09.6 - Deep Learning - PyTorch

November 16, 2020

1 Programming for Data Science and Artificial Intelligence

1.1 9.6 Deep Learning - PyTorch

- [WEIDMAN] Ch7
- https://pytorch.org/tutorials/
- https://github.com/yunjey/pytorch-tutorial

Here we introduce PyTorch, an increasingly popular neural network framework based on **automatic differentiation**, which we introduced at the beginning of previous chapter.

As in the rest of the book, we'll write our code in a way that maps to the mental models of how neural networks work, writing classes for Layers, Trainers, and so on. I recommend you to watch the 60-min blitz to understand different features of pytorch (https://pytorch.org/tutorials/). Pytorch is really cool and simple! For example, there is a function tensor.to(cuda) so you can use the GPU to run any model which is much faster. I highly recommend checking it out.

1.1.1 Basics

1. Basic autograd example
2. Loading data from numpy
3. Input pipline

1. Basic autograd example 1

[1]:
```
import torch

# Create tensors.
# only tensors of floating point dtype can get gradient
x = torch.tensor(1., requires_grad=True)
w = torch.tensor(2., requires_grad=True)
b = torch.tensor(3., requires_grad=True)

# Build a computational graph.
y = w * x + b    # y = 2 * x + 3

# Compute gradients
# Pytorch tensor can automatically compute the derivative
```

```
# of the parameters in respect to loss
y.backward()

# Print out the gradients.
print("Gradient of x: ", x.grad)    # x.grad = 2
print("Gradient of w: ", w.grad)    # w.grad = 1
print("Gradient of b: ", b.grad)    # b.grad = 1
```

```
Gradient of x:  tensor(2.)
Gradient of w:  tensor(1.)
Gradient of b:  tensor(1.)
```

2. Loading data from numpy

[2]:
```
import numpy as np

# Create a numpy array.
x = np.array([[1, 2], [3, 4]])

# Convert the numpy array to a torch tensor.
y = torch.from_numpy(x)

# Convert the torch tensor to a numpy array.
z = y.numpy()
```

3. Input pipeline

[3]:
```
import torchvision
import torchvision.transforms as transforms

#put this for puffers.cs.ait.ac.th
#os.environ['http_proxy'] = 'http://192.41.170.23:3128'
#os.environ['https_proxy'] = 'http://192.41.170.23:3128'

# Download and construct CIFAR-10 dataset.
train_dataset = torchvision.datasets.CIFAR10(root='data',
                                             train=True,
                                             transform=transforms.ToTensor(),
                                             download=True)
```

```
Files already downloaded and verified
```

[4]:
```
# Fetch one data pair (read data from disk).
image, label = train_dataset[0]
print (image.size())
print (label)
```

```
torch.Size([3, 32, 32])
6
```

[5]:
```
# Retrieve batch of data
# Data loader (this provides queues and threads in a very simple way).
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=64,
                                           shuffle=True)
# When iteration starts, queue and thread start to load data from files.
data_iter = iter(train_loader)

# Mini-batch images and labels.
images, labels = data_iter.next()

print(images.size())
print(labels.size())
```

```
torch.Size([64, 3, 32, 32])
torch.Size([64])
```

[6]:
```
# Actual usage of the data loader is as below.
for images, labels in train_loader:
    # Training code should be written here.
    pass
```

1.1.2 Linear Regression

Let's have linear regression as a case study to study the different components of pyTorch. You will fall in love with it (at least, compared to writing from scratch!). These are the following components we will be covering:

1. Specifying input and target
2. Dataset and DataLoader
3. nn.Linear (Dense)
4. Define loss function
5. Define optimizer function
6. Train the model

Consider this data:

n a linear regression model, each target variable is estimated to be a weighted sum of the input variables, offset by some constant, known as a bias :

$$yield_{apple} = w_{11} * temp + w_{12} * rainfall + w_{13} * humidity + b_1$$

$$yield_{orange} = w_{21} * temp + w_{22} * rainfall + w_{23} * humidity + b_2$$

Visually, it means that the yield of apples is a linear or planar function of temperature, rainfall and humidity:

