Jinchao Xu

PSU-PKU Joint Course (497): An Introduction to Deep Learning

Summer 2019

1

Project

1.1 Introduction to train a Neural Network

1.1.1 PyTorch Basics

In this section, we will introduce PyTorch [1] which is an open source machine learning framework that accelerates the path from research prototyping to production deployment.

The basic class underlying pytorch is the tensor class. This class simply represents an array of numeric values (usually floats or doubles) and behaves very similar to the corresponding class in numpy.

```python
import torch
import numpy as np

# We can initialize a pytorch tensor from a python list.
x = torch.tensor([[1,2],[3,4]])
print(’x is’, x)

# Or from a numpy array.
y = torch.tensor(np.array([[1,2],[3,4]]))
print(’y is’, y)

# We can also initialize a zeroed tensor.
a = torch.zeros([2,2])
b = torch.zeros([3,3], dtype=torch.int32)

# Note that the default type of such a tensor is a 32-bit float.
print(’Type of a:’, a.dtype)
print(’Type of b:’, b.dtype)
```

The output will be

[1] https://pytorch.org

```
x is tensor([[1, 2],
        [3, 4]])
y is tensor([[1, 2],
        [3, 4]])
Type of a: torch.float32
Type of b: torch.int32
```

Arithmetic operations on PyTorch tensors work as one would expect.

```
z = 2*x
print(z)

z = y - z
print(z)

z = torch.matmul(x,y)
print(z)

# etc. see the documentation for more details.
```

It gives

```
tensor([[2, 4],
        [6, 8]])
tensor([[-1, -2],
        [-3, -4]])
tensor([[ 7, 10],
        [15, 22]])
```

At this point, there isn't much difference between Pytorch and Numpy. What distinguishes Pytorch and makes it useful for Deep Learning is the automatic differentiation supported by Pytorch (essentially, Pytorch can automatically apply the chain rule to calculate derivatives). We illustrate this below with a simple example.

```
# In this example we calculate the derivative of (x.y)^2 w.r.t
                                x.

# In order to calculate the gradient with respect to a tensor,
# we must set the requires_grad flag to True.
x = torch.tensor([1.0, 0.0], requires_grad=True)
y = torch.tensor([1.0, 1.0])

# By default, requires_grad is set to False if possible.
print('y.requires_grad:', y.requires_grad)

z = torch.dot(x, y)

# Because z depends on x, which requires a gradient, z.
                               requires_grad
```

```
# is automatically set to true. This is done because the
                gradient of z
# is needed to calculate the gradient of x.
print('z.requires_grad:', z.requires_grad)

out = z * z

# Calculate the derivative of out w.r.t. x. Automatically
                applies
# the chain rule as needed.
grad = torch.autograd.grad(outputs=out, inputs=x)
print(grad)
```

The output will be

```
y.requires_grad: False
z.requires_grad: True
(tensor([2., 2.]),)
```

Using what we have so far, we can automatically differentiate polynomials (since they are just compositions of elementary arithmetic operations). But what about more complicated differentiable functions? We can deal with those by subclassing torch.autograd.Function as we show in the next example.

```
# We illustrate this by implementing the pointwise ReLU
                function
# (which is of course already implemented in Pytorch, but this
# process can be used to define more complex custom functions
                as well).

# We subclass torch.autograd.Function to define new functions
class MyReLU(torch.autograd.Function):

  # The forward method applies the function to the input
                argument.
  # The ctx argument can be used to cache information for the
                subsequent
  # gradient computation. You can cache an object using using
                the
  # ctx.save_for_backward method.
  @staticmethod
  def forward(ctx, input):
    # The input is needed later to calculate the gradient.
    ctx.save_for_backward(input)
    return input.clamp(min=0)

  # The backward method calculates the gradient of the input,
                given that
  # the gradient of the output is the output_grad argument.
  def backward(ctx, output_grad):
    # Recover the original input from ctx.
    input, = ctx.saved_tensors
    grad_input = output_grad.clone()
    # Zero out the output gradient where the input is negative
                                    to obtain
    # the input gradient.
    grad_input[input < 0] = 0
    return grad_input

# Apply the new function to a tensor.
print(MyReLU.apply(x - 0.5))

# Calculating derivatives utilizes the backward method.
z = torch.mul(MyReLU.apply(x - 0.5), y)
# In this case z is not a scalar, so we must provide the
                               derivatives
# of the outputs. What is actually computed is the gradient of
# z dot output_grad with respect to x.
output_grad = torch.ones([2])
print(torch.autograd.grad(outputs=z, inputs=x, grad_outputs =
                               output_grad))
```

```
tensor([0.5000, 0.0000], grad_fn=<MyReLUBackward>)
(tensor([1., 0.]),)
```

We have seen how to define a custom function. Luckily, we won't have to do this much work very often. As we will see, many common functions in machine learning have already been implemented for us.

Next we introduce the Variable class in torch.autograd. This class used to be necessary for autograd, but since regular tensors now allow for automatic differentiation, it is basically just a wrapper around the tensor class which introduces a few additional methods, most notably the .backward() method.

```
from torch.autograd import Variable

# Variables can be copy constructed from tensors. When
                               constructed, we can
# also specify whether the gradient is required or not.
x = Variable(x, requires_grad=True)
y = Variable(y, requires_grad=False)

# All functions which can be applied to tensors can also be
                               applied to
# Variables. In the code below, z will be a Variable since x
                               and y are.
# In general, if at least one input is a Variable, then the
                               output will be as
# well.
z = torch.mul(MyReLU.apply(x - 0.5), y)
```

```
# Notably, we can now call .backward() from z_variable, which
                               will compute
# the gradient with respect to z for all tensors it depends on
                                (if their
# requires_grad field is True). Note that since z isn't a
                               scalar, we still
# need to pass the gradients of each of its components.
z.backward(gradient=output_grad)

# The resulting gradients are stored in the .grad field of the
                                corresponding
# tensors. The gradients are only calculated if requires_grad
                               = True.
print('x gradient:', x.grad)
print('y gradient:', y.grad)
```

```
x gradient: tensor([1., 0.])
y gradient: None
```

Since Variables contain all of the functionality as tensors, but contain additional methods (especially the very useful .backward() method) as well, we recommend using Variables for all non-constant quantities in your code. We illustrate some of the additional behaviors of the .backward() function below.

```
# A very important note is that gradients are accumulated, i.e
                               . added to
# whatever is already stored in the .grad field.
z = torch.mul(MyReLU.apply(x - 0.5), y)
z.backward(gradient=output_grad)
print('x.grad has now been doubled:', x.grad)

# In order to avoid this, we must first clear the gradient of
                               x. Note also
# that in the above example we need to recompute z in terms of
                                x and y
# in order to recompute the gradient. We can avoid this by
                               setting
# retain_graph = True
x.grad.data.zero_()
z = torch.mul(MyReLU.apply(x - 0.5), y)
z.backward(gradient=output_grad, retain_graph=True)
print('x gradient:', x.grad)

# Since we set retain_graph=True during the last backward call
                               , we can
# recompute the gradients without recalculating z. This
                               feature is not
# commonly used and considered bad practice, though.
```

```
x.grad.data.zero_()
z.backward(gradient=output_grad)
print('x gradient:', x.grad)
```

```
x.grad has now been doubled: tensor([2., 0.])
x gradient: tensor([1., 0.])
x gradient: tensor([1., 0.])
```

1.1.2 Building Neural Networks

Building and training neural networks directly using Pytorch Variables and automatic differentiation would be very cumbersome. Luckily, Pytorch provides built-in functionality which makes it much easier to build and train neural networks. In this section, we will describe how to use this functionality to build (i.e. define) a network.

The most important library for building neural networks is the torch.nn library. This library allows us to build neural networks by concatenating different types of layers.

```
import torch.nn as nn

# Here we define a sample neural network. The class defining
                               our
# network should inherit from nn.Module.
class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    # Here we define the layers of the network.
    # The nn.Sequential method constructs a model by
                                   concatenating
    # the layers which are input to it.
    self.apply = nn.Sequential( # Sequentially apply a
        nn.Linear(10,20),       # Linear function R^10 -> R^20
        nn.ReLU(),              # Pointwise ReLU
        nn.Linear(20,20),       # Linear function R^20 -> R^20
        nn.ReLU(),              # Pointwise ReLU
        nn.Linear(20,10))       # Linear function R^20 -> R^10

  # The following method must be overloaded. It specifies how
                                 to
  # evaluate the model given an input x.
  def forward(self, x):
    return self.apply(x) # In our instance we simply pass x
                                   through
                         # the previously defined layers. This
                                                      could
                         # potentially contains something more
```

```
                                                        complex
                                                        .
```

Implementing a Logistic Regression We will now consider the problem of implementing a logistic regression (trained using stochastic gradient descent) using Pytorch. We will test this model on the MNIST handwritten data imageset.

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

# Transforms images from [0,255] to [0,1] range.
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize(mean=[0], std=[1])])

# Load the set of training images.
trainset = torchvision.datasets.MNIST(root='./data', train=
                               True, download=True, transform=
                               transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size
                               =4, shuffle=True, num_workers=2
                               )

# Load the set of test images.
testset = torchvision.datasets.MNIST(root='./data', train=
                               False, download=True, transform
                               =transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4
                               , shuffle=False, num_workers=2)

# Let's plot some of the images to see what we're dealing with
                               .
def plot_images(imgs):
  for i in range(imgs.size()[0]):
    npimg = imgs.numpy()[i,0,:,:]
    plt.imshow(npimg, cmap='gray')
    plt.ion()
    plt.show()
    plt.pause(.05)

data = iter(testloader)
images, labels = data.next()
print(labels)
plot_images(images)
```

Now that we've loaded our dataset, let's build the logistic regression model. In the language of deep learning, the logistic regression is the same as a neural network with no hidden layers trained with a cross entropy loss (exercise: work this out yourself!).

