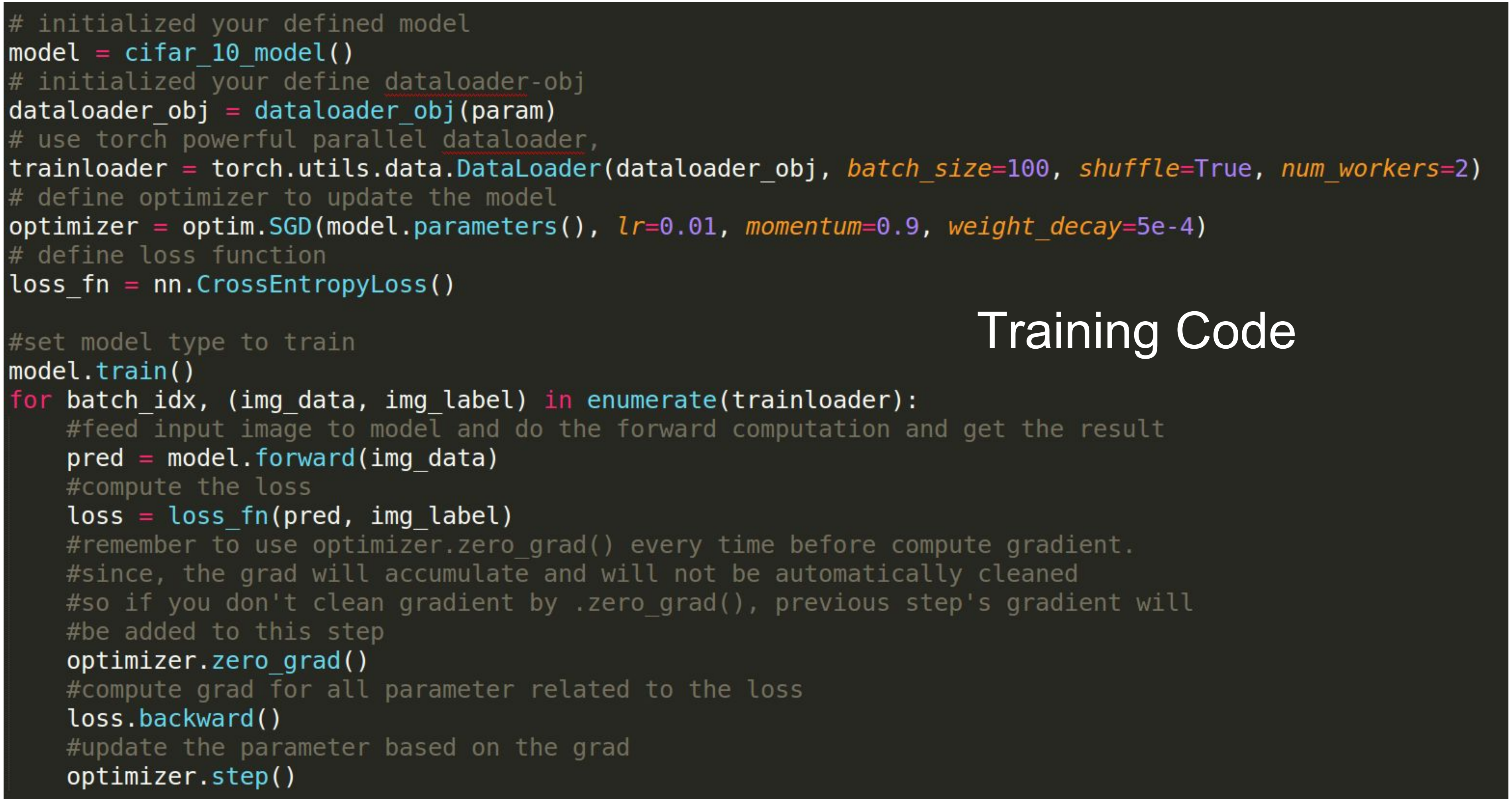

```python
# define the model that contains layers and functions
class cifar_10_model(nn.Module):
    def __init__(self):
        super(cifar_10_model, self).__init__()
        # pre-define layer here
        self.conv1 = nn.Conv2d(3, 32, 5, stride = 1, padding = 2)
        self.conv2 = nn.Conv2d(32, 32, 5, stride = 1, padding = 2)
        self.conv3 = nn.Conv2d(32, 64, 5, stride = 1, padding = 2)
        self.fc1   = nn.Linear(8*8*64, 64)
        self.fc2   = nn.Linear(64, 10)
        self.bn1   = nn.BatchNorm2d(32)
        self.bn2   = nn.BatchNorm2d(32)
        self.bn3   = nn.BatchNorm2d(64)
        self.bn4   = nn.BatchNorm2d(64)

    def forward(self, x):
        # do the forward computation
        x = self.conv1(x)
        x = F.relu(self.bn1(x))
        x = F.avg_pool2d(x, 2)
        x = self.conv2(x)
        x = F.relu(self.bn2(x))
        x = F.avg_pool2d(x, 2)
        x = self.conv3(x)
        x = F.relu(self.bn3(x))
        x = F.avg_pool2d(x, 2)
        # reshape x from 4D to 2D, before reshape, x.size() = N*C*H*W, after reshape, x.size() = N*D
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.relu(self.bn4(x))
        x = self.fc2(x)
        return x
```

Define Model

Define Dataloader obj

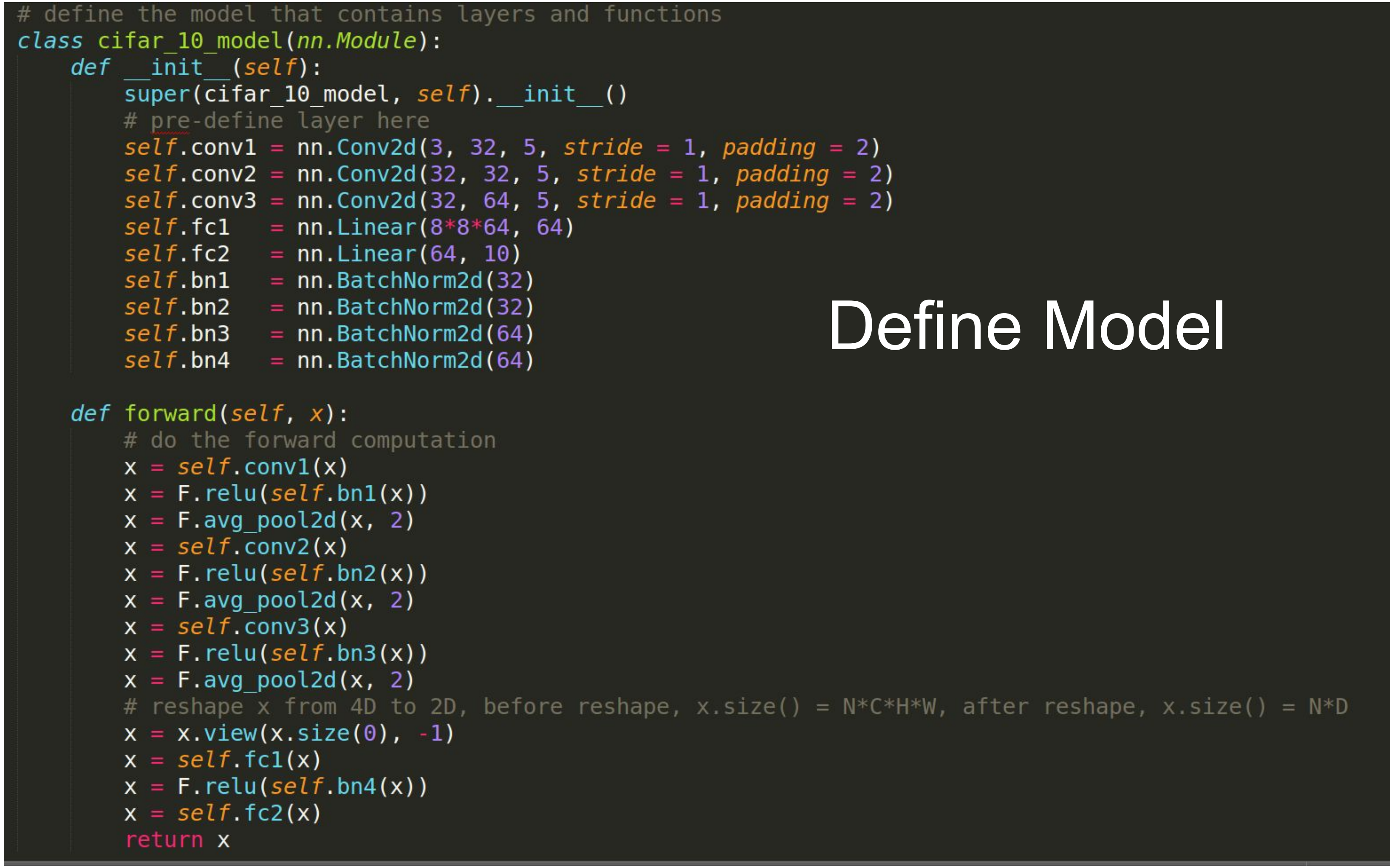


```python
class dataloader_obj(data.Dataset):
    def __init__(self, param):
        f_data = open(param['data_path'], 'rb')
        self.data = pickle.load(f_data)
        #reshape to N*C*H*W
        self.data = self.data.reshape(self.data.shape[0],3,32,32)
        f_data.close()
        f_label = open(param['data_label'], 'rb')
        self.label = pickle.load(f_label)
        f_label.close()
        self.mean_val = param['mean_val']
        self.std = param['std']

    def __getitem__(self, index):
        #every time the data loader is called, it will input a index,
        #the getitem function will return the image based on the index
        #the maximum index number is defined in __len__ method below
        #for each calling, you could do the image preprocessing, flipping or cropping
        img = self.data[index,:,:,:][np.newaxis,...]
        # use broadcasting to vectorizely normalize image
        img = (img - self.mean_val.reshape(1,3,1,1))/(self.std.reshape(1,3,1,1))
        label = self.label[index]
        # convert numpy array to torch tensor variable
        img = Variable(torch.from_numpy(img.astype(np.float32)))
        label = Variable(torch.from_numpy(label)).type(torch.LongTensor)
        return img, label

    def __len__(self):
        #this function define the upper bound of input index
        #it's usually set to the data image number
        return self.data.shape[0]
```