The learning part of linear regression is to figure out a set of weights w11, w12,… w23, b1 & b2 by looking at the training data, to make accurate predictions for new data (i.e. to predict the yields for apples and oranges in a new region using the average temperature, rainfall and humidity). This is done by adjusting the weights slightly many times to make better predictions, using an optimization technique called gradient descent

1. Specifiying input and target

[7]:
```
# Input (temp, rainfall, humidity)
x_train = np.array([[73, 67, 43], [91, 88, 64], [87, 134, 58],
                    [102, 43, 37], [69, 96, 70], [73, 67, 43],
                    [91, 88, 64], [87, 134, 58], [102, 43, 37],
                    [69, 96, 70], [73, 67, 43], [91, 88, 64],
                    [87, 134, 58], [102, 43, 37], [69, 96, 70]],
                   dtype='float32')

# Targets (apples, oranges)
y_train = np.array([[56, 70], [81, 101], [119, 133],
                    [22, 37], [103, 119], [56, 70],
                    [81, 101], [119, 133], [22, 37],
                    [103, 119], [56, 70], [81, 101],
                    [119, 133], [22, 37], [103, 119]],
                   dtype='float32')

inputs = torch.from_numpy(x_train)
targets = torch.from_numpy(y_train)
print(inputs.size())
print(targets.size())
```

```
torch.Size([15, 3])
torch.Size([15, 2])
```

2. Dataset and DataLoader We'll create a TensorDataset, which allows access to rows from inputs and targets as tuples, and provides standard APIs for working with many different types of datasets in PyTorch.

[8]:
```
from torch.utils.data import TensorDataset
```

[9]:
```
# Define dataset
train_ds = TensorDataset(inputs, targets)
train_ds[0:3]
```

[9]:
```
(tensor([[ 73.,  67.,  43.],
         [ 91.,  88.,  64.],
         [ 87., 134.,  58.]]),
 tensor([[ 56.,  70.],
         [ 81., 101.],
         [119., 133.]]))
```

The TensorDataset allows us to access a small section of the training data using the array indexing notation ([0:3] in the above code). It returns a tuple (or pair), in which the first element contains the input variables for the selected rows, and the second contains the targets.

We'll also create a DataLoader, which can split the data into batches of a predefined size while training. It also provides other utilities like shuffling and random sampling of the data.

[10]:
```
from torch.utils.data import DataLoader
```

[11]:
```
# Define data loader
batch_size = 15
train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

The data loader is typically used in a for-in loop. Let's look at an example

[12]:
```
for xb, yb in train_dl:
    print(xb)
    print(yb)
    break
```

```
tensor([[ 73.,  67.,  43.],
        [ 91.,  88.,  64.],
        [ 91.,  88.,  64.],
        [ 69.,  96.,  70.],
        [ 87., 134.,  58.],
        [ 73.,  67.,  43.],
        [ 73.,  67.,  43.],
        [ 87., 134.,  58.],
        [ 69.,  96.,  70.],
        [ 87., 134.,  58.],
        [ 69.,  96.,  70.],
        [102.,  43.,  37.],
        [102.,  43.,  37.],
        [102.,  43.,  37.],
        [ 91.,  88.,  64.]])
tensor([[ 56.,  70.],
        [ 81., 101.],
        [ 81., 101.],
        [103., 119.],
        [119., 133.],
        [ 56.,  70.],
        [ 56.,  70.],
        [119., 133.],
        [103., 119.],
        [119., 133.],
        [103., 119.],
        [ 22.,  37.],
        [ 22.,  37.],
        [ 22.,  37.],
```

```
        [ 81., 101.]])
```

In each iteration, the data loader returns one batch of data, with the given batch size. If shuffle is set to True, it shuffles the training data before creating batches. Shuffling helps randomize the input to the optimization algorithm, which can lead to faster reduction in the loss.

3. Define some layer - nn.Linear (same as Dense) Instead of initializing the weights & biases manually, we can define the model using the nn.Linear class from PyTorch, which does it automatically.

[13]:
```
import torch.nn as nn

# Define model
model = nn.Linear(3, 2)  #nn.Linear assume this shape (in_features,␣
 ↪out_features)
print(model.weight)
print(model.weight.size()) # (out_features, in_features)
print(model.bias)
print(model.bias.size()) #(out_features)
```