```
# This library contains a lot of useful classes for
                               constructing
# neural networks and other machine learning models.
import torch.nn as nn

# This model simply takes an input of dimension 784 and
                               multiplies it
# by a 784x10 matrix of parameters.
class LogisticRegression(nn.Module):
  def __init__(self):
    super(LogisticRegression, self).__init__()
    self.dense_linear = nn.Sequential(
      nn.Linear(28 * 28, 10))

  def forward(self, x):
    x = x.view(-1,28*28)
    x = self.dense_linear(x)
    return x

# Construct an instance of the model.
model = LogisticRegression()
```

Now we need to fit the parameters of this logistic regression model (the 784x10 matrix). We do this by defining a loss function (the cross entropy) and using the stochastic gradient descent optimization algorithm. Luckily, both of these have already been implemented for you in Pytorch!

```
# This library contains implementations of a number of useful
                               optimization algorithms.
import torch.optim as optim

# The cross entropy loss is already implemented in Pytorch.
criterion = nn.CrossEntropyLoss()

# The stochastic gradient descent algorithm with a step size
                               of 0.1.
optimizer = optim.SGD(model.parameters(), lr=0.1)

# Write a loop to train the model using the given optimizer
                               and loss functions.
for i in range(20):
  for data in trainloader:
    # extract the images and labels.
    inputs, labels = data
    # This must be called to zero out the accumulated
                                   gradients.
    optimizer.zero_grad()

    # Calculate the predictions made based on the model.
    outputs = model(inputs)

    # Evaluate the loss function based on the model
                                   predictions.
    loss = criterion(outputs, labels)

    # Calculate the gradient of the parameters with respect to
                                    the loss.
    loss.backward()

    # Take a optimization step.
    optimizer.step()

  print(’Completed epoch %d’ % i)
print(’Completed training’)
```

Now that we’ve trained our model, we will test its accuracy on the test dataset.

```
# Calculate the total number of test samples and the number of
                                correctly
# classified test samples
correct = 0
total = 0
for data in testloader:
  images, labels = data
  outputs = model(images)
  # Take the most likely label as the predicted label.
  _, predicted = torch.max(outputs.data, 1)
  total += labels.size(0)
  correct += (predicted == labels).sum()

print(’Out of %d samples, the model correctly classified %d’ %
                                (total, correct))
```

1.1.3 Datasets

In this section, we will talk about how to train a neural network. According to the course in the morning, there are a lot of pictures in a dataset, so, at the beginning, we will introduce some famous and popular datasets.

- **MNIST**[2] contains 60,000 pictures of the ten handwritten digits from 0 to 9. The size of each picture is 28 × 28 pixels. There are 60,000 pictures for training and 10,000 for testing.

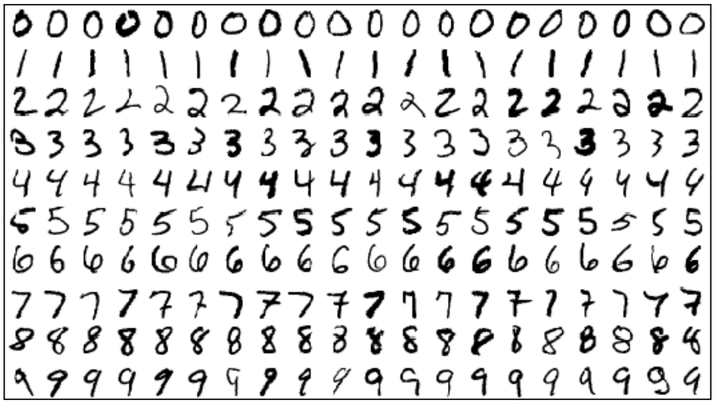


Fig. 1.1. MNIST

- **CIFAR10**[3] is a datasets consists of 60, 000 color images in 10 classes, with 6, 000 images per class. There are 50, 000 training data and 10, 000 testing data. Since there are colored pictures, there are three channels, red, green and blue, of the input data. The size is 32 × 32 pixels for each. It's a subclass of CIFAR100.
- **ImageNet**[4] is a much larger dataset containing 1.2 millions pictures. The size is 224 × 224 pixels. The size is much bigger and the quantity is also much larger. So you may get a good performance with a simple neural network on MNIST or CIFAR10, but it's much hard to get good accuracy on ImageNet.

Fig. 1.3. ImageNet

[2] http://yann.lecun.com/exdb/mnist/
[3] https://www.cs.toronto.edu/~kriz/cifar.html
[4] http://www.image-net.org

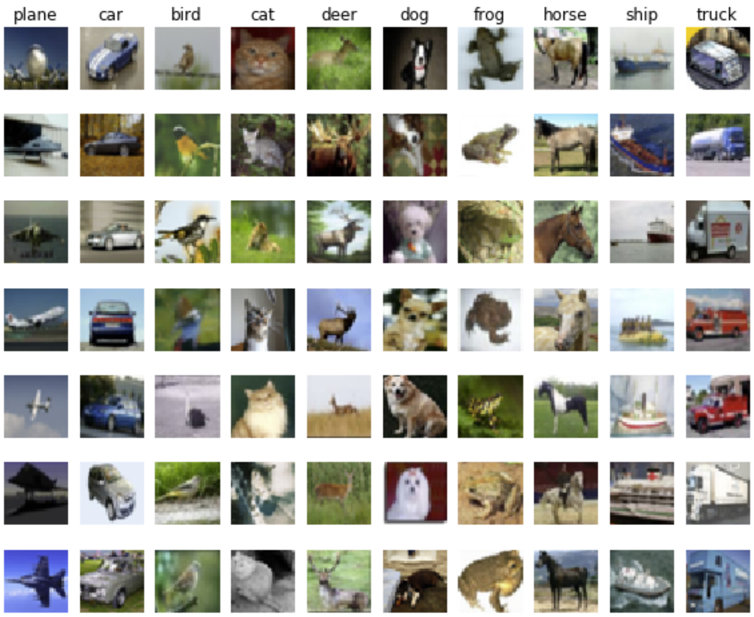


Fig. 1.2. CIFAR10

These are three common used datasets that we will use to train our neural networks.

1.1.4 Train an image classifier

Now we move to a brief PyTorch [5] tutorial. To train your first image classifier, these are basically five steps in order:

1. Load and normalizing the CIFAR10 training and test datasets using torchvision (Here we use CIFAR10 as an example).
2. Define a Convolutional Neural Network. (You can also use a Deep Neural Networks or Recurrent Neural Networks whichever you like.)
3. Define a loss function. (As we just defined before as the function f that is the sum of the total losses.)
4. Train the network on the training data. (Given the loss function and training data, we can do the steepest descent, stochastic gradient descent or some other optimization algorithms to decrease the loss function on the training set.)
5. Test the network on the test data. (After you have got a good model on your neural network from the last step, then you test your neural network on the test set to see how well it performs. So the final criterion that judges your neural network is the test accuracy because the test set doesn't involve the training process. So it is very important to get a good test accuracy.)

[5] see a quick tutorial https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

These are all the steps you need to train a neural network. Now let start from the first step.

Load a dataset

To load the dataset, we need the libraries, like torch and some others. So we need to import the libraries first.

```
import torch
import torchvision
import torchvision.transforms as transforms
```

This is the first step for all the other codes. And then here is some preprocess of the dataset. Because pictures can be various, you need to preprocess and normalize the pictures to make them have better properties, like the zero mean and small variance, for better interpretation.

```
transform = transforms.Compose(
    [transforms.ToTensor(), # Transform the picture to the
                              torch tensor
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
                               # and then normalize it
```

Then we define a training set using the *'CIFAR10'* function since CIFAR10 is a default dataset in PyTorch. The *'root'* is the folder to which we want to store the data. Since we are using the training set, set *'train'* is true. And because we haven't download it before, set *'download'* is true. *'transform=transform'* means we want to transform these pictures following the *'transform'* function that we defined in the preprocess part.

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=
                                        True, download=True, transform=
                                        transform)
```

Now we have already defined our training set. But there are still some problems. The dataset is too large to load them all to compute the gradient on the full batch. That is why we choose the stochastic gradient descent. So we choose a small batch (mini-batch) to calculate the corresponding gradient. Here we set the batch size to be 4. So we can compute the loss and the gradient very conveniently on each step.

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size
                                          =4, shuffle=True, num_workers=2
                                          )
```

It is the same for the test set except that the *'train'* and *'shuffle'* are false.

```
testset = torchvision.datasets.CIFAR10(root='./data', train=
                                       False,download=True, transform=
                                       transform)
```

```
testloader = torch.utils.data.DataLoader(testset, batch_size=4
                                         ,shuffle=False, num_workers=2)
```

Here are just some parts to show the images. If you are interested about what the images look like, you can use the *'matplotlib'* library to plot them. If not, just skip this part.

```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image

def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

That's basically how you load a dataset. Let's go to the second step.

Define a Convolution Neural Network

Since we are dealing with the image classification, we prefer to using the Convolution Neural Network. And we will introduce more details the part later. Now we just go through the codes for the simple example line by line. The first part is to initialize the class. Then we define how we compute a forward process by given an input as *'x'* in second part. First, the input will be convoluted with the 2-d convolution operation, then use the ReLU function as the nonlinear activation function, after that we make a pooling. This is what a convolution layer composed, first a convolution operator, then a nonlinear activation, and then a pooling. We define two convolution layers this way. After that, the function *'x.view'* is used to flat the picture, because the output of the convolution layer is still a two dimensional picture. Here, we use the *'view'* function to flat the two dimensional picture to be one dimensional of length 400. Then we make a fully connect operator and a ReLU function. After that, we have the final classification layer to output the prediction outcome. So that is basically what the CNN composed.

```
import torch.nn as nn
import torch.nn.functional as F
```

```
class Net(nn.Module):
    def __init__(self):  # Initialize the class
        super(Net, self).__init__()
        # the construction of the neural network
        self.conv1 = nn.Conv2d(3, 6, 5) # a 2-d convolution
                                  layer named as ‘conv1’
        self.pool = nn.MaxPool2d(2, 2) # a max pooling layer
                                  named ‘pool’
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120) # linear layer
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

Different choices of these hyper-parameters may lead to different results. We usually need to tune these hyper-parameters to get a better model. Since the input of each layer in the neural network is the output of its previous layer, there are two parts of hyper-parameters of an operator, like *‘Conv2d’*, *‘Linear’*. One part depends on the input, which is the output of the previous layer, of this operator. The other can be arbitrarily choose to get a good model, and this part may decide a part of parameter of the next operator/layer. For example, ‘Linear(input dimension, output dimension)’, the dimension of output of *‘fc1’* is 120, then the first parameter of *‘fc2’* should be same as it, 120.

Another important hyper-parameters of convolution layers is channel. For *‘Conv2d (number of input channels, number of output channels, kernel size)’*, we need keep the number of input channel same as the number of output channel of the previous layer.

To choose parameters (called hyper-parameters in neural network) of the torch functions, we recommend to read the corresponding documents in PyTorch website for more precise details.

Here we just give a simple working example. Later on, we will talk about more details and how to modify them.

Define a loss function and optimization

There are actually a lot of possible loss functions. For image classification, the common choice is the cross-entropy. You can also use the square loss or any reasonable functions based on the problems. Actually, the common loss functions have already been defined in the *'torch.nn'* library. The way to use them is like the *'criterion'* in the following code. And the *'optimizer'* is the corresponding optimization algorithm used to decrease the loss function. The common algorithms can also be found in the *'torch.optim'* library. In the example, we choose stochastic gradient descent (SGD). And there are a few optional parameters, like learning rate *'lr'*, momentum. For large learning rate, you may behave very widely, but for small learning rate, you may zig-zaging around very small range. The learning rate is one of the most important hyper-parameter that we need to tune to get a good model.

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss() #use the cross-entropy loss
                                   defined in torch library
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9
                                   )
```

Now we have almost defined everything. Then we can start to train our networks.