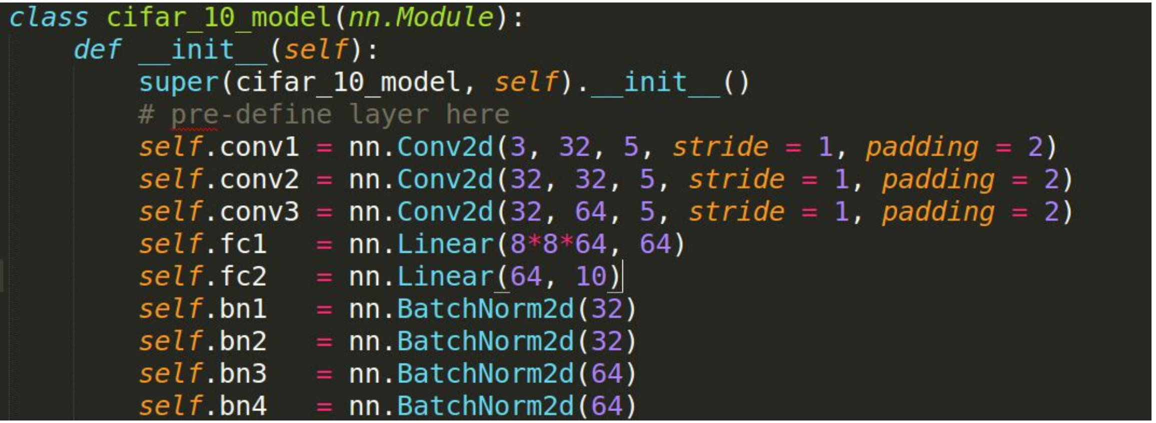
```
# initialized your defined model
model = cifar_10_model()
# initialized your define dataloader-obj
dataloader_obj = dataloader_obj(param)
# use torch powerful parallel dataloader,
trainloader = torch.utils.data.DataLoader(dataloader_obj, batch_size=100, shuffle=True, num_workers=2)
# define optimizer to update the model
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
# define loss function
loss_fn = nn.CrossEntropyLoss()

#set model type to train
model.train()
for batch_idx, (img_data, img_label) in enumerate(trainloader):
    #feed input image to model and do the forward computation and get the result
    pred = model.forward(img_data)
    #compute the loss
    loss = loss_fn(pred, img_label)
    #remember to use optimizer.zero_grad() every time before compute gradient.
    #since, the grad will accumulate and will not be automatically cleaned
    #so if you don't clean gradient by .zero_grad(), previous step's gradient will
    #be added to this step
    optimizer.zero_grad()
    #compute grad for all parameter related to the loss
    loss.backward()
    #update the parameter based on the grad
    optimizer.step()
```

Training Code



```
# define the model that contains layers and functions
class cifar_10_model(nn.Module):
    def __init__(self):
        super(cifar_10_model, self).__init__()
        # pre-define layer here
        self.conv1 = nn.Conv2d(3, 32, 5, stride = 1, padding = 2)
        self.conv2 = nn.Conv2d(32, 32, 5, stride = 1, padding = 2)
        self.conv3 = nn.Conv2d(32, 64, 5, stride = 1, padding = 2)
        self.fc1   = nn.Linear(8*8*64, 64)
        self.fc2   = nn.Linear(64, 10)
        self.bn1   = nn.BatchNorm2d(32)
        self.bn2   = nn.BatchNorm2d(32)
        self.bn3   = nn.BatchNorm2d(64)
        self.bn4   = nn.BatchNorm2d(64)

    def forward(self, x):
        # do the forward computation
        x = self.conv1(x)
        x = F.relu(self.bn1(x))
        x = F.avg_pool2d(x, 2)
        x = self.conv2(x)
        x = F.relu(self.bn2(x))
        x = F.avg_pool2d(x, 2)
        x = self.conv3(x)
        x = F.relu(self.bn3(x))
        x = F.avg_pool2d(x, 2)
        # reshape x from 4D to 2D, before reshape, x.size() = N*C*H*W, after reshape, x.size() = N*D
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.relu(self.bn4(x))
        x = self.fc2(x)
        return x
```

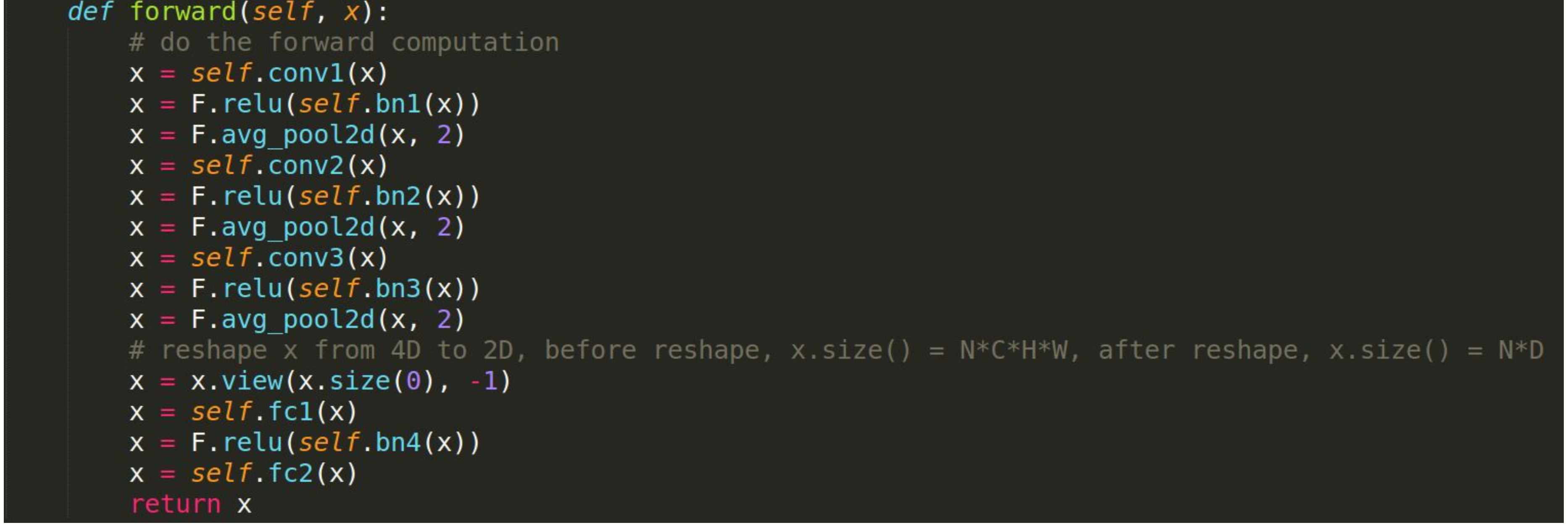
Define Model



```
class cifar_10_model(nn.Module):
    def __init__(self):
        super(cifar_10_model, self).__init__()
        # pre-define layer here
        self.conv1 = nn.Conv2d(3, 32, 5, stride = 1, padding = 2)
        self.conv2 = nn.Conv2d(32, 32, 5, stride = 1, padding = 2)
        self.conv3 = nn.Conv2d(32, 64, 5, stride = 1, padding = 2)
        self.fc1   = nn.Linear(8*8*64, 64)
        self.fc2   = nn.Linear(64, 10)
        self.bn1   = nn.BatchNorm2d(32)
        self.bn2   = nn.BatchNorm2d(32)
        self.bn3   = nn.BatchNorm2d(64)
        self.bn4   = nn.BatchNorm2d(64)
```

In pytorch, model is just a wrapper to store the network layer, the layer’s parameter and do forward computation

Basically, you need to pre-define all the layer that may include trainable parameters you may use like **convolution, linear(fully connected), batchnorm,** when initializing model.