```
Parameter containing:
tensor([[ 0.5513,  0.4977, -0.2445],
        [-0.4788, -0.0623,  0.3356]], requires_grad=True)
torch.Size([2, 3])
Parameter containing:
tensor([-0.0798, -0.5302], requires_grad=True)
torch.Size([2])
```

In fact, our model is simply a function that performs a matrix multiplication of the inputs and the weights w and adds the bias b (for each observation)

PyTorch models also have a helpful .parameters method, which returns a list containing all the weights and bias matrices present in the model. For our linear regression model, we have one weight matrix and one bias matrix.

[14]:
```
# Parameters
list(model.parameters())  #model.param returns a generator
```

[14]:
```
[Parameter containing:
 tensor([[ 0.5513,  0.4977, -0.2445],
         [-0.4788, -0.0623,  0.3356]], requires_grad=True),
 Parameter containing:
 tensor([-0.0798, -0.5302], requires_grad=True)]
```

We can use the model(tensor) API to perform a forward-pass that generate predictions

[15]:
```
# Generate predictions
preds = model(inputs)
preds
```

[15]:
```
tensor([[ 62.9915, -25.2235],
        [ 78.2305, -28.1016],
        [100.3850, -31.0664],
        [ 68.5008, -39.6267],
        [ 68.6173, -16.0530],
        [ 62.9915, -25.2235],
        [ 78.2305, -28.1016],
        [100.3850, -31.0664],
        [ 68.5008, -39.6267],
        [ 68.6173, -16.0530],
        [ 62.9915, -25.2235],
        [ 78.2305, -28.1016],
        [100.3850, -31.0664],
        [ 68.5008, -39.6267],
        [ 68.6173, -16.0530]], grad_fn=<AddmmBackward>)
```

4. Define loss function The nn module contains a lot of useful loss function like this:

[16]:
```
criterion_mse = nn.MSELoss()
criterion_softmax_cross_entropy_loss = nn.CrossEntropyLoss()
```

[17]:
```
mse = criterion_mse(preds, targets)
print(mse)
print(mse.item())  ##print out the loss number
```

```
tensor(8051.1069, grad_fn=<MseLossBackward>)
8051.10693359375
```

5. Define the optimizer *Learning rate and momentum*

Instead of manually manipulating the model's weights & biases using gradients, we can use torch.optim API. We can use optim.SGD to perform stochastic gradient descent where samples are selected in batches (often with random shuffling) instead of as a single group.

Note that model.parameters() is passed as an argument to optim.SGD, so that the optimizer knows which matrices should be modified during the update step. Also, we can specify a learning rate which controls the amount by which the parameters are modified.

[18]:
```
# Define optimizer
opt = torch.optim.SGD(model.parameters(), lr=0.0001, momentum=0.9)
```

Let's talk a bit about other techniques we have used before and how they look like in PyTorch. Since this is simple linear regression problem, we only use learning rate and momentum, but it's worth to talk it here and perhaps we can see the code in the neural network section

Weight initialization

We don't need to worry about weight initialization at all: the weights in most PyTorch operations involving parameters, including nn.Linear, are automatically scaled based on the size of the layer.

Dropout

Dropout is similarly easy. Just as PyTorch has a built-in Module nn.Linear(n_in, n_out) that computes the operations of a Dense layer from before, the Module nn.Dropout(dropout_prob) implements the Dropout operation, with the caveat that the probability passed in is by default the probability of dropping a given neuron.

Learning rate decay

PyTorch has an lr_scheduler class that can be used to decay the learning rate over the epochs. The key import you need to get started is from torch.optim import lr_scheduler We shall use it in the latter section

6. Train the model We are now ready to train the model. We'll follow the exact same process to implement gradient descent:

1. Forward pass
2. Calculate the loss
3. Compute gradients w.r.t the weights and biases
4. Adjust the weights by subtracting a small quantity proportional to the gradient

[19]:
```
# Utility function to train the model
def fit(num_epochs, model, loss_fn, opt, train_dl):

    # Repeat for given number of epochs
    for epoch in range(num_epochs):

        # Train with batches of data
        for xb,yb in train_dl:

            # 1. Forward pass
            pred = model(xb)

            # 2. Calculate loss
            loss = loss_fn(pred, yb)

            # 3. Backward and optimize
            opt.zero_grad()  #if not, the gradients will accumlate
            loss.backward()