Train the network

For the training, we first define what the epoch means. Because we have like 60,000 pictures in this training set, and we divided it into a few small batches (mini-batch). Whenever you go over all the pictures in the training set, then it means you finish an epoch. And when we do one optimization step (calculate gradient and update the parameters) in one mini-batch, we say it an iteration.

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0): # i is the index
                                          and data are pictures in
                                         the corresponding mini-
                                         batch
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs) # forward propagation to get the
                                              output
        loss = criterion(outputs, labels) # compute the loss,
                                        see if it satisfy the
                                        terminal criterion
        loss.backward() # backward propagation to get the
                                        gradient
        optimizer.step() # update the parameters by
                                        optimization algorithm
                                        (like SGD) to decrease
                                        the loss

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-
                                        batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training') # After 2 epochs, we finish the
                                training.
```

Now, after training process, we want to see how well the model behave. So let’s start to test the model we got on the test set, which is independent to the training.

Test the network on the test data

```
correct = 0
total = 0
with torch.no_grad(): # We don't need compute the gradients in
                                 the test process since we don'
                                t need optimization
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1) # get the
                                        predicted class by
                                        neural network
        total += labels.size(0)
        correct += (predicted == labels).sum().item() #
                                        compare the predicted
                                        and the real label

print('Accuracy of the network on the 10000 test images: %d %%
                                ' % (100 * correct / total))
```

Now we finish training a simple neural network here.

1.2 CIFAR10 example

In this section, we will give more details about training a neural network with PyTorch. We will start with the simple CIFAR10 example introduced before, and explain how to modify each component of it.

The first of all, the first six lines here just consist the *import* statements. Basically all this does is bringing code from this imported libraries from PyTorch in your program allows to use the classes, the functions that for implement each of these libraries.

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

1.2.1 The *‘Net’* class

Then each neural network we should to implement should exactly contain the class in the following. Which you does is that you define a new class called Net, which is a subclass of nn.Module. The nn.Module is a class implemented in the package torch.nn. Bascially you don’t have to know exactly what the Module class does, but the point here is Network is going to be a subclass of it. Let go to it step by step.

Any class defined in Python has to have the *__init__* function. It will run when the class be created. Late on, in the code when you implement like *‘conv_net = Net()’*, it will create an instance of the Net class. And when the instance is created, the *__init__* function is run. And we can see that the *__init__* function create each of the pieces of the neural network, which are the operators we need for our neural network. This linear (*nn.Linear*), convolution (*nn.Conv2d*) and max pool (*nn.MaxPool2d*) is implement in somewhere else. Roughly speaking, they simply contain how much parameters and a forward function. For example the *‘nn.Linear’* class:

- contains parameters which consists from a matrix and a vector (W, b),
- and the ‘forward’ function is

$$\text{forward}(x) = Wx + b.$$

Later on, we will talk about what *nn.Conv2d* and *nn.MaxPool2d* do.

And then to make is useful, we need to define this forward function. Basically, it tells you, given this neural network and some inputs, what the output is. In this particular case, what the network does is to apply the first convolution, and a ReLU function. The ReLU function is

$$\text{ReLU}(x) = \max\{0, x\}.$$

And then to apply a max pooling. Then applies the next convolution, a ReLU function and pooling layer. Then, what the *'x.view'* does is take *x*, which is still an image (2 dimensional) and flat it out into a vector. Then applies the first linear layer, a ReLu function, and the second linear layer, a ReLU function, and finally the third linear layer. And it returns *x*. What is contained in the class is all the parameters for each of these layers and the forward function which tells you the order and how to apply them. So that is what network consist.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

1.2.2 The 'main' function

What we want to do is to load a bunch of data, and train this network (all the parameters) on these data. So the *main* function is going to do that. The two lines *trainset* and *trainloader* load the train set, and the two lines *testset* and *testloader* load the test set. The *trainloader* is a class which specifies some way of giving you the data. In our particular case, we construct the *trainloader*, we pass the *trainset*, the images, to it and also a bunch of the parameters, like the batch size, shuffle, number of workers. When you calculate the gradient, you don't use the whole dataset at once, and we only use some of them. The number of data we use to calculate the gradient is the batch size. Here we pass the *'batch_size'* to the *DataLoader* what happen is the later on when we loop over everything in the *trainloader* (see the for loop in the main function), it will return the images four at a time because we set 'batch_size=4'. The shuffle is true means that you return the data without replacement. It the same thing to the test set.

The *criterion* is a class containing the loss function. Given the output of the neural network and the true label applies the loss function. The optimize class, we will talk about it later on. When you construct the *'optim'* class, the parameters you have to pass are *‘net.parameters’* that tell the optimizer which parameters need to be optimized, and other hyperparameters. It has a function called *‘step’*. Whenever the *‘step’* function is called it going to modify the parameters in some way based on whatever the gradients are. The optimizer assume you already calculated the gradients and then it does some sort of step based on the gradients. That explains why when you actually run the training we have to call *‘optimizer.zero_grad’* and *‘ loss.backward’* independently of *‘optimizer.step’*. *‘optimizer.step’* doesn’t handle taking the gradient, it assumes that all the gradients already been taking properly.

```
def main():
  transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5,
                                 0.5), (0.5, 0.5, 0.5))])
  trainset = torchvision.datasets.CIFAR10(root='./data', train
                                =True,download=True, transform=
                                transform)
  trainloader = torch.utils.data.DataLoader(trainset,
                                batch_size=4, shuffle=True,
                                num_workers=2)
  testset = torchvision.datasets.CIFAR10(root='./data', train=
                                False, download=True, transform
                                =transform)
  testloader = torch.utils.data.DataLoader(testset, batch_size
                                =4, shufflex=False, num_workers
                                =2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

conv_net = Net()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9
                                )

for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-
                                       batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')

correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%
                               ' % (
    100 * correct / total))
```

1.2.3 *'DataLoader'*

Next, we want to talk about how can we change the Dataloader and how can we make it provide the data samples in a different way. In particular, we already talk about the difference between sampling with and without replacement. Let us recall that a little bit.

The loss function:

$$L(\theta) = \sum_{i=1}^{n} l(x_i, \theta)$$

we want to calculate the gradient of the loss function.

$$\nabla L(\theta) = \sum_{i=1}^{n} \nabla_\theta l(x_i, \theta)$$

and n generally could be very large, $n \sim 10^4 - 10^7$. In this situation, to compute the gradients over all the data point is computationally not feasible. So just approximate the gradients by sampling some of the data points. Initially, consider the computation complexity. Stochastic gradient descent

- approximate the gradient by considering a sample of data points

  $$\{x_{i_1}, \ldots, x_{i_k}\}$$

  where $\{i_1, \ldots, i_k\}$ is randomly chosen in each iteration, then

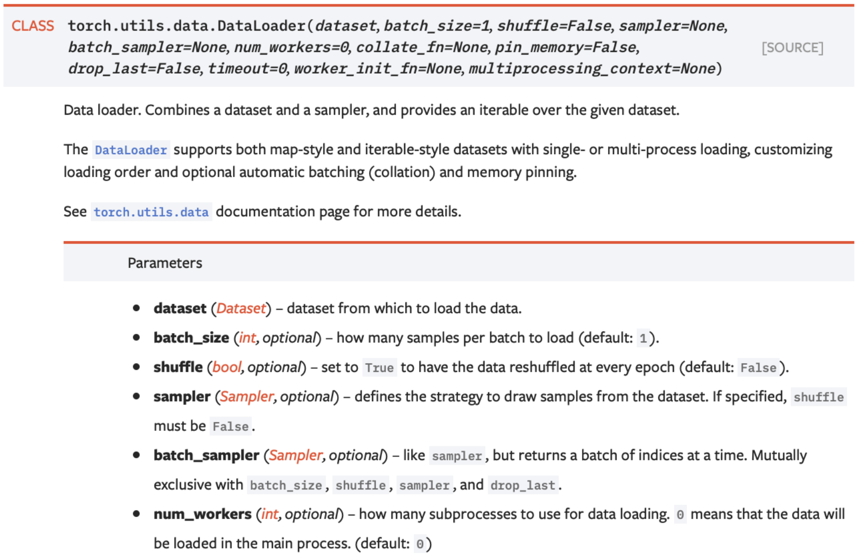
  $$\nabla L(\theta) \approx \sum_{l=1}^{k} \nabla_\theta l(x_{i_l}, \theta)$$

- k is called the mini-batch size (controls the accuracy of ”noise” in the sample). This is a sampled gradient, containing some noise. If I have larger mini-batch size, we will get a more accuracy gradients, so the mini-batch size k is an important hyper-parameter.
- $i_1, \ldots, i_k$ sampled with/without replacement.
  - with replacement: there can be repetitions
  - without replacement: there can’t be repetitions

What we implement is sampling without replacement cross the whole epoch. When we randomly choose these data points, we can’t choose the same data points again until we around the whole data set.

Now, let’s see how we can change it. First, go to the PyTorch documents page[6]. Here is an explanation what exactly is this class does, and the important thing for us is what the possible inputs. So when I construct a DataLoader, I need to pass some variables. Whenever we construct a DataLoader, we have to pass it a dataset because there is no default value. In our program, we passed the train set, CIFAR10. You also can pass something else you want to it. And all the other variables here have some default values. There are options here to keep the defaults or to change it to what you want We can see that some of the defaults actually already changed in our program. Instead of the default value 1, we set the batch size as 4, and shuffle as true instead of false. Now, we know how to change the sampling strategies. We probably want to change some of these other inputs. Look here, we can see the description are. In particular, you will see sampler, the default value be none, which defined the strategy. Most likely, if we construct the samlper and pass it to the DataLoader, we can change the with and without replacement strategy.

[6] https://pytorch.org/docs/stable/data.html



CLASS `torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, sampler=None, batch_sampler=None, num_workers=0, collate_fn=None, pin_memory=False, drop_last=False, timeout=0, worker_init_fn=None, multiprocessing_context=None)` [SOURCE]

Data loader. Combines a dataset and a sampler, and provides an iterable over the given dataset.

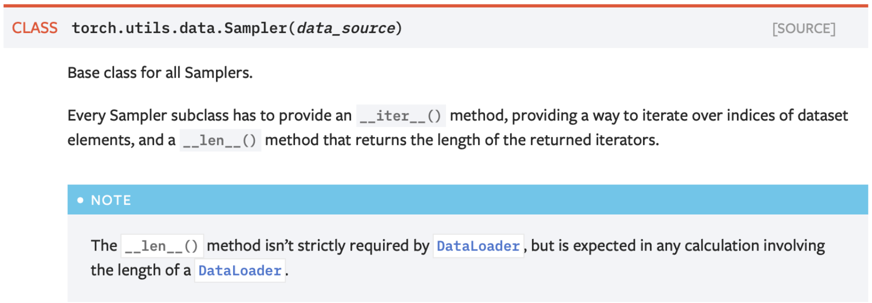
The `DataLoader` supports both map-style and iterable-style datasets with single- or multi-process loading, customizing loading order and optional automatic batching (collation) and memory pinning.

See `torch.utils.data` documentation page for more details.

Parameters

- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: `1`).
- **shuffle** (*bool, optional*) – set to `True` to have the data reshuffled at every epoch (default: `False`).
- **sampler** (*Sampler, optional*) – defines the strategy to draw samples from the dataset. If specified, `shuffle` must be `False`.
- **batch_sampler** (*Sampler, optional*) – like `sampler`, but returns a batch of indices at a time. Mutually exclusive with `batch_size`, `shuffle`, `sampler`, and `drop_last`.
- **num_workers** (*int, optional*) – how many subprocesses to use for data loading. `0` means that the data will be loaded in the main process. (default: `0`)

Now, we need to look at the sampler class. If we want to change the sampler, we need to define a class that is a subclass of the sampler, and contain the sampler strategy.



CLASS `torch.utils.data.Sampler(data_source)` [SOURCE]

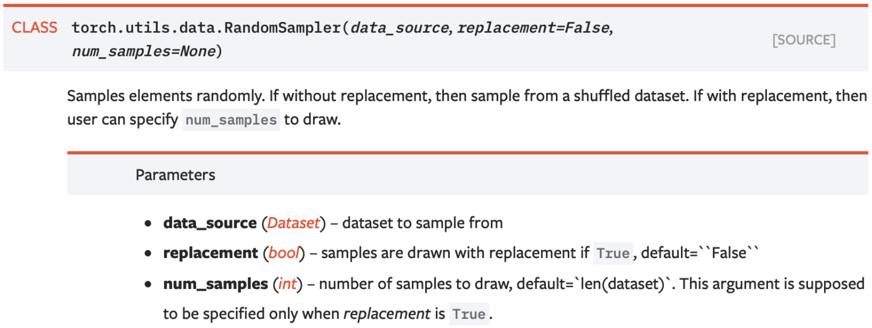
Base class for all Samplers.

Every Sampler subclass has to provide an `__iter__()` method, providing a way to iterate over indices of dataset elements, and a `__len__()` method that returns the length of the returned iterators.

• NOTE

The `__len__()` method isn't strictly required by `DataLoader`, but is expected in any calculation involving the length of a `DataLoader`.

In our case, if you want change that with or without replacement, somebody has already written it. It turns out the RandomSampler.



CLASS `torch.utils.data.RandomSampler(`*`data_source, replacement=False, num_samples=None`*`)` [SOURCE]

Samples elements randomly. If without replacement, then sample from a shuffled dataset. If with replacement, then user can specify `num_samples` to draw.

Parameters

- **data_source** (*Dataset*) – dataset to sample from
- **replacement** (*bool*) – samples are drawn with replacement if `True`, default=``False``
- **num_samples** (*int*) – number of samples to draw, default=`len(dataset)`. This argument is supposed to be specified only when *replacement* is `True`.

Let's use our program as an example to show how to use a sampler class. First, we need to create an instance of the RandomSampler class and pass it to the DataLoader as the sampler variable.

```
  transform = transforms.Compose([transforms.ToTensor(),
                                 transforms.Normalize((0.5, 0.5,
                                  0.5), (0.5, 0.5, 0.5))])
  trainset = torchvision.datasets.CIFAR10(root='./data', train
                                 =True,download=True, transform=
                                 transform)
  with_replacement_sampler = torch.utils.data.RandomSampler(
                                 trainset, replacement=True)
  trainloader = torch.utils.data.DataLoader(trainset,
                                 batch_size=4, shuffle=True,
                                 sampler=
                                 with_replacement_sampler,
                                 num_workers=2)
```

Now if we run this code, the data will be sampled with replacement. This is just to illustrate how you go about changing something about the model and the training process. You just look up the classes, look up in the documentation, see what they do. And often the most of thing that you may want to do, someone already program them in a simple way to do them. You can just turn on like we just doing it. Just notice that you don't have to implement it yourself in this case. You just need to look out these classes do and figure out which tools for use. So we recommend just for fun just try to the CIFAR10 example with and without replacement to see difference.

Codes for the CIFAR10 example

```
# cifar10_example.py

import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

def main():
  transform = transforms.Compose([transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5,
                                0.5), (0.5, 0.5, 0.5))])
  trainset = torchvision.datasets.CIFAR10(root='./data', train
                               =True,download=True, transform=
                               transform)
  trainloader = torch.utils.data.DataLoader(trainset,
                               batch_size=4, shuffle=True,
                               num_workers=2)
  testset = torchvision.datasets.CIFAR10(root='./data', train=
                               False, download=True, transform
                               =transform)
  testloader = torch.utils.data.DataLoader(testset, batch_size
                               =4, shuffle=False, num_workers=
                               2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

conv_net = Net()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9
                               )
```

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-
                                       batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')

correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%
                               ' % (
    100 * correct / total))
```

1.2.4 The training part of CIFAR10 example

In this section, we will analysis the training part code in the CIFAR10 example.

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-
                                        batches
            print(’[%d, %5d] loss: %.3f’ %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print(’Finished Training’)
```

In particular, we will see what the *loss.backward* and *optimizer.step* do

loss.backward

Let’s start with the automatically differentiation of PyTorch. This function *loss.backward* hides some pretty complicated code that automatically figure out how to calculate the gradient w.r.t. the output. Let’s start with some simple example.

```
import torch
from torch.autograd import Variable
x = Variable(torch.randn(3,3), requires_grad = True)
```

Variable is a fundamental data type in PyTorch, which contains a tensor and some other things, like a flag in our example called *requires_grad*. So if I set *requires_grad* is true, it just indicating to PyTorch that in the future whatever x is involved in (computational graph), it going to calculate the gradient of x. Also, one of the other things the Variable contains is the gradient value. Which right now should be empty. You can check it by running *x.grad*. Now let’s see what happen when we use this Variable x in some calculations.

```
y = torch.sum(x)
```

Here, *torch.sum* means all the values of x. Here, we create a new variable y which consists of the sum of the entries of x. And PyTorch allows us to automatically calculate the gradient of the output w.r.t. any inputs. This is an extremely simple example.

$$\begin{aligned}
x &= \begin{pmatrix} * & \cdots & * \\ \vdots & \ddots & \vdots \\ * & \cdots & * \end{pmatrix} \\
y &= \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}^{\top} x \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \sum_{i=1}^{3}\sum_{j=1}^{3} x_{ij} \\
\frac{\mathrm{d}y}{\mathrm{d}x} &= \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}
\end{aligned} \tag{1.1}$$

How to let PyTorch compute this. Simply call

```
torch.autograd.grad(y,x)
```

This function automatically calculate the derivative of the second augment w.r.t the first. And other way, which is more commonly used, is to call

```
y.backward()
```

This function will calculate the derivative of y w.r.t everything that depends on the requirements of gradients (see the flag *requires_grad*). In our case, it will calculate the gradient of y w.r.t. x and store the gradient value in *x,grad*. These are two common ways that let PyTorch to calculate the gradients. We just give a simple example with one input and one output, but this whole thing works even for much more complicated examples with multiple input variables. And PyTorch can automatically calculate the gradients of these variables.

And there is one important thing to note is now if I call the *y.backwark* again, what happen to the *x.grad*? It will keep the same? or it will be changed? The answer is, unfortunately, it adds the gradient to whatever is already stored in the *x.grad*. So, this explains why we have to zero the gradients of all of the parameters (see *optimizer.zero_grad*) before we call *loss.backward* in our codes.

optimizer.step

The *loss.backward* step calculates the gradients of all of our parameters and stores in *.grad* fields. Now, let's look at our *optimizer*. This *optimizer* may be a class you want to write yourself if you want to try some new algorithm. It will help to know how it works. Notice that the function we are calling here is *optimizer.step*. For example, we set optimizer as SGD. After we calculate the gradients of all of the parameters, then we call this step function, it will add the values of the multiplication of learning rate and negative gradients to current values of the parameters. The *optimizer* class assume that gradients already been calculated and stored in *.grad* fields, and the class just decide what to do with the gradients.

Now, let use Logistic regression as an example.