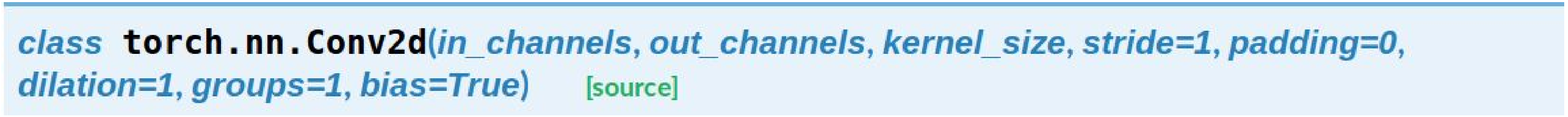

```
    def forward(self, x):
        # do the forward computation
        x = self.conv1(x)
        x = F.relu(self.bn1(x))
        x = F.avg_pool2d(x, 2)
        x = self.conv2(x)
        x = F.relu(self.bn2(x))
        x = F.avg_pool2d(x, 2)
        x = self.conv3(x)
        x = F.relu(self.bn3(x))
        x = F.avg_pool2d(x, 2)
        # reshape x from 4D to 2D, before reshape, x.size() = N*C*H*W, after reshape, x.size() = N*D
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.relu(self.bn4(x))
        x = self.fc2(x)
        return x
```

After define the layer, you need to define the forward computation with these layer and also, the additional function if needed

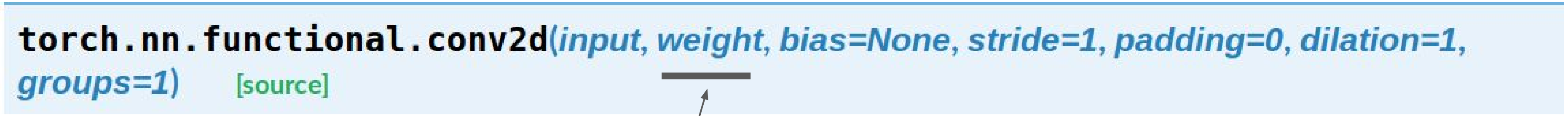
Pytorch has too type of network API:

torch.nn and **torch.nn.functional**

Each type of layer has the equivalent function both in torch.nn and torch.nn.functional

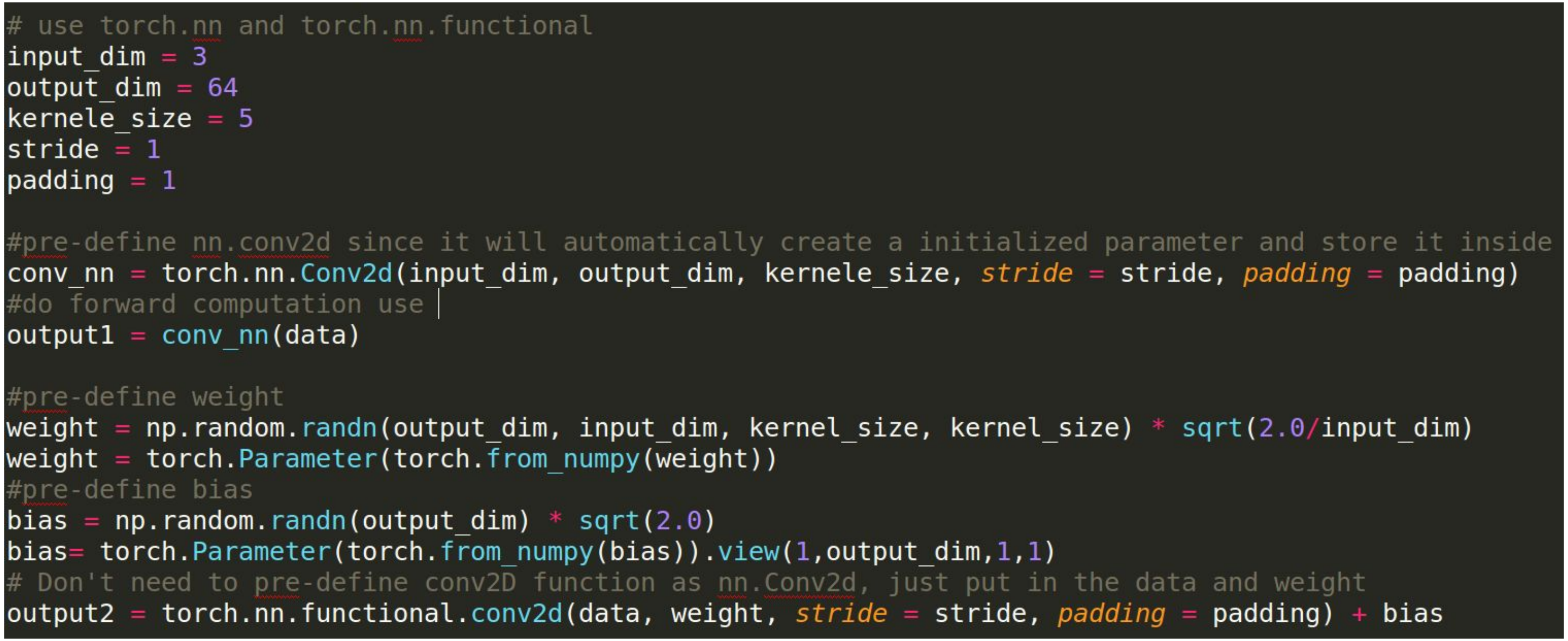


class `torch.nn.Conv2d`(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*) [source]

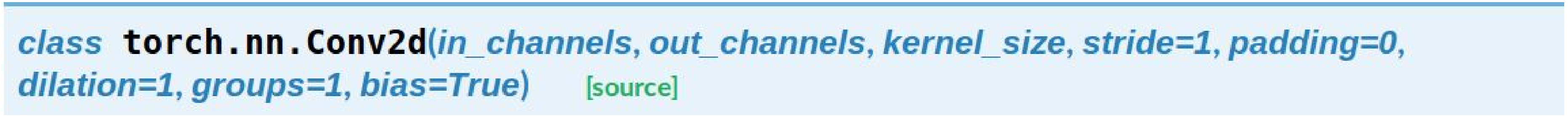


`torch.nn.functional.conv2d`(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*) [source]

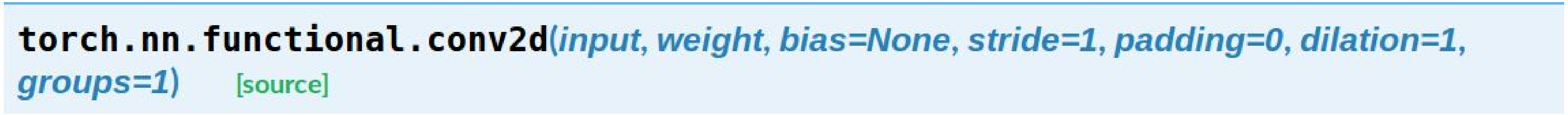
Pre-initialized weight



```python
# use torch.nn and torch.nn.functional
input_dim = 3
output_dim = 64
kernele_size = 5
stride = 1
padding = 1

#pre-define nn.conv2d since it will automatically create a initialized parameter and store it inside
conv_nn = torch.nn.Conv2d(input_dim, output_dim, kernele_size, stride = stride, padding = padding)
#do forward computation use 
output1 = conv_nn(data)

#pre-define weight
weight = np.random.randn(output_dim, input_dim, kernel_size, kernel_size) * sqrt(2.0/input_dim)
weight = torch.Parameter(torch.from_numpy(weight))
#pre-define bias
bias = np.random.randn(output_dim) * sqrt(2.0)
bias= torch.Parameter(torch.from_numpy(bias)).view(1,output_dim,1,1)
# Don't need to pre-define conv2D function as nn.Conv2d, just put in the data and weight
output2 = torch.nn.functional.conv2d(data, weight, stride = stride, padding = padding) + bias
```



class **torch.nn.Conv2d**(*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True*) [source]



torch.nn.functional.conv2d(*input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1*) [source]

Pytorch convolution format

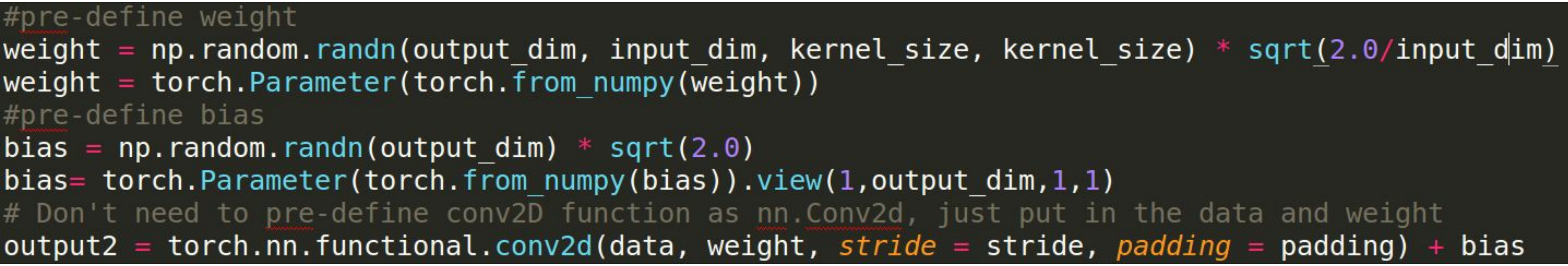
N * C * H * W

N:	Batch_size
C:	Channel
H:	Height
W:	Width

Pytoch Convolution Parameter format

OC * IC * K_h * K_w

OC:	Output Channel
IC:	Input Channel
K_h:	kernel height
K_w:	Kernel width



```
#pre-define weight
weight = np.random.randn(output_dim, input_dim, kernel_size, kernel_size) * sqrt(2.0/input_dim)
weight = torch.Parameter(torch.from_numpy(weight))
#pre-define bias
bias = np.random.randn(output_dim) * sqrt(2.0)
bias= torch.Parameter(torch.from_numpy(bias)).view(1,output_dim,1,1)
# Don't need to pre-define conv2D function as nn.Conv2d, just put in the data and weight
output2 = torch.nn.functional.conv2d(data, weight, stride = stride, padding = padding) + bias
```

When you want to use torch.nn:

1. The function has learnable parameter,

    nn.Conv2d, nn.Linear, nn.BatchNorm2d ...