            # Print out the gradients.
            #print ('dL/dw: ', model.weight.grad)
            #print ('dL/db: ', model.bias.grad)

            # 4. Update parameters using gradients
            opt.step()

        # Print the progress
        if (epoch+1) % 10 == 0:
```

```
            print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs,␣
↪loss.item()))
```

Some things to note above:

- We use the data loader defined earlier to get batches of data for every iteration.
- Instead of updating parameters (weights and biases) manually, we use opt.step to perform the update, and opt.zero_grad to reset the gradients to zero.
- We've also added a log statement which prints the loss from the last batch of data for every 10th epoch, to track the progress of training. loss.item returns the actual value stored in the loss tensor.

Let's train the model for 100 epochs.

[20]:
```
fit(100, model, criterion_mse, opt, train_dl)
```

```
Epoch [10/100], Loss: 307.9830
Epoch [20/100], Loss: 236.2670
Epoch [30/100], Loss: 425.8909
Epoch [40/100], Loss: 230.8056
Epoch [50/100], Loss: 62.0403
Epoch [60/100], Loss: 6.6486
Epoch [70/100], Loss: 0.6125
Epoch [80/100], Loss: 1.8263
Epoch [90/100], Loss: 1.5746
Epoch [100/100], Loss: 0.8950
```

[21]:
```
# Generate predictions
preds = model(inputs)
loss = criterion_mse(preds, targets)
print(loss.item())
```

```
0.548854649066925
```

1.1.3 Fully-Connected Neural Network

Let's load the MNIST dataset. Our architecture is simple:

1. Input layer receiving 784 features
2. Hidden layer with size of 89 neurons
3. Output layer with size of 10 neurons

We will be using Sigmoid activation.

[22]:
```
# Device configuration
# cuda refers to any NVIDIA GPU that you can use to run your code
# it will be much faster
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[23]:
```
# Hyper-parameters
input_size = 784
hidden_size = 89
num_classes = 10
num_epochs = 5
batch_size = 100
learning_rate = 0.001
```

[24]:
```
# MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='data',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='data',
                                          train=False,
                                          transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

Let's define a fully-connected neural network with one hidden layer. Actually, you can use nn.Sequential to easily do this. I will be showing you how to do this using a class way.

[25]:
```
# Fully connected neural network with one hidden layer
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()  #super(Model, self)
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.sigmoid = nn.Sigmoid()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.sigmoid(out)
        out = self.fc2(out)
        return out
```

Let's now define the model using the class. Every nn.Module can also use the .to(device) to fully use the GPU capabilities.

[26]:
```
model = NeuralNet(input_size, hidden_size, num_classes).to(device)
```

Let's define the Loss and optimizer.

Here we will be using Adam which is an adaptive learning rate optimization. Comparing Adam and SGD, Adam is more adaptive in terms of how it uses momentum and learning rate. Namely, Adam uses the **squared gradients to scale the learning rate** and it takes advantage of momentum by using **moving average of the gradient** instead of gradient itself like SGD with momentum

Whether Adam vs. SGD is still very debatable. Adam is proposed in 2015 to great success and many recent papers found that SGD can be more generalized than Adam...so I really don't know. It's best to try both, I guess.

[27]:
```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()  #this is softmax indeed
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Let's train the model