```python
# lr_example.py

import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim


class LogisticRegression(nn.Module):
    def __init__(self):
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(10, 5)

    def forward(self, x):
        return self.linear(x)
```

We just write a logistic regression class with 10 input features and 5 output features. Then we will use this example to see what a the network class exactly contains. Run the following codes in python terminal

```python
import torch
from lr_example import LogisticRegression

net = LogisticRegression()
x = torch.randn(10)
y = net(x)
```

Here we create an instance of LogisticRegression class, and set x as input and y as output. The variables this class contains is the parameters which is a 10-by-5 matrix and a vector as the bias. And take a vector x as input, and apply the forward step (see *net(x)*, 'forward' can be dropped), it will run the forward function to calculate $Wx + b$.

There is one thing have to know is that when you call *backward* on something has to be a scalar. It can't be for example a tensor. For example, if I try to call *y.backward*, it will be given an error, because you can only call this for scalar outputs. It won't to calculate a Jacobian matrix of a vector output w.r.t. a vector input. However, if you pass it the derivatives of each of this components, then it can continuous do the *.backward* function. So, for example, if we we want to calculate the derivative of the sum of y, we can do it as follows:

```python
y.backward(torch.ones(5))
```

If you have a vector that you want to call *.backward* on, you have to pass it a vector with the same size, it gives the gradients of each of the components of that vector w.r.t. the output. Actually, it calculate the Jacobian times the vector. In general, it can't calculate the Jacobian of a non-scalar output, but if you know what you want to multiply the Jacobian with, then you can pass it to *.backward* as well.

Then we can see the parameter of a network

```
params = list(net.parameters())
```

Now *params* contains matrix W and vector b. Also, we can see the gradient values of them.

```
params[0].grad
params[1].grad
```

Here we give an explanation about it. For $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^d$, y is some function of x.

$$y = f(x)$$

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \begin{pmatrix} \frac{\mathrm{d}y_1}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}y_d}{\mathrm{d}x_1} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}y_1}{\mathrm{d}x_n} & \cdots & \frac{\mathrm{d}y_d}{\mathrm{d}x_n} \end{pmatrix} \tag{1.2}$$

In this case, *y.backward()* will return an error, but we can call *y.backward(v)* for some vector $v \in$. Then it will calculate

$$\frac{\mathrm{d}y}{\mathrm{d}x} \cdot v = \begin{pmatrix} \frac{\mathrm{d}y_1}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}y_d}{\mathrm{d}x_1} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}y_1}{\mathrm{d}x_n} & \cdots & \frac{\mathrm{d}y_d}{\mathrm{d}x_n} \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix} = \nabla(y \cdot v) \tag{1.3}$$

This is basically because if you want to compute the whole Jacobian matrix, it requires a lot of time and space and it always not necessary.

Maybe this will make more sense if I set another variable z as the sum of y,

```
net.zero.grad()
y = net(x)
z =torch.sum(y)
```

In our example, we can see z is a function of y and $z = g(y) = y \cdot v$. If I take *z.backward*, this is the same computation with the previous did. Because the derivatives of z w.r.t. y is $\frac{\mathrm{d}z}{\mathrm{d}y} = (1, 1, 1, 1, 1)^\top$. This is what the *.backward* does. It calculate the gradients multiplied by the Jacobian of previous layer to get the previous gradients, and so on. You can call

```
z.backward()
params[0].grad
params[1].grad
```

and compare the output with the previous one.

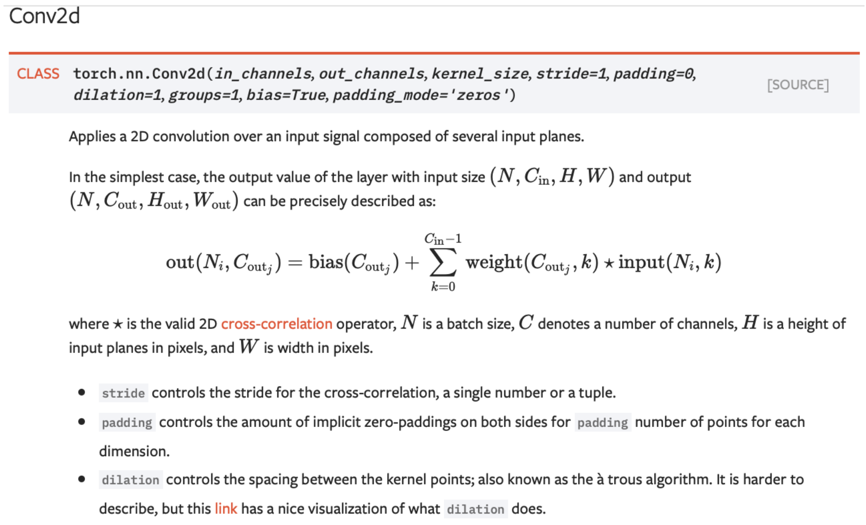
In practice, we would construct some loss function, and call *backward* on it. Particular, in our cifar10 example, it consists of a bunch of parameters, particular, each of these layers has its parameters, and when you call forward, it performs some complicated calculations. It will build some graph that contains all the dependencies of these calculations. And then, after you apply the network and apply a loss function to it, you call *loss.backward*, this step gives you all the gradients of the parameters, and then this is a special optimizer class, which takes those stored gradients and use them to perform like a forward step. The most important thing to remember are don't forget the *optimizer.zero_grad* because the *backward* step adds the new calculated gradients to that already stored in the *grad* field. Of course, if you have two loss functions, called *loss1* and *loss2*, you can call

```
loss1.backward()
loss2.backward()
```

and gradients now is the sum of those two gradients.

1.3 Convolutions

In this section, we will talk about the convolutions, and we will focus more about implementation of it in PyTorch. To see the documentation of the *Conv2d* operator in PyTorch, there are a bunch of parameters, like number of channels, kernel size. We want to talk about what exactly those things mean here.



Conv2d

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

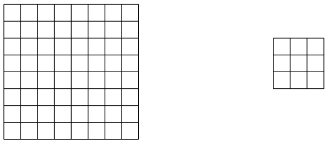
where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

- `stride` controls the stride for the cross-correlation, a single number or a tuple.
- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points for each dimension.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.

So let's first talk about the simplest case with 1 input channel and 1 output channel, which means 1 input and 1 output images.

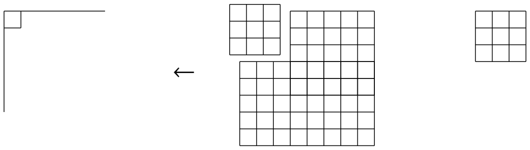
- kernel size

- stride
- padding
- dilation



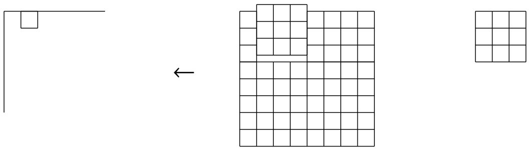
Input image I kernel K

Suppose, we have an n-by-n input image(i.e an n-by-n array of numbers), each of this box represents a real number, and a kernel, which is a much smaller array of numbers, usually 3-by 3 or 5-by-5. And the convolution $K * I$ will be a new image, called output image O. Now let's just immediately not worry about what size of this image is. We just talk about how to fill in its entries. Now we want to form this convolution w.r.t. the kernel K with the image I, what will be the top lefthand entry? How to determine it? Since we have a 3-by-3 kernel, we take the dot product of kernel K and the first 3-by-3 top left block of the image, and it gives the first top left entry value of the output image.



Output O Input I Kernel K

Similarly of the next entry in the right, we just shift the 3-by-3 block of I over by 1 to the right, and take the dot product of the kernel with it.



Output O Input I Kernel K

And the rest of output entries are similarly. Based on this, take the convolution of an n-by-n input image with a 3-by-3 kernel, and we can get the output image with dimension $(n-2)\times(n-2)$. And in this simplest case, the kernel size is 3×3 and stride and dilation are 1. And there is no padding. So what do these parameters mean?

- kernel size is the dimension of the kernel matrix, and it doesn't have to be square.
- stride tells us if we move one pixel in the output image, how far do I shift the kernel in the input image to get the dot product. (In our example, we shift 1 at a time.). And the large stride will give a small output.
- padding tells you how many zeros to add around the input images.
- dilation tells you, to take the dot product, when we move 1 in the kernel, how much do I move in the block of input image.

Notice that both of stride and padding, we can have different values in each directions. And there is a nice digit illustration of convolution https://github.com/vdumoulin/conv_arithmetic. One more thing have to mention is the convolutions also have a bias term. The bias term is just added to $K * I$. Now, what happens with more than one input/output channels. Suppose we have n input channels and m output channels. Then

- there will be $n \times n$ kernels, denoted by K_{ij}, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.
- input is n images, denoted by $I_1^{\text{in}}, \ldots, I_m^{\text{in}}$
- m output images, denoted by $I_1^{\text{out}}, \ldots, I_m^{\text{out}}$

Here,

$$I_i^{\text{out}} = \sum_{j=1}^{n} K_{ij} * I_j^{\text{in}}$$

Basically, densely connected across channels. Every channel talks to every channels via some kernels. And also, we have different bias term for every output channels. So this is how convolutional layers work, now let's look what happens when you construct the *Conv2d* class and what the weights look like. Suppose that we construct a convolution layer with 1 input channel, 1 output channel, kernel size with 3, and all the other augments are default, like the simplest example.

```
import torch
import torch.nn as nn
layer = nn.Conv2d(1,1,3)
parameters = list(layer.parameters())
```

We can call *parameters = list(layer.parameters())* to see that the parameters of the convolution layer we just constructed are a 3×3 kernel and a bias term. And in the simplest case, take $n = 10$, which means the input image is 10×10. We can check the dimension of the output channel is $(n-2) \times (n-2)$.