When you want to use torch.nn.functional:

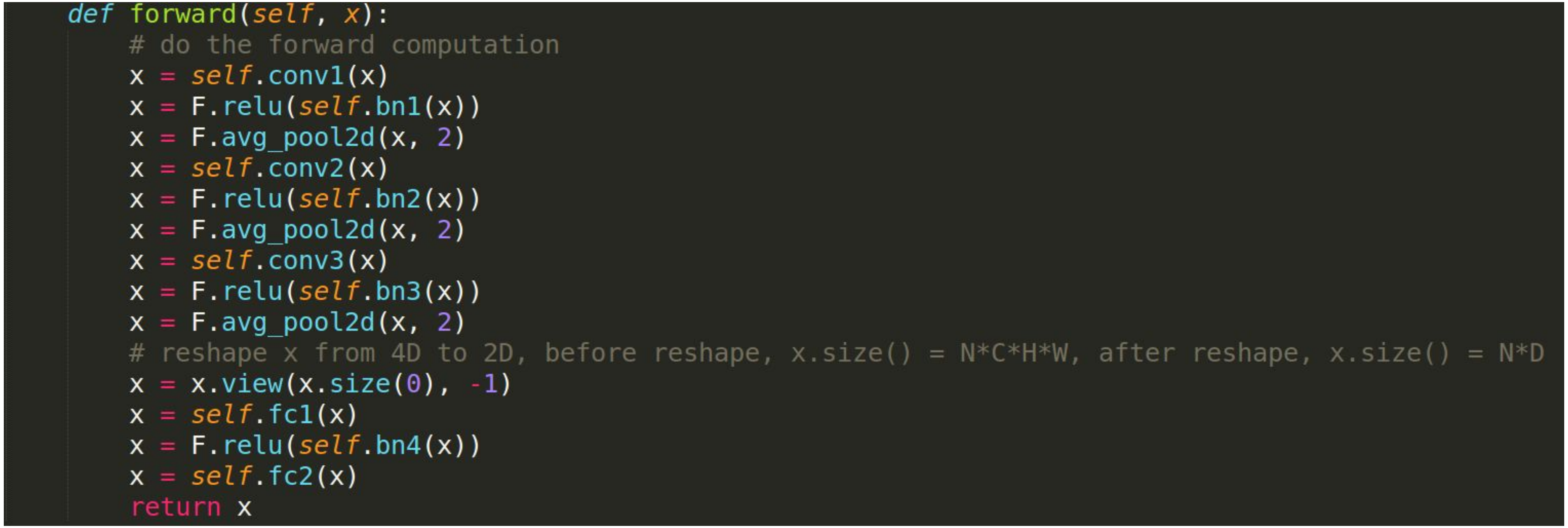
1 . The functional don’t have learnable paramter,

    F.relu, F.softmax, F.upsample, F.max_pool2d ...

But you can use torch.nn for non-learnable function

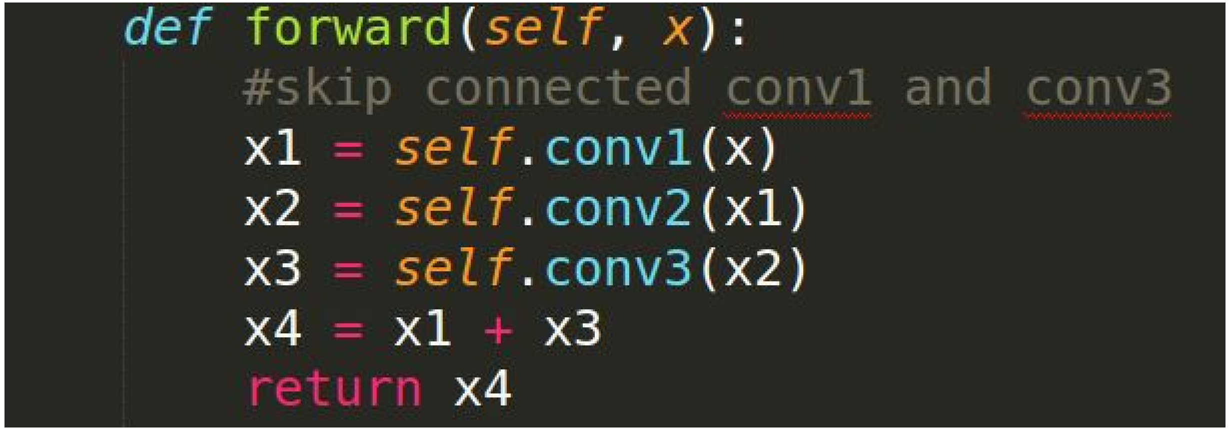
    nn.MaxPool2d, nn.CrossEntropyLoss

Use torch.nn.functional is not convenient for learnable function, because you need to manually pass the each weight to optimizer
But for torch.nn initialized inside the model, you can simply pass all the parameter to optimizer by model.parameters()



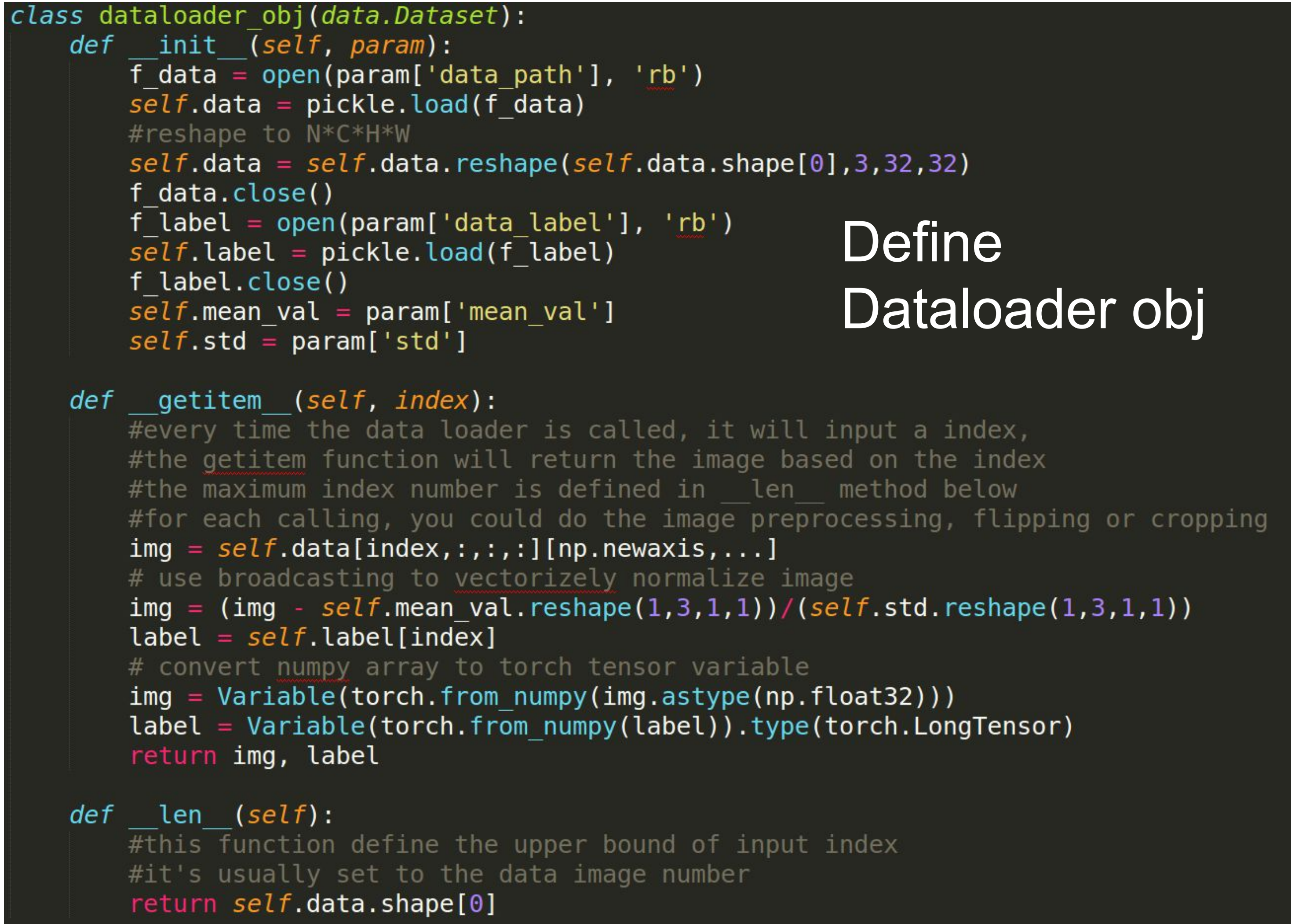
```
def forward(self, x):
    # do the forward computation
    x = self.conv1(x)
    x = F.relu(self.bn1(x))
    x = F.avg_pool2d(x, 2)
    x = self.conv2(x)
    x = F.relu(self.bn2(x))
    x = F.avg_pool2d(x, 2)
    x = self.conv3(x)
    x = F.relu(self.bn3(x))
    x = F.avg_pool2d(x, 2)
    # reshape x from 4D to 2D, before reshape, x.size() = N*C*H*W, after reshape, x.size() = N*D
    x = x.view(x.size(0), -1)
    x = self.fc1(x)
    x = F.relu(self.bn4(x))
    x = self.fc2(x)
    return x
```