[28]:
```
# Train the model
total_step = len(train_loader)  #for printing purpose
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        #images shape is [100, 1, 28, 28]

        # Move tensors to the configured device
        # also reshape to [100, 784] so it can be inputted into the Dense layer
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))
```

```
Epoch [1/5], Step [100/600], Loss: 1.0640
Epoch [1/5], Step [200/600], Loss: 0.6722
Epoch [1/5], Step [300/600], Loss: 0.4505
Epoch [1/5], Step [400/600], Loss: 0.3537
Epoch [1/5], Step [500/600], Loss: 0.3003
Epoch [1/5], Step [600/600], Loss: 0.2745
Epoch [2/5], Step [100/600], Loss: 0.4221
```

```
Epoch [2/5], Step [200/600], Loss: 0.2999
Epoch [2/5], Step [300/600], Loss: 0.2181
Epoch [2/5], Step [400/600], Loss: 0.2126
Epoch [2/5], Step [500/600], Loss: 0.2194
Epoch [2/5], Step [600/600], Loss: 0.1523
Epoch [3/5], Step [100/600], Loss: 0.1854
Epoch [3/5], Step [200/600], Loss: 0.1377
Epoch [3/5], Step [300/600], Loss: 0.2328
Epoch [3/5], Step [400/600], Loss: 0.2377
Epoch [3/5], Step [500/600], Loss: 0.1994
Epoch [3/5], Step [600/600], Loss: 0.1487
Epoch [4/5], Step [100/600], Loss: 0.2948
Epoch [4/5], Step [200/600], Loss: 0.1214
Epoch [4/5], Step [300/600], Loss: 0.0889
Epoch [4/5], Step [400/600], Loss: 0.1469
Epoch [4/5], Step [500/600], Loss: 0.1938
Epoch [4/5], Step [600/600], Loss: 0.1554
Epoch [5/5], Step [100/600], Loss: 0.1197
Epoch [5/5], Step [200/600], Loss: 0.1506
Epoch [5/5], Step [300/600], Loss: 0.1683
Epoch [5/5], Step [400/600], Loss: 0.1834
Epoch [5/5], Step [500/600], Loss: 0.1074
Epoch [5/5], Step [600/600], Loss: 0.1583
```

Let's test the model

[29]:
```
# Test the model
# In test phase, we don't need to compute gradients (for memory efficiency)
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)  #returns max value, indices
        total += labels.size(0)  #keep track of total
        correct += (predicted == labels).sum().item()  #.item() give the raw␣
 ↪number

    print('Accuracy of the network on the 10000 test images: {} %'.format(100 *␣
 ↪correct / total))

# Save the model checkpoint
torch.save(model.state_dict(), 'models/dense-mnist.ckpt')
```

```
Accuracy of the network on the 10000 test images: 95.45 %
```

1.1.4 Convolutional Neural Network

Here we will be exploring how to use pyTorch for CNN.

We will also be exploring more typical CNN architectures, as opposed to only simple one conv layer that we use earlier.

[30]:
```
# Hyper parameters
num_epochs = 5
num_classes = 10
batch_size = 100
learning_rate = 0.001

# MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='data',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='data',
                                          train=False,
                                          transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

Typical CNN architectures:

1. **Conv2d**. Definitely all CNN will put some Conv layer. **Number of channels** are often a power of 2, because this often results in more efficient processing. The more channels you have, the greater granularity you give to the network. Also, for **filter size**, typically it will be 3 or 5, where in general, smaller filter size gives better results, while larger filer size speed up the training. For image input, it is desirable to have **squared images** because they are easy to work with the squared filter. In the case with non-squared images, we can use the squared-filter to extract square patches anyway. As for **stride**, it is most common to use stride sizes of 1 in most settings. At most, we use 2 but no more higher. **Padding** is selected using the formula $(filtersize - 1)/2$ to maintain same spatial footprint. To know the output size yield by the Conv2d layer, we can use the equation:

$$\frac{W - F + 2P}{S} + 1$$

where W is the img width, F is the filter size, P is the padding, and S is the stride

In this case (code below), if our W is 28, F is 5, P is 2, and S is 1 then the final img size is

$$\frac{28 - 5 + 2 * 2}{1} + 1 = 28.$$

which maintains the same image size, coincidentally!

2. **BatchNorm**. As you add more layers, one key issue is **vanishing gradients and exploding gradients**. Another one is **covariate shift**. We have already talked about **vanishing gradients**, in which if the gradient is less than 1, as it backpropagates, these gradients drop off exponentially. We know we can fix this by using non-saturating activation function like ReLu where its gradients are not small and does not disappear after some number. Likewise, if our gradient is greater than 1, as it backpropagtes, these gradients increase exponentially. This is called **exploding gradients**. As for **covariate shift**, if you are training a neural network of "red watermelon", but then your model won't perform as well when training for "green watermelon". Why? Because the input distribution is different. This is known as **covariate shift**. All of these problems can be addressed effectively by batch normalization. The idea is that even the values change from input layers to hidden layers, their mean and standard deviation remain the same, thus, reducing **covariate shift**. Of course, since the values are kept small, this actually help solve **exploding gradients**. In a nutshell, it reduce the influence of earlier layers to latter layers, hence, allow each layer to learn more independently. This allows latter layers to be more stable and thus allow training to reach convergence much faster. **Phew...in a super more simple sense, batch normalization keeps the neural network more stable and keep changes more regular** Since batchnorm is simply a normalization procedure, the output size does not change from input size

3. **ReLu**. Convolution operation works well with ReLu activation. ReLu has many advantages. It is faster to compute, and its derivative is also fast to compute. It has non-saturation of gradient (which mean the gradient does not vanish) since its derivative is 1. Since relu is simply an one-to-one mapping, the output size does not change from input size.

4. **Max-pooling**. Back then, convolution operation typically use with max-pooling. Aside from reduction of image size, it also help extract sharpest features of an image as for max pooling, and extract smooth features for average pooling. Anyhow, as we said in earlier lesson, max pooling works somehow but there are also experiments showing worse results. You may want to replace max-pooling with a convolution layer with stride of 2 and see whether it change anything. To know the resulting output size, since pooling is actually a filter itself, we can use the same formula:

$$\frac{W - F + 2P}{S} + 1$$

In this case, if our W is 28, our pooling filter size is 2, padding is 0, and stride is 2, thus the resulting image size is:

$$\frac{28 - 2}{2} + 1 = 14$$

Thus the resulting image size is 14

5. **Dropout**. This is similar to what we have discussed in the previous class so I will not talk too much here. But I guess I should mention that the typical best practices is to choose the probability to be between 0.2 to 0.5 for percentage of values to be zeroed down.

[31]:
```
# Convolutional neural network (two convolutional layers)
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ConvNet, self).__init__()