```
x = torch.randn(1,1,10,10)
layer(x).size()
```

It will return *[1,1,8,8]*. To add more channels,

```
layer = nn.Conv2d(2,2,3) # 2 input channels, 2 output channels and 3-by-3 kernel
parameters = list(layer.parameters())
```

Now, we can see the parameters for the new convolution layer contains 4 kernels with size 3 × 3 and 2 bias term.

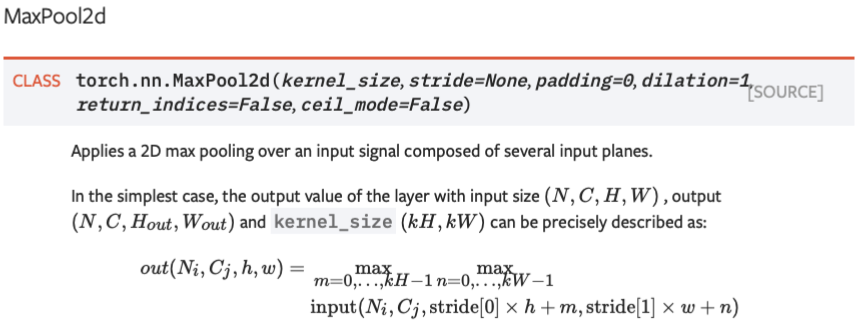
1.4 Max pooling

In this section, we will introduce max pooling. We already learned the multigrid methods and the relationship between convolution neural network and multigrid methods. When implementing multigrid methods, we have a sequence of coarse and fine grids. In CNN, you want images to get the grids from one to another, and people use max pooling. Input: An image I with multiple channels
Output: a smaller, "lower-resolution" image with the same number of channels

- Channels do not interact, pooling is done on each channel independently.

The parameters, kernel size, stride, padding and dilation, in max pooling play the same roles as in convolution.

- Notice that basically there will be no kernel here, because all this important is the size.



MaxPool2d

CLASS `torch.nn.MaxPool2d(`*`kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False`*`)` [SOURCE]

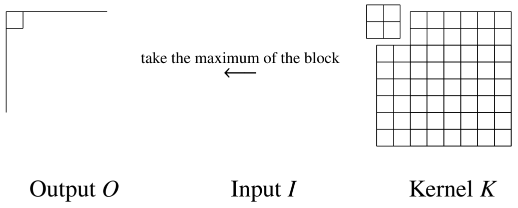
Applies a 2D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

Like the convolution operator, the max pooling with kernel size 2 works as Max pooling has the same parameters with the convolution. Basically, it is the same thing, instead of taking some convolution with the kernel, you just take the maximal value of these entries to get the corresponding entry of output. In particular, there is no model parameters associated with the max pooling layers.

And let's just talk about stride again. The PyTorch default for max pooling is to set the stride equal to the kernel size. So the default behavior is, if you kernel size is 2, to reduce the dimension in each direction by 2. It is similar to taking it to a coarse grid.



And let's start with creating a 6-by-6 torch tensor.

```
import torch
import torch.nn as nn
x = torch.randn(1,1,6,6)
```

The first two augments of x represent the batch size and the number of channels. So, let just create a max pooling layer.

```
max_pool = nn.MaxPool2d(2)
```

Now, if the max pooling is applied to x, then what is the dimension of *max_pool(x)*? The answer is 3×3 since the stride is default. And let's set the stride to be something different, say stride=1, what now?

```
max_pool = nn.MaxPool2d(2, stride=1)
```

Call *max_pool(x)* and see the difference. Now, the size of output will be 5×5. Because, we just shift the window over by 1 each time.

Actually, max pooling is a non-linear function, you can just construct a neural network consisting entirely convolution and max pooling, you don't need ReLU. And also a piecewise linear function, behaves similar to the ReLU. There is an example of max pooling convolution network.

```
# maxconv.py
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim


class MaxConvNet(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv_step = nn.Sequential(
        nn.Conv2d(1, 4, 2),
        nn.MaxPool2d(kernel_size=(2, 2), stride=(1,1)),
        nn.Conv2d(4, 8, 4),
        nn.MaxPool2d(kernel_size=(4, 4), stride=(1,1)),
        nn.Conv2d(8, 15, 4),
        nn.MaxPool2d(kernel_size=(4, 4)))
        self.dense_linear = nn.Linear(4 * 4 * 15, 10)

    def forward(self, x):
        x = self.conv_step(x)
        x = x.view(-1, 4 * 4 * 15)
        x = self.dense_linear(x)
        return x
```

Notice that there are no ReLU, but the max pooling introduce the non-linearity. That allows it to fit the data. We can create an instance of the network.

```
import torch
from maxconv import MaxConvNet
model = MaxConvNet()
```

And you can check that even it's a small network, it still has less than 5 thousand parameters.

```
sum(p.numel() for p in model.parameters())
```

And also can test its accuracy. You can try different activation function to see what will happen.

1.5 Dropout

This is an idea invented by Hinton, 2012. It is an integral component of AlexNet. And it can be thought of as a regularization which improve the generalization accuracy.

What is the idea? Consider the neural network model $f(x, \theta) = NN(x, \theta)$, where θ are the parameters in each layer. We try to learn a function f that is of the neural network form by optimizing a loss function

$$\min_{\theta} \; L(f(x, \theta)).$$

Idea behind dropout is to consider a whole family of network. Given a network, we randomly remove some neurons and all of the corresponding connections.

for each neuron, remove with probability p. produces a random new neural network let $f(x, \theta)$ = average of $\tilde{NN}(x, \theta)$ over all sampled network $\tilde{NN}$, in mathematical $\mathbb{E}_{\sigma}(\tilde{NN}(x, \theta))$. However we can not really calculate the expectation over the exponential large number of neural networks. To make this tractable, two approximations:

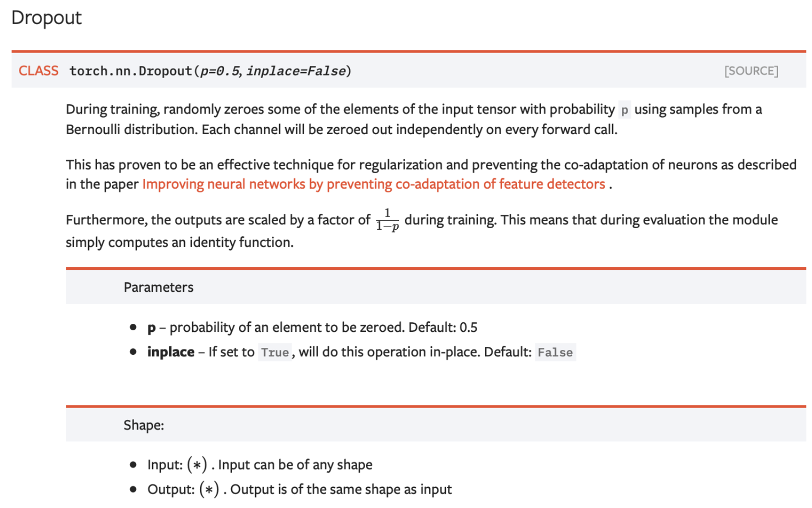
1. $\min_{\theta} \; L(\mathbb{E}_{\sigma}(NN_{\sigma}(x, \theta))) \approx \mathbb{E}_{\sigma} L(NN_{\sigma}(x, \theta))$
   Now we can use SGD by randomly sampling σ (i.e. a collection of neurons to remove) and performing SGD on the result smaller network. Don't update weights that were removed. (gradients at removed weights are 0)

2. When testing, replace $\mathbb{E}_\sigma(NN_\sigma(x, \theta))$ by moving the expectation into the weight. For each neuron, at the training time, multiply the output of the neuron by p.

In PyTorch, we can find there are two functions *model.train()* and *model.eval()* to set the neural network in training or evaluation modes. Now we see one reason is sometimes we need different operations in these two different process.

So I recommend adding dropout in you network, seeing what happens. The idea is because you average all of the different neural networks. What if these neurons die, and you network still works well. Looking at the average of a bunch of pruned networks, makes this process more robust to the generalization accuracy. And in practice, dropout works very well.

Now, let's see how to implement the dropout and batch normalization. In PyTorch, we can create this dropout layers. Here we can see, how to construct it, really



Dropout

CLASS `torch.nn.Dropout(p=0.5, inplace=False)` [SOURCE]

During training, randomly zeroes some of the elements of the input tensor with probability `p` using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call.

This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper Improving neural networks by preventing co-adaptation of feature detectors .

Furthermore, the outputs are scaled by a factor of $\frac{1}{1-p}$ during training. This means that during evaluation the module simply computes an identity function.

Parameters

- **p** – probability of an element to be zeroed. Default: 0.5
- **inplace** – If set to `True` , will do this operation in-place. Default: `False`

Shape:

- Input: $(*)$. Input can be of any shape
- Output: $(*)$. Output is of the same shape as input

all you need to pass it is a single probability. That is the probability that the neurons in that layers is dropped out. For example, notice that the values that go into each neuron is the output of previous layer. If you apply a dropout, you just take the value x and pass it through the dropout layer before fill it in the next layer. The essential thing dropout does is take the input and set its entry to zero with probability p.

So all dropout does is when it see its input, it randomly set it to zero with probability p, and rescale everything that wasn't set to zero by $\frac{1}{1-p}$ in the training. And then during evaluation/testing, it doesn't do anything.

Now let's see how to add the dropout in our CIFAR example. We only need to modify the *Net* class part as follows:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
        self.dropout = nn.Dropout(p=0.3)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = self.dropout(F.relu(self.fc1(x)))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

We create a dropout layers in *__int__* and put it between the first and the second linear layers. Now in this network, before the second linear layer, dropout is applied. It's the same thing as removing randomly a bunch of neurons in the second layer. You can add this dropout layer anywhere else as well.

And when start training, you should be sure you model is set to the training mode, and in evaluation mode before testing. So

```
conv_net = Net()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9
                               )

net.train()

for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

...

net.eval()

correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

So, we need to add *net.train()* before training and *net.eval()* before testing. When we call *net.train()*, this cause all these dropout layers to get activate, and when we call *net.eval()* it turns off that dropout layers. And it the same for the batch normalization and other things that perform different in training and testing.

1.6 Batch Normalization

Similar to the dropout, Batch Normalization (BN) is also a "regularization" technique which improves generalization accuracy. Then idea of BN is to make the neuron outputs to be distributed "nicely". To achieve this, we shift and rescale the outputs at each neuron (during the training process) with the mean and standard deviation.

Given some $x_1, \ldots, x_k \in \mathbb{R}$ shift and rescale the x, so that mean = 0, standard deviation std = 1.