Wisely choosing torch.nn.functional and torch.nn to write concise code



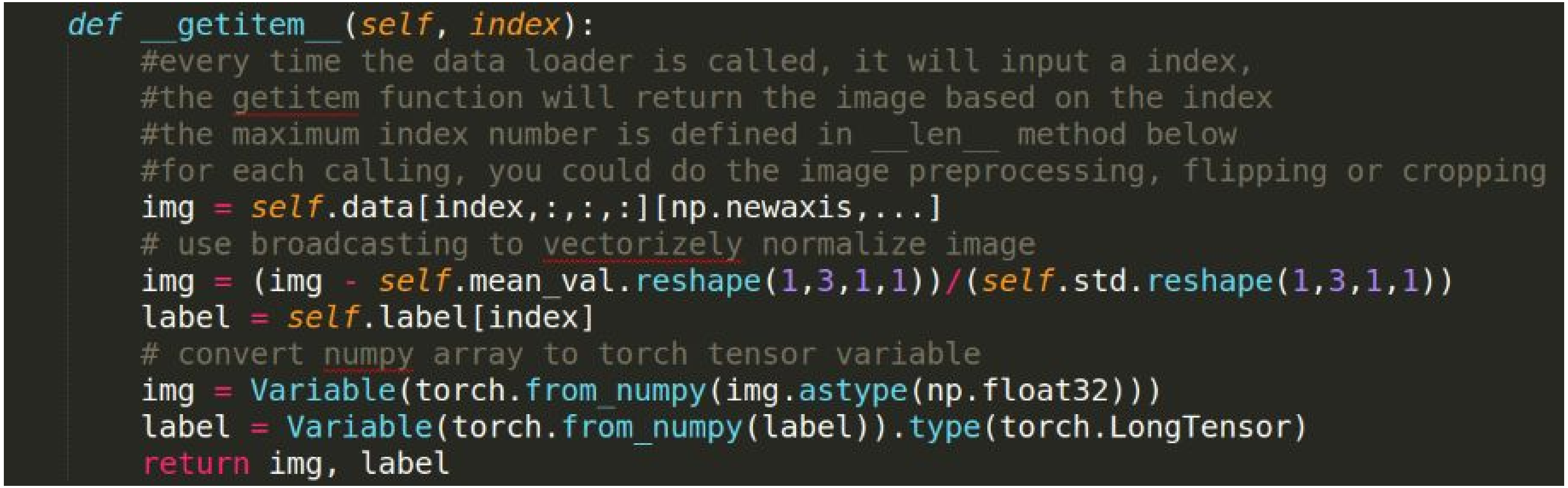
```
def forward(self, x):
    #skip connected conv1 and conv3
    x1 = self.conv1(x)
    x2 = self.conv2(x1)
    x3 = self.conv3(x2)
    x4 = x1 + x3
    return x4
```

The complex structure is easy to achieve in pytorch



```python
class dataloader_obj(data.Dataset):
    def __init__(self, param):
        f_data = open(param['data_path'], 'rb')
        self.data = pickle.load(f_data)
        #reshape to N*C*H*W
        self.data = self.data.reshape(self.data.shape[0],3,32,32)
        f_data.close()
        f_label = open(param['data_label'], 'rb')
        self.label = pickle.load(f_label)
        f_label.close()
        self.mean_val = param['mean_val']
        self.std = param['std']

    def __getitem__(self, index):
        #every time the data loader is called, it will input a index,
        #the getitem function will return the image based on the index
        #the maximum index number is defined in __len__ method below
        #for each calling, you could do the image preprocessing, flipping or cropping
        img = self.data[index,:,:,:][np.newaxis,...]
        # use broadcasting to vectorizely normalize image
        img = (img - self.mean_val.reshape(1,3,1,1))/(self.std.reshape(1,3,1,1))
        label = self.label[index]
        # convert numpy array to torch tensor variable
        img = Variable(torch.from_numpy(img.astype(np.float32)))
        label = Variable(torch.from_numpy(label)).type(torch.LongTensor)
        return img, label

    def __len__(self):
        #this function define the upper bound of input index
        #it's usually set to the data image number
        return self.data.shape[0]
```

Define Dataloader obj



```
def __getitem__(self, index):
    #every time the data loader is called, it will input a index,
    #the getitem function will return the image based on the index
    #the maximum index number is defined in __len__ method below
    #for each calling, you could do the image preprocessing, flipping or cropping
    img = self.data[index,:,:,:][np.newaxis,...]
    # use broadcasting to vectorizely normalize image
    img = (img - self.mean_val.reshape(1,3,1,1))/(self.std.reshape(1,3,1,1))
    label = self.label[index]
    # convert numpy array to torch tensor variable
    img = Variable(torch.from_numpy(img.astype(np.float32)))
    label = Variable(torch.from_numpy(label)).type(torch.LongTensor)
    return img, label
```

You can treat dataloader_obj in pytorch as a iterator. Each time it's called, it will passed in an index and return a single image based on the index.

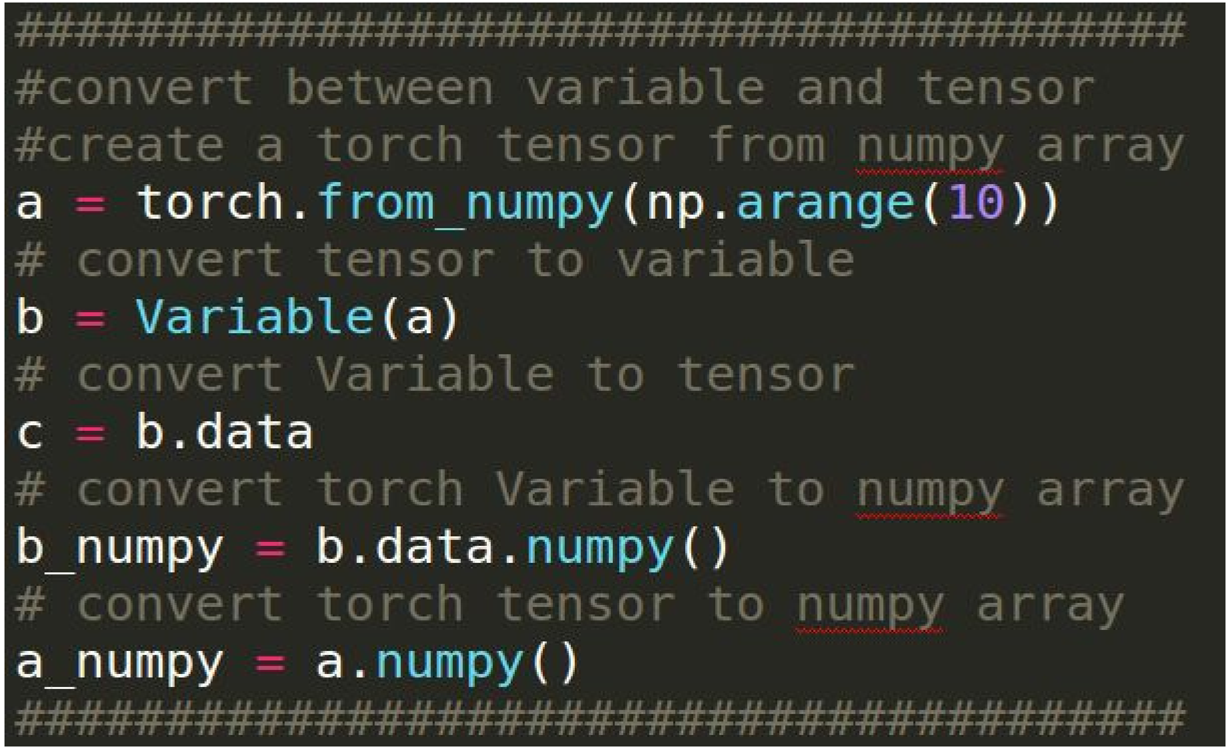
Suppose you have image array (N*H*W*3) or list of image path(N), It's common to return the index'th image from the list.

Pytorch has two datatype:

Variable
Tensor

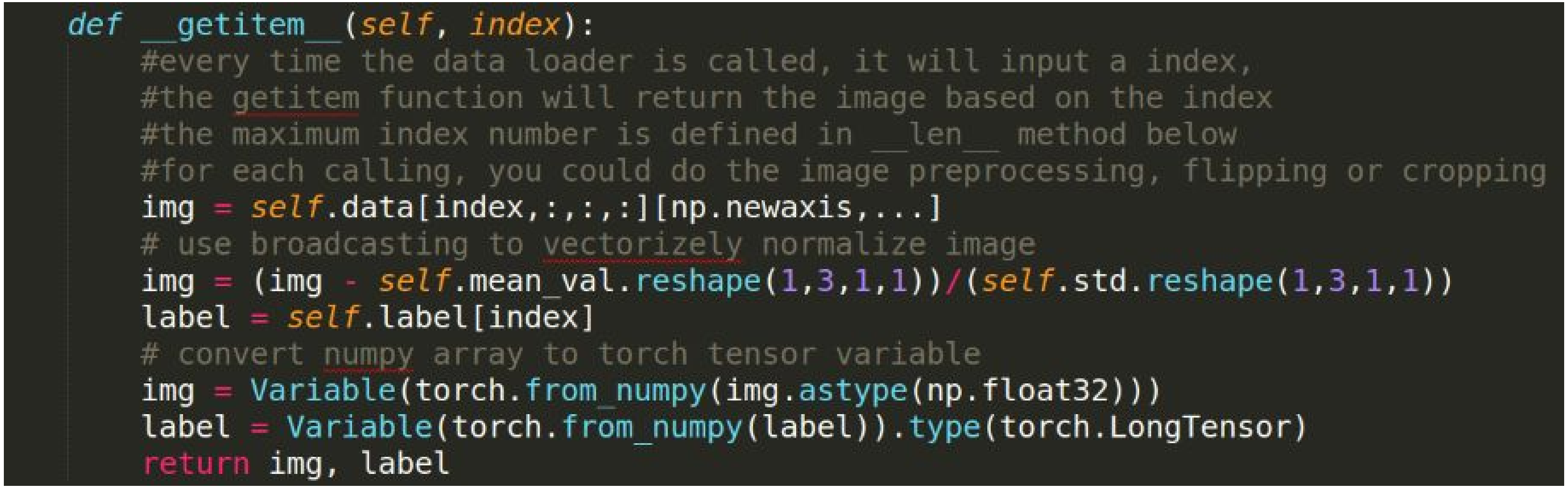
The computation start from input Variable and the output is also a Variable