        #using sequential helps bind multiple operations together
        self.layer1 = nn.Sequential(
            #in_channel = 1
            #out_channel = 16
            #padding = (kernel_size - 1) / 2 = 2
            nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        #after layer 1 will be of shape [100, 16, 14, 14]
        self.layer2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        #after layer 2 will be of shape [100, 32, 7, 7]
        self.fc = nn.Linear(32*7*7, num_classes)
        self.drop_out = nn.Dropout(p=0.2)  #zeroed 0.2% data
        #after fc will be of shape [100, 10]

    def forward(self, x):
        #x shape: [batch, in_channel, img_width, img_height]
        #[100, 1, 28, 28]
        out = self.layer1(x)
        out = self.drop_out(out)
        #after layer 1: shape: [100, 16, 14, 14]
        out = self.layer2(out)
        out = self.drop_out(out)
        #after layer 2: shape: [100, 32, 7, 7]
        out = out.reshape(out.size(0), -1)
        #after squeezing: shape: [100, 1568]
        #we squeeze so that it can be inputted into the fc layer
        out = self.fc(out)
        #after fc layer: shape: [100, 10]
        return out
```

Define the model

[32]:
```
model = ConvNet(num_classes).to(device)
```

Define the loss and optimizer

[33]:
```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

[34]:
```
# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        #con2d expects (batch, channel, width, height)
        images = images.to(device)
        labels = labels.to(device)

        #print(images.size())
        #print(labels.size())

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))
```

```
Epoch [1/5], Step [100/600], Loss: 0.1847
Epoch [1/5], Step [200/600], Loss: 0.0516
Epoch [1/5], Step [300/600], Loss: 0.1148
Epoch [1/5], Step [400/600], Loss: 0.0672
Epoch [1/5], Step [500/600], Loss: 0.0617
Epoch [1/5], Step [600/600], Loss: 0.0846
Epoch [2/5], Step [100/600], Loss: 0.0460
Epoch [2/5], Step [200/600], Loss: 0.0523
Epoch [2/5], Step [300/600], Loss: 0.0572
Epoch [2/5], Step [400/600], Loss: 0.1265
Epoch [2/5], Step [500/600], Loss: 0.0801
Epoch [2/5], Step [600/600], Loss: 0.0315
Epoch [3/5], Step [100/600], Loss: 0.0747
Epoch [3/5], Step [200/600], Loss: 0.0165
Epoch [3/5], Step [300/600], Loss: 0.0271
Epoch [3/5], Step [400/600], Loss: 0.0513
Epoch [3/5], Step [500/600], Loss: 0.0614
```

```
Epoch [3/5], Step [600/600], Loss: 0.0328
Epoch [4/5], Step [100/600], Loss: 0.0994
Epoch [4/5], Step [200/600], Loss: 0.0195
Epoch [4/5], Step [300/600], Loss: 0.0135
Epoch [4/5], Step [400/600], Loss: 0.0618
Epoch [4/5], Step [500/600], Loss: 0.0467
Epoch [4/5], Step [600/600], Loss: 0.0740
Epoch [5/5], Step [100/600], Loss: 0.0508
Epoch [5/5], Step [200/600], Loss: 0.0170
Epoch [5/5], Step [300/600], Loss: 0.0570
Epoch [5/5], Step [400/600], Loss: 0.0165
Epoch [5/5], Step [500/600], Loss: 0.0288
Epoch [5/5], Step [600/600], Loss: 0.0697
```