- subtract $\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$ from each point
- divide by $\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \mu)^2}$

And do this across each mini-batch.

Normalize the outputs of each neuron for all data points in a mini batch so these values have mean=0 and std=1. However, mean=0, std=1 might not be right (the real/best distribution might not be mean=1, std=1), so we introduce two new parameters for each neuron, one for shifting and one for rescaling, which control the actual mean, std of its output. There are two steps in the training process.

- Shift and rescale with the sample mean and std,
- and shift and rescale again with the new tunable parameters.

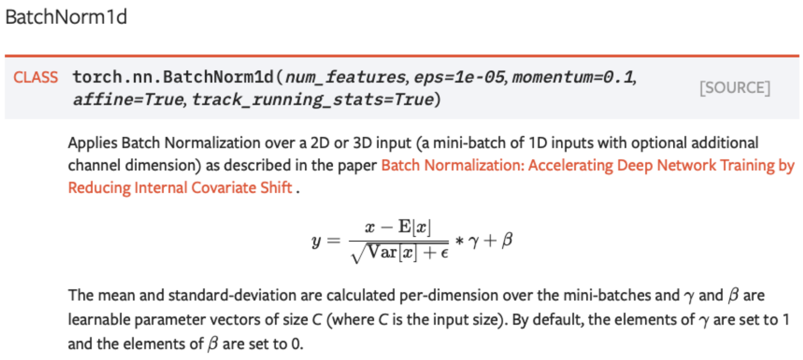
How do we evaluate the network after training? (Now we don't have mini-batches, the network will have one image as input at a time, so there are no mean and std. So we don't now how to do the first step.) In the evaluating/testing process,

- fix μ, σ to a weighted average of the values seen for each mini-batch during training, and shift and rescale with these two fixed values in the first step.

Notice that, as with dropout, the forward step(evaluating the model/network) is different during training and testing. So be sure to switch between training and eval mode!

1.6.1 Implementation for batch normalization

Let's see the batch normalization. There are three different type of batch normalization in PyTorch, for 1d, 2d and 3d data. If you have vectors, you would like to use *BatchNorm1d*



BatchNorm1d

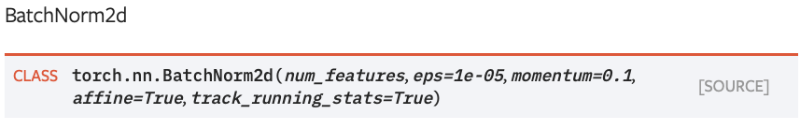
CLASS `torch.nn.BatchNorm1d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)` [SOURCE]

Applies Batch Normalization over a 2D or 3D input (a mini-batch of 1D inputs with optional additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ and β are learnable parameter vectors of size C (where C is the input size). By default, the elements of γ are set to 1 and the elements of β are set to 0.

and use *BatchNorm2d* for image/matrix.



BatchNorm2d

CLASS `torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)` [SOURCE]

And you will want to add it after your layers that will cause to normalize it and train. And you should, the same, think about to switch the training/evaluating mode. For example,

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
        self.dropout = nn.Dropout(p=0.3)
        self.bn_image = nn.BatchNorm2d(6)
        self.bn = nn.BatchNorm1d(84)

    def forward(self, x):
        x = self.bn_image(self.pool(F.relu(self.conv1(x))))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.bn(F.relu(self.fc2(x)))
        x = self.fc3(x)
        return x
```

And the parameters here. The number of features, what is explained in the documentation, is equal to the number of channels for *BatchNorm2d*, which should be 6 after the first convolution layer. Similar in the fully connected part, we want to take *BatchNorm1d*, and the parameter here is the dimension of the input vector, which is 84 after the second linear layer.

And of course, we need to use *net.train()* to set model to training mode before training process, and call *net.eval()* to set model to evaluating mode before test when batch normalization involved.

1.7 Multigrid Methods

1.7.1 Dual spaces

Definition: Given a vector space V, the dual space

$$V' = \{\varphi : V \to \mathbb{R} : \varphi \text{ is a linear map}\}$$

Here V' itself is also a vector space. It's easy to verify $(\varphi_1 + \varphi_2) : V \to \mathbb{R}$. (Because $(\varphi_1 + \varphi_2)(v) = \varphi_1(v) + \varphi_2(v)$).

Let $v_1, \dots, v_n \in V$ be a basis, define functions $\varphi_i : V \to \mathbb{R}$ by

$$\varphi_i(v_j) = \delta_{ij} := \begin{cases} 1, \ i = j \\ 0, \ i \neq j \end{cases} \tag{1.4}$$

Exercise: show that $\varphi_1, \dots, \varphi_n$ is a basis of V'.

Let's give some properties of the dual space here. Suppose $f : V \to W$ is a linear map. Let W' be the dual space of W, defined by

$$W' = \{\psi : W \to \mathbb{R}, \ \text{s.t. } W \text{ is linear}\}. \tag{1.5}$$

So $\psi \circ f : V \to \mathbb{R}$ is linear, which means $\psi \circ f \in V'$. Composition with f gives a map from W' to V'.

Furthermore, suppose $W \subset V$ is a subspace, a $\varphi \in V'$ is a function from V to $\mathbb{R}$, and is also a function from W to $\mathbb{R}$.

1.7.2 Multigrid methods

Consider to solving a linear system as follows

$$Au = \varphi \tag{1.6}$$

where $\varphi \in V'$, we are looking for $u \in V$, and $A : V \to V'$ is a linear map. Here V' is the dual space of V.

Now, we consider a method to solving the linear system (1.6). Notice that V is a very high dimensional, so the system is very difficult to solve. So we consider if we can approximate this equation by some lower dimensional system.

Consider a subspace $W \subset V$, then we can consider the system

$$Aw = \psi \tag{1.7}$$

where $w \in W$ and ψ equals to φ restricted to W. Basically, there is a restriction/projection operation $V' \to W'$.Now, we can choose some subspace W, and solve system (1.7) to get the approximation of the solution.

Abstract Algorithm

Consider a sequence of subspaces $W_1, W_2, \ldots, W_n \subset V$.

- Pick $v_0 \in V$, consider residual equation

  $$Av = r_0 = \varphi - Av_0 \tag{1.8}$$

- Restrict $Av = r_0$ to W_1.
- Solve it on W_1 and add the solution to v_0 to get v_1.

Example

We give an 1-d example here. Consider a 1-D problem.

$$\begin{aligned} -\frac{\mathrm{d}^2}{\mathrm{d}x^2} u &= f \\ u(0) = u(1) &= 0 \end{aligned} \tag{1.9}$$

Here we try to find a function defined on $[0, 1]$. The function values of endpoints are 0, and the second derivative is given.

The First thing we need to do is to formulate $A : V \to V'$. Choose a smooth function $g(x)$. Integrate both sides of (1.9) against g (view both sides in dual space).

$$\int_0^1 \left(-\frac{\mathrm{d}^2}{\mathrm{d}x^2} u(x)\right) \cdot g(x)\mathrm{d}x = \int_0^1 f(x)g(x)\mathrm{d}x \tag{1.10}$$

Using integration by part, the left hand side of (1.10) equals

$$\int_0^1 \left(\frac{\mathrm{d}}{\mathrm{d}x} u\right)\left(\frac{\mathrm{d}}{\mathrm{d}x} g\right) \mathrm{d}x \tag{1.11}$$

It gives

$$\int_0^1 \left(\frac{\mathrm{d}}{\mathrm{d}x} u\right)\left(\frac{\mathrm{d}}{\mathrm{d}x} g\right) \mathrm{d}x = \int_0^1 fg\mathrm{d}x \quad \forall g \tag{1.12}$$

But the above still can not be solve, because it is infinite dimensional. So we apply the restriction mentioned before from it to a finite dimensional subspace.

Restrict to a finite dimensional subspace

Divide $[0, 1]$ to n pieces, $0 = x_0 < x_1 < \ldots < x_n = 1$. Let

$$V = \{ \text{ Piecewise linear functions which are linear on } (x_i, x_{i+1}] \} .$$

And this is the simplest finite element space. We need to restrict u and g to space V.

Now we need bases of vector spaces V and V' which are dual. Let's talk first about the basis in V'. A good choice of the basis for the dual space V', functions from V to real numbers. Consider functions in V. We have $n+1$ different grid points, and the function is uniquely determined by its values at all these grid points. Take a particular point x_i, and evaluating in x_i.

$$\varphi_i(v) = v(x_i) \quad \text{and} \quad \varphi_i \in V' \tag{1.13}$$

So evaluation at one of these grid points is a dual vector. And particular because the function is uniquely determined by values on these grid points, this actually is the dual basis.d Now we obtain the dual basis. That $\{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ is a basis of V'. Then consider functions in V with the property that $v \in V$ such that

$$\varphi_i(v_j) = \begin{cases} 1, \ i = j \\ 0, \ i \neq j \end{cases} \tag{1.14}$$

So we get the 'hat' function v_i that is a piecewise linear function. and v_i equals one on the grid x_i and zero on the others. Then we have $\{v_1, v_2, \ldots, v_n\}$ is the dual basis of V. In order to turn (1.10) into an equation like (1.9). We want to represent u and $g$as a simple linear combination of $\{v_1, v_2, \ldots, v_n\}$:

$$u = \sum_{i=1}^{n} a_i v_i \tag{1.15}$$

And the right hand side of (1.10) should be a dual vector and equal to

$$\sum_{i=1}^{n} b_i \varphi_i, \tag{1.16}$$

that is

$$b_i = \int_0^1 f v_i \mathrm{d}x. \tag{1.17}$$

And we also need to figure out what $\int_0^1 \left(\frac{\mathrm{d}}{\mathrm{d}x} v_i\right)\left(\frac{\mathrm{d}}{\mathrm{d}x} v_j\right) \mathrm{d}x$ is. Notice that v_i is the 'hat' function. Then, it gives

$$\begin{pmatrix} \frac{2}{h} & -\frac{1}{h} & 0 & \cdots & 0 \\ -\frac{1}{h} & \frac{2}{h} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & -\frac{1}{h} & \frac{2}{h} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{1.18}$$

where $h = \frac{1}{n}$ is the length of each segmentation. So far, we take the integral equation (1.10) in the infinite dimensional vector space, and restrict it to a finite dimensional vector space. This is what people do to solve differential equations.

Now consider the multigrid method. We start with the equation in the finite dimensional subspace V, then we restrict it to a coarse space (a smaller subspace of V).

In this particular problem, let $W \subset V$ be the subspace of functions which are linear in $[x_{2n}, x_{2n+2}]$. Notice that we get coarse grids $0 = w_0 < w_1 < \ldots < w_{\frac{n}{2}}$. As before, we can construct two bases, like $\{\varphi_i\}$ and $\{v_i\}$. A basis of W'

$$\{\psi'_1, \psi'_2, \ldots, \psi'_{\frac{n}{2}}\} \tag{1.19}$$

and and the dual basis of W

$$\{w'_1, w'_2, \ldots, w'_{\frac{n}{2}}\}. \tag{1.20}$$

And w_i is also a 'hat' function which is piecewise linear and equal to one at grid x_{2i} and zero at the others x_{2j} with $j = 0, 1, \ldots, i-1, i+1, \ldots, \frac{n}{2}$. Now, we want to do the same restriction, as we done before, from V to W. Consider the right hand side

$$\sum_{i=1}^{n} b_i \varphi_i \in V' \tag{1.21}$$

So

$$\sum_{i=1}^{n} b_i \varphi_i(w_j) = \frac{1}{2} b_{2j-1} + b_{2j} + \frac{1}{2} b_{2j+1} \tag{1.22}$$

Then we get

$$\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix} \to \begin{pmatrix} - \\ b'_1 \\ - \\ b'_2 \\ \vdots \\ - \\ b'_n \end{pmatrix} \tag{1.23}$$

And the value of $b'_i = \frac{1}{2} b_{2i-1} + b_{2i} + \frac{1}{2} b_{2i+1}$. That how we get the right hand side of a small system from a large system. Notice this can be regard as the convolution with stride two. And the other thing is please to figure out, given a linear combination of the basis $\{w_1, \ldots, w_{\frac{n}{2}}\}$, how to get the coefficients of this linear combination w.r.t. the basis $\{v_1, \ldots, v_{\frac{n}{2}}\}$. That is the map from the small subspace W to the large space V.

1.8 Exercise

1.8.1 Problem 6 (Homework 3)

For $\phi \in \mathrm{DNN}_l$, ϕ is a continuous and piecewise linear function. $\mathbb{R}^d = \bigcup_i \bar{D}_i$, D_i is a polyhedron. ϕ is linear on D_i. Plot some of these functions for $d = 2$.

A neural network with ReLU activation functions produces a piecewise linear functions: $NN(x, \theta)$ is piecewise in x. Why is this a piecewise linear function?

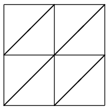
$$NN(x, \theta) = W_l\sigma(\cdots W_3\sigma(W_2\sigma(W_1x + b_1) + b_2) + b_3 \cdots) + b_l \tag{1.24}$$

Notice that

- linear map is piecewise linear,
- and ReLU function is piecewise linear.
- Composition of piecewise linear functions are piecewise linear. (Not trivial)

You should convince you self the third result is true. Being piecewise linear is the same thing that Hessian is zero or almost everywhere. The whole point of this exercise is to study the piecewise linear functions which are results from neural network process (1.24).

For example, Consider the grid as follows.



And consider functions that piecewise linear on each triangles. In finite element, we consider piecewise linear functions which are linear on a 'nice' set of polygon that tile the plane. Now we show how to implement it. There is a network we will plot.

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import matplotlib as plt

import numpy as np


class TestNet(nn.Module):
  def __init__(self):
    super(TestNet, self):__init__()
    self.function = nn.Sequential(nn.Linear(2, 25), nn.ReLU(),
                                nn.Linear(25, 25), nn.ReLU(),
                               nn.Linear(25, 1))
  def forward(self, x):
    return self.finction(x)

def main():
  net = TestNet()
  dy = 0.05
  dx = 0.05
  size = 400
  df_x = torch.zeros(size, size)
  df_y = torch.zeros(size, size)
  for i in range(size):
    for i in range(size)
      q = Variable(torch.Tensor([-1 + (i + 0.5) * dx, -1 + (j
                               + 0.5) * dy]), requires_grad=
                               True)
      output = net(q)
      df = torch.autograd.grad(output, q)
      df_x[i, j] = df[0][0]
      df_y[i, j] = df[0][1]
    df_x = (df_x - torch.min(df_x)) / (torch.max(df_x) - torch
                               .min(df_x))
    df_y = (df_y - torch.min(df_y)) / (torch.max(df_y) - torch
                               .min(df_y))
    df_color = torch.zeros(size, size, 3)
    df_color[:, :, 0] = df_x
    df_color[:, :, 1] = df_y
    plt.imshow(df_color)
    locs._labels = plt.xticks()
    locs = locs[1:]
    new_labels = []
    for i in locs:
      new_labels.append(-1.0 + i * dx)
    plt.xticks(locs, new_labels)
    plt.yticks(locs, new_labels)
    plt.show()
```