The tensor exist inside intermediate computation step (the computation inside nn.conv2d is performed on tensor)



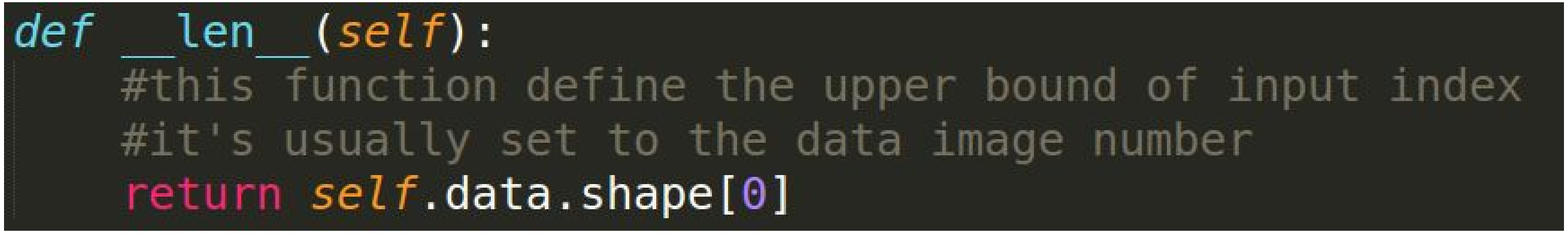
```
########################################
#convert between variable and tensor
#create a torch tensor from numpy array
a = torch.from_numpy(np.arange(10))
# convert tensor to variable
b = Variable(a)
# convert Variable to tensor
c = b.data
# convert torch Variable to numpy array
b_numpy = b.data.numpy()
# convert torch tensor to numpy array
a_numpy = a.numpy()
########################################
```

Conversion among Variable, Tensor and Numpy array



```
    def __getitem__(self, index):
        #every time the data loader is called, it will input a index,
        #the getitem function will return the image based on the index
        #the maximum index number is defined in __len__ method below
        #for each calling, you could do the image preprocessing, flipping or cropping
        img = self.data[index,:,:,:][np.newaxis,...]
        # use broadcasting to vectorizely normalize image
        img = (img - self.mean_val.reshape(1,3,1,1))/(self.std.reshape(1,3,1,1))
        label = self.label[index]
        # convert numpy array to torch tensor variable
        img = Variable(torch.from_numpy(img.astype(np.float32)))
        label = Variable(torch.from_numpy(label)).type(torch.LongTensor)
        return img, label
```

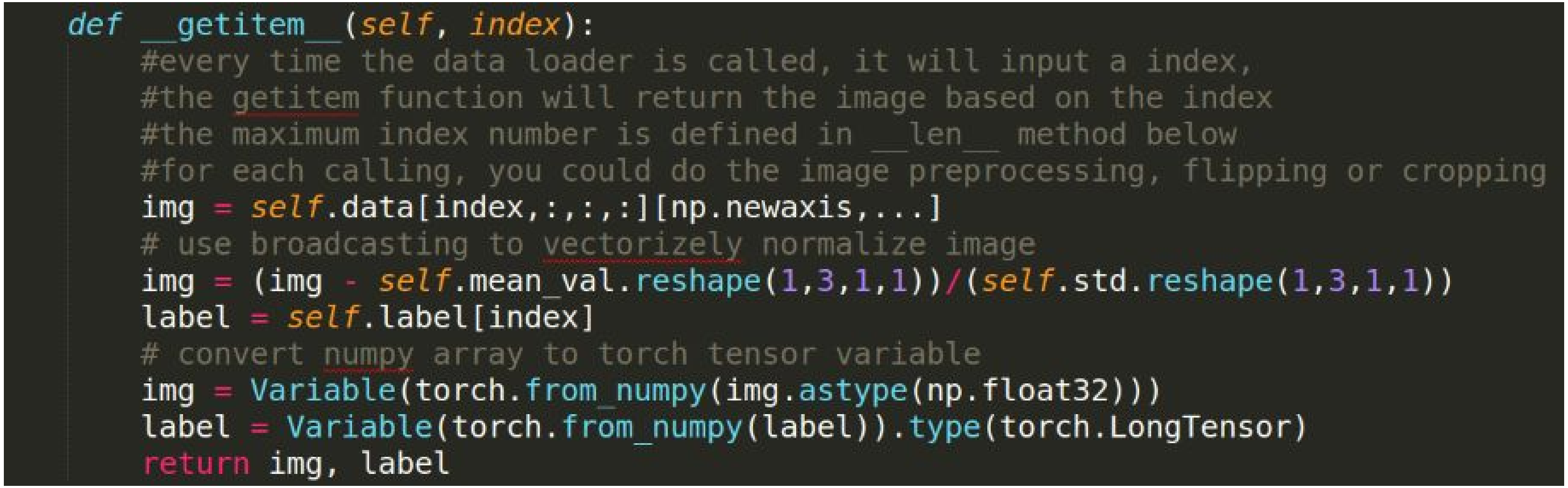
It's better to wrap the numpy data to torch Variable inside the dataloader, because every operation inside the dataloader is in parallel.
So as efficiency concern, please put all the preprocessing step inside the dataloader



```python
def __len__(self):
    #this function define the upper bound of input index
    #it's usually set to the data image number
    return self.data.shape[0]
```

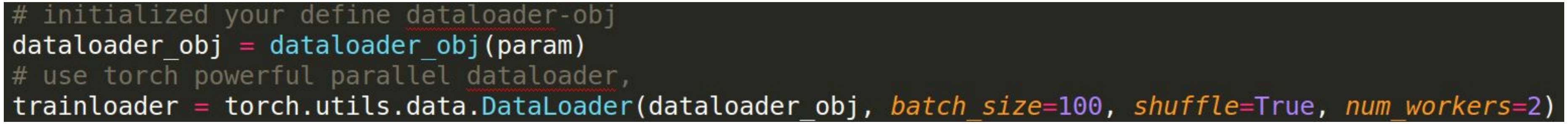
This method define the maximum index will be passed to the __getitem__ method.

It’s common to set the dataloader_obj’s length same as image number



```
def __getitem__(self, index):
    #every time the data loader is called, it will input a index,
    #the getitem function will return the image based on the index
    #the maximum index number is defined in __len__ method below
    #for each calling, you could do the image preprocessing, flipping or cropping
    img = self.data[index,:,:,:][np.newaxis,...]
    # use broadcasting to vectorizely normalize image
    img = (img - self.mean_val.reshape(1,3,1,1))/(self.std.reshape(1,3,1,1))
    label = self.label[index]
    # convert numpy array to torch tensor variable
    img = Variable(torch.from_numpy(img.astype(np.float32)))
    label = Variable(torch.from_numpy(label)).type(torch.LongTensor)
    return img, label
```

Recap:
The image should have form (N*C*H*W) and each time the dataloader_obj is called, it will upload 1 image (1*C*H*W) and its label (1)



```
# initialized your define dataloader-obj
dataloader_obj = dataloader_obj(param)
# use torch powerful parallel dataloader,
trainloader = torch.utils.data.DataLoader(dataloader_obj, batch_size=100, shuffle=True, num_workers=2)
```

Remember each time we call the dataloader_obj, it will only return one image and one label.
Now if we set batch_size = 100, we expect every time we call the dataloader, it will return 100 images and corresponding labels.

To achieve this, we can utilize the powerful pytorch Dataloader.

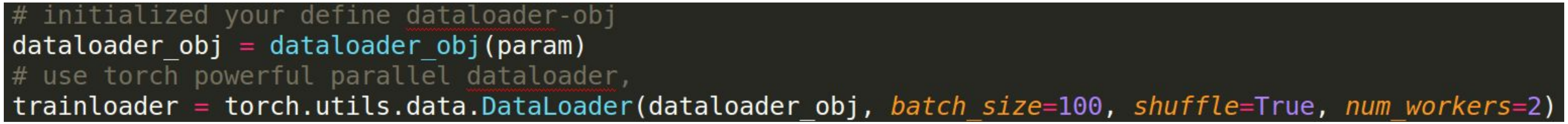
Clarify:

dataloader_obj :

Inherited from torch.utils.data.Dataset, it’s a class you can customize define the operation for loading image

torch.utils.data.DataLoader :

It’s a function, it takes input a dataloader_obj and parallely call dataloader_obj, do the shuffle and wrap loaded image till the batch_size



```
# initialized your define dataloader-obj
dataloader_obj = dataloader_obj(param)
# use torch powerful parallel dataloader,
trainloader = torch.utils.data.DataLoader(dataloader_obj, batch_size=100, shuffle=True, num_workers=2)
```

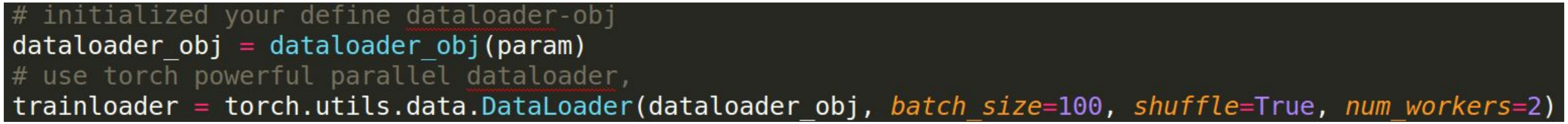
Recap

In dataloader_obj, we define each time it's called, it will return a image Variable (1*C*H*W) and it's label Variable (1)

Now if we want to set the batch_size = 100, we want (100*C*H*W) image variable and (100) label for forward computing, we just need to set batch_size parameter = 100.

The trainloader will return a full batch_size image when the dataloader_obj is called 100 times and the image will be stacked along the first dimension.

If you defined in dataloader_obj: every time it return 3 image Variable(3*C*H*W) and 2 label Variable (2) and set batch_size =100, the dataloader will return (300*C*H*W) image variable and (200) label Variable.

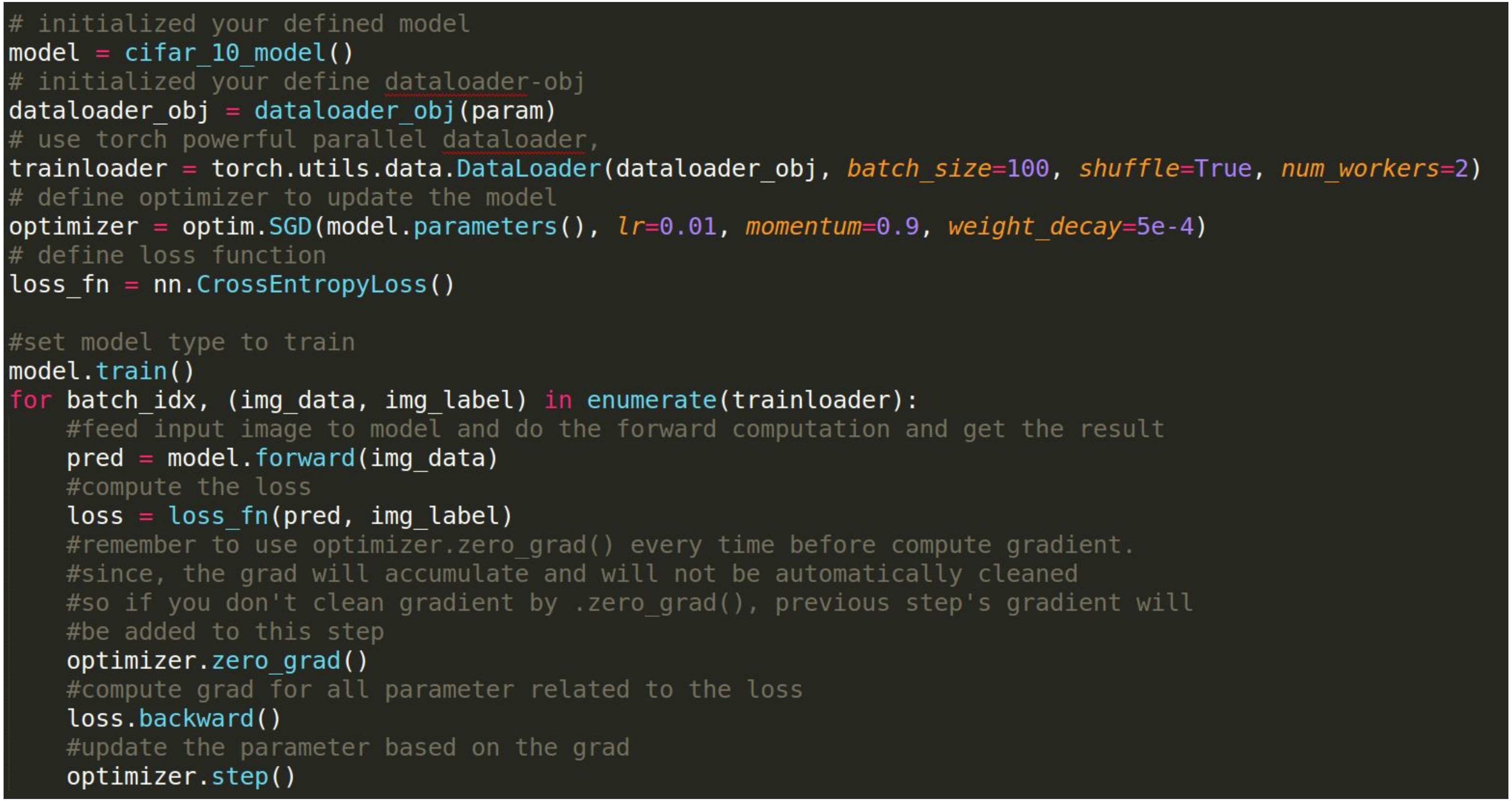


```
# initialized your define dataloader-obj
dataloader_obj = dataloader_obj(param)
# use torch powerful parallel dataloader,
trainloader = torch.utils.data.DataLoader(dataloader_obj, batch_size=100, shuffle=True, num_workers=2)
```

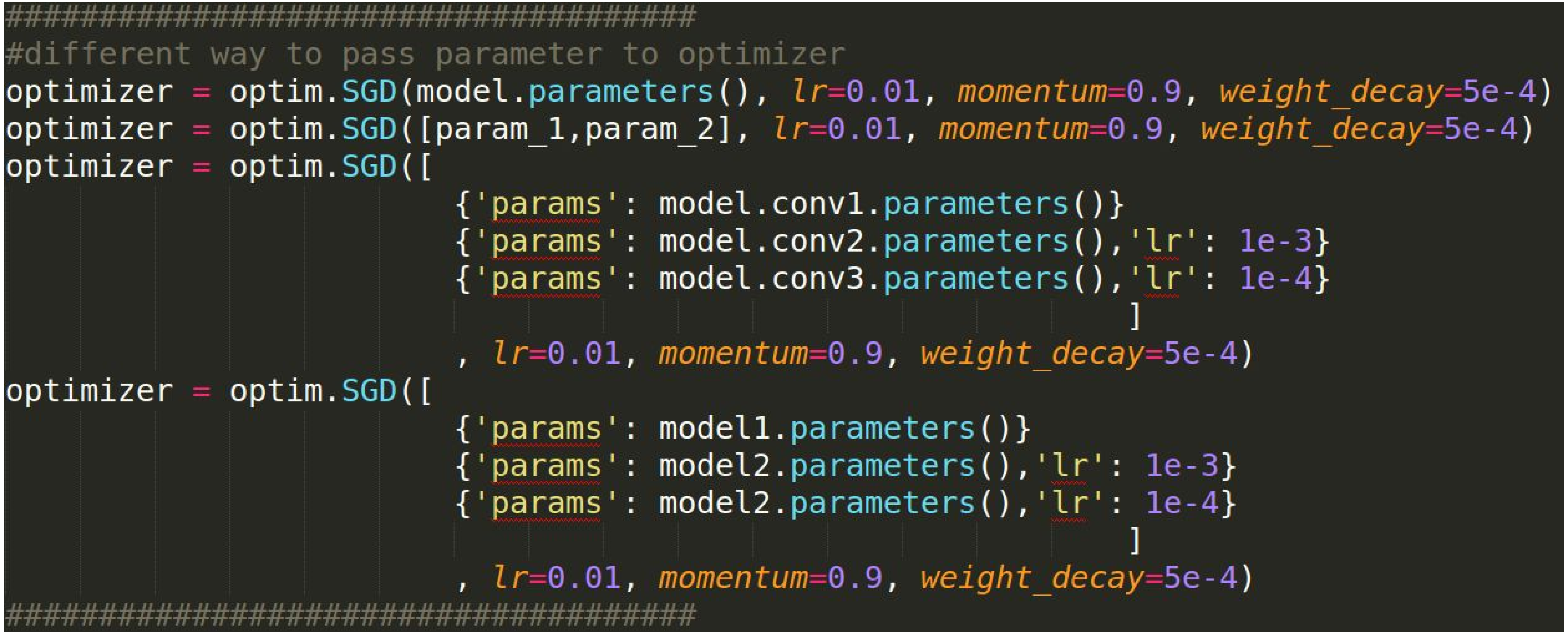
Shuffle:

Randomized pass a index to dataloader_obj __getitem__(self, index).