1.8.2 Tips for Final Project

In this section, we will introduce some tools that might be used in the final project. PyTorch has a library here call *torchvision.models*. PyTorch also implement all of the model in the list for you, so you don't need to build it yourselves.

- AlexNet
- VGG
- ResNet
- SqueezeNet
- DenseNet

- Inception v3
- GoogLeNet
- ShuffleNet v2
- MobileNet v2
- ResNeXt
- Wide ResNet
- MNASNet

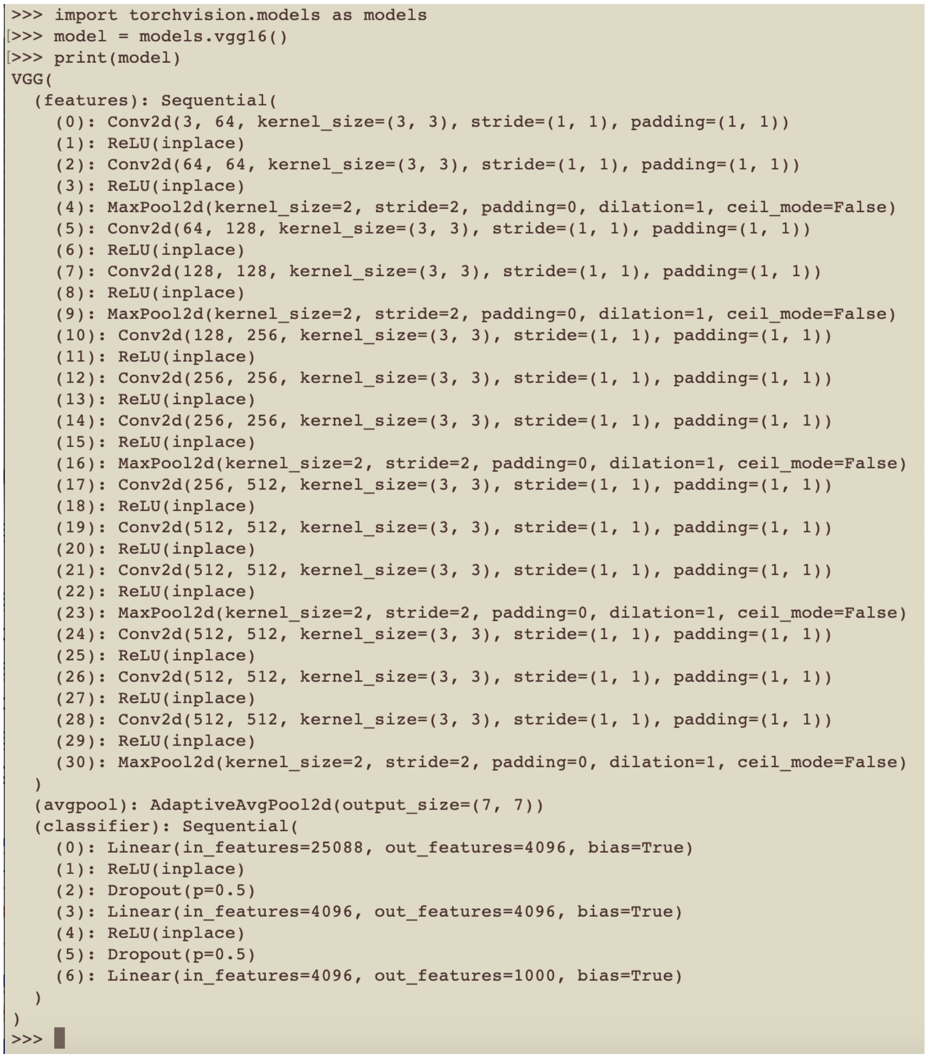
However, all of the models implemented here are built for ImageNet, which is a dataset consisting much bigger images than CIFAR10.

For example, if we look at one of the models, the VGG16 model. You can create a model:

```
import torchvision.models as models
model = models.vgg16()
```

You can use *print(model)* to see the list of the layers implemented. You may notice that this is pretty deep, and there is a lot of max pooling layers. Each of max pooling step will decrease the size of the image by factor 2. If you plug in the CIFAR10 image which is 32×32, then after a couple of these max pooling, it will be a single pixel. So the point is the model is too deep for CIFAR10. Because this model is implemented for ImageNet and too big, you can just take the model and implement it. But it's better to use this model as a template for VGG and make it smaller for CIFAR10. Use fewer channels in each convolution and use fewer layers.

The other model is the ResNet model. We can print the list of the layers in ResNet in the same way. This is a bit more complicated and also written for ImageNet. You should reduce number of channels and number of layers, and keep the same structure.



```
>>> import torchvision.models as models
>>> model = models.vgg16()
>>> print(model)
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
>>> 
```

References

[1] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456, 2015.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[3] Y. LeCun, L. Bottou, G. B. Orr, K.-R. Müller, et al. Neural networks: Tricks of the trade. *Springer Lecture Notes in Computer Sciences*, 1524(5-50):6, 1998.

[4] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

[5] A. Pinkus. Approximation theory of the mlp model in neural networks. *Acta numerica*, 8:143–195, 1999.

[6] S. Wiesler and H. Ney. A convergence analysis of log-linear training. In *Advances in Neural Information Processing Systems*, pages 657–665, 2011.