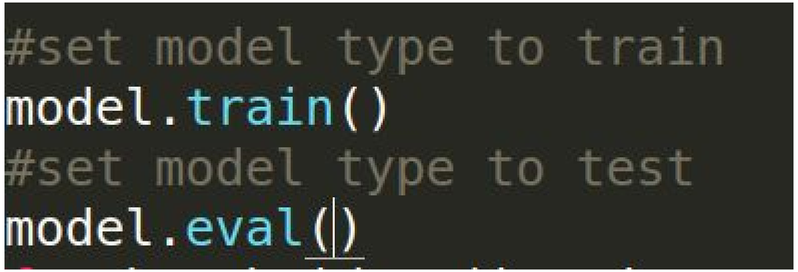
Once one index is passed, it will not be shown again, so that the output image will not be duplicated.



```python
# initialized your defined model
model = cifar_10_model()
# initialized your define dataloader-obj
dataloader_obj = dataloader_obj(param)
# use torch powerful parallel dataloader,
trainloader = torch.utils.data.DataLoader(dataloader_obj, batch_size=100, shuffle=True, num_workers=2)
# define optimizer to update the model
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
# define loss function
loss_fn = nn.CrossEntropyLoss()

#set model type to train
model.train()
for batch_idx, (img_data, img_label) in enumerate(trainloader):
    #feed input image to model and do the forward computation and get the result
    pred = model.forward(img_data)
    #compute the loss
    loss = loss_fn(pred, img_label)
    #remember to use optimizer.zero_grad() every time before compute gradient.
    #since, the grad will accumulate and will not be automatically cleaned
    #so if you don't clean gradient by .zero_grad(), previous step's gradient will
    #be added to this step
    optimizer.zero_grad()
    #compute grad for all parameter related to the loss
    loss.backward()
    #update the parameter based on the grad
    optimizer.step()
```



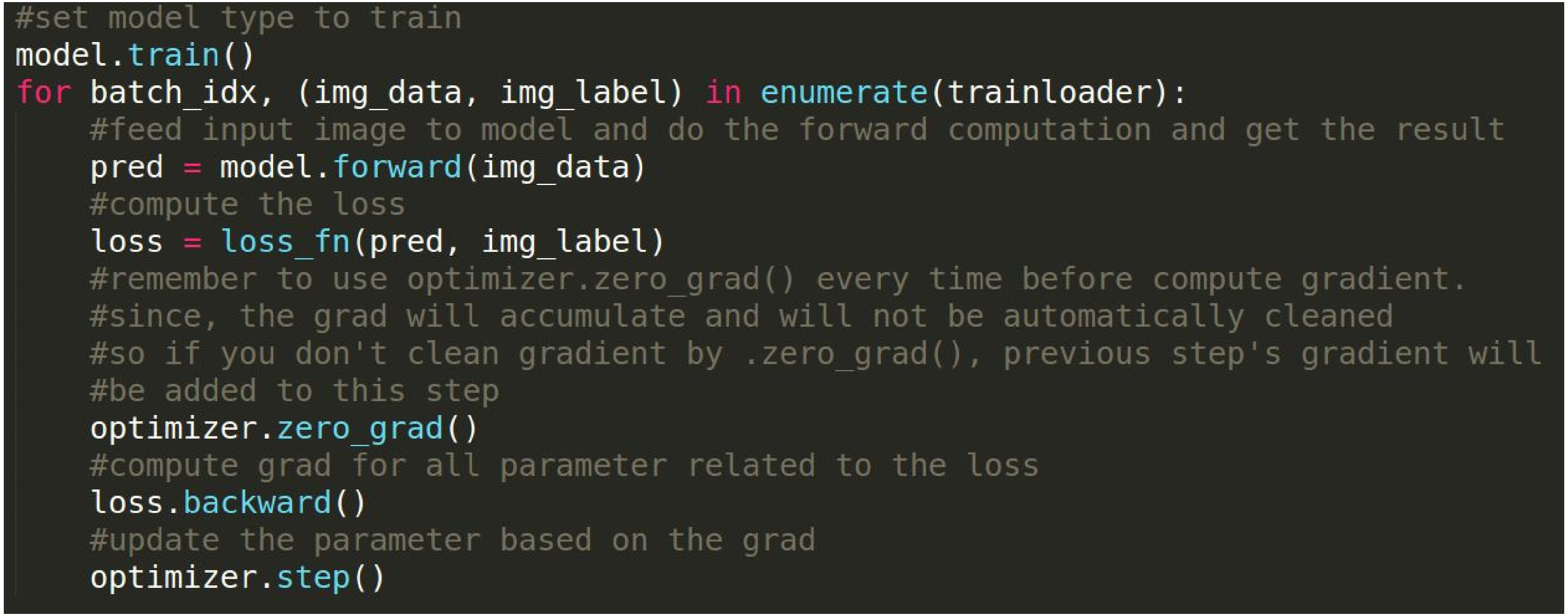
```
#####################################
#different way to pass parameter to optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-4)
optimizer = optim.SGD([param_1,param_2], lr=0.01, momentum=0.9, weight_decay=5e-4)
optimizer = optim.SGD([
                        {'params': model.conv1.parameters()}
                        {'params': model.conv2.parameters(),'lr': 1e-3}
                        {'params': model.conv3.parameters(),'lr': 1e-4}
                                                            ]
                        , lr=0.01, momentum=0.9, weight_decay=5e-4)
optimizer = optim.SGD([
                        {'params': model1.parameters()}
                        {'params': model2.parameters(),'lr': 1e-3}
                        {'params': model2.parameters(),'lr': 1e-4}
                                                            ]
                        , lr=0.01, momentum=0.9, weight_decay=5e-4)
#####################################
```



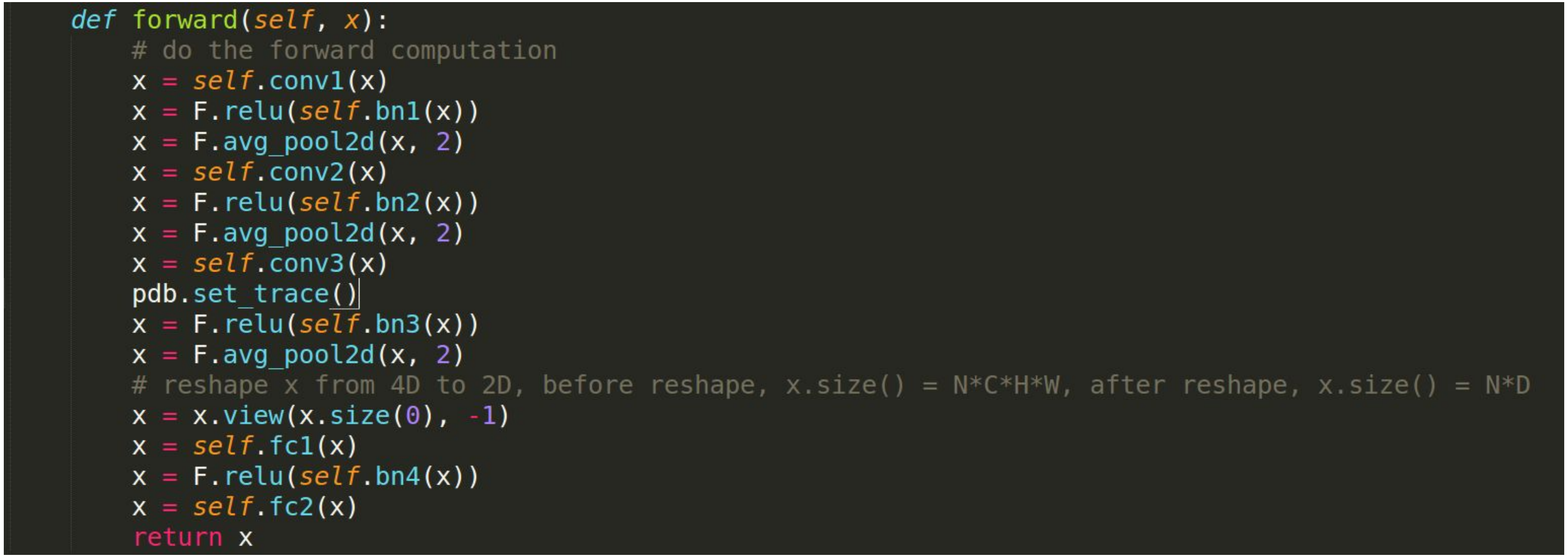
```
#set model type to train
model.train()
#set model type to test
model.eval()
```

Change model mode before training or testing

Cause some layer behave differently for training and testing



```python
#set model type to train
model.train()
for batch_idx, (img_data, img_label) in enumerate(trainloader):
    #feed input image to model and do the forward computation and get the result
    pred = model.forward(img_data)
    #compute the loss
    loss = loss_fn(pred, img_label)
    #remember to use optimizer.zero_grad() every time before compute gradient.
    #since, the grad will accumulate and will not be automatically cleaned
    #so if you don't clean gradient by .zero_grad(), previous step's gradient will
    #be added to this step
    optimizer.zero_grad()
    #compute grad for all parameter related to the loss
    loss.backward()
    #update the parameter based on the grad
    optimizer.step()
```



```
    def forward(self, x):
        # do the forward computation
        x = self.conv1(x)
        x = F.relu(self.bn1(x))
        x = F.avg_pool2d(x, 2)
        x = self.conv2(x)
        x = F.relu(self.bn2(x))
        x = F.avg_pool2d(x, 2)
        x = self.conv3(x)
        pdb.set_trace()
        x = F.relu(self.bn3(x))
        x = F.avg_pool2d(x, 2)
        # reshape x from 4D to 2D, before reshape, x.size() = N*C*H*W, after reshape, x.size() = N*D
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = F.relu(self.bn4(x))
        x = self.fc2(x)
        return x
```

Set the break point anywhere you like, using IDE’s break point or set the pdb.set_trace(), you can print out the immediately result any time you like.

1. Remember to convert the image data to np.float32 before transfer to torch.variable. Float32 is a good trade off between speed and accuracy.
2. Remember to check the function document before using it, especially the input data shape and input datatype.