Making predictions Same code.

In previous class, we talk about inference mode in which real prediction happens. In such mode, usually, we do not apply dropout or we apply more generalized algorithm. Anyhow, in pytorch, we do not need to manually code this, we simply do model.eval() to signal that the model will be run in evaluation mode.

[35]:
```
# Test the model
model.eval()  # eval mode (batchnorm uses moving mean/variance instead of␣
 ↪mini-batch mean/variance)
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.
 ↪format(100 * correct / total))

# Save the model checkpoint
torch.save(model.state_dict(), 'models/cnn.ckpt')
```

```
Test Accuracy of the model on the 10000 test images: 98.81 %
```

1.1.5 Recurrent Neural Network

Here we will be exploring how to use pyTorch for RNN.

Just like CNN, we will be exploring typical RNN architectures

[36]:
```
# Hyper-parameters
sequence_length = 28
input_size = 28
hidden_size = 128
num_layers = 2
num_classes = 10
batch_size = 100
num_epochs = 2
learning_rate = 0.01
```

[37]:
```
# Recurrent neural network (many-to-one)
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,␣
↪batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # Set initial hidden and cell states
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).
↪to(device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).
↪to(device)

        # Forward propagate LSTM
        out, _ = self.lstm(x, (h0, c0))  # out: tensor of shape (batch_size,␣
↪seq_length, hidden_size)

        # Decode the hidden state of the last time step
        out = self.fc(out[:, -1, :])
        return out
```

[39]:
```
model = RNN(input_size, hidden_size, num_layers, num_classes).to(device)
```

[40]:
```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

[41]:
```
# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.reshape(-1, sequence_length, input_size).to(device)
        labels = labels.to(device)
```

```
        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))
```

```
Epoch [1/2], Step [100/600], Loss: 0.6756
Epoch [1/2], Step [200/600], Loss: 0.5046
Epoch [1/2], Step [300/600], Loss: 0.2798
Epoch [1/2], Step [400/600], Loss: 0.1001
Epoch [1/2], Step [500/600], Loss: 0.0847
Epoch [1/2], Step [600/600], Loss: 0.1568
Epoch [2/2], Step [100/600], Loss: 0.0223
Epoch [2/2], Step [200/600], Loss: 0.0878
Epoch [2/2], Step [300/600], Loss: 0.0865
Epoch [2/2], Step [400/600], Loss: 0.0154
Epoch [2/2], Step [500/600], Loss: 0.1367
Epoch [2/2], Step [600/600], Loss: 0.0714
```

[42]:
```
# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.reshape(-1, sequence_length, input_size).to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.
 ↪format(100 * correct / total))
```

```
Test Accuracy of the model on the 10000 test images: 97.63 %
```

1.2 What's next in your adventure?

So this is our last lecture....sadly!

We have come a long way? Can you remember the "you" four months ago and now? Did you learn a lot? I hope so! I hope my scratch series help you better understand things from the first principle.

In fact, I have only **covered the basic things**. Seriously, I have barely scratched the surface. Dense for structured data, CNN for images, RNN for sequence data. They are all concepts that have been known for almost 10 years. These are the following things that you probably want to keep studying. If you are doing thesis, you are likely going to come across these techniques; CNN or RNN won't suffice for your thesis:

1. **Generative Adversarial Network**: would it be cool if you give the AI a picture of Mona-Lisa and the AI can learn to draw similar picture? This is the network containing a **generator** model and **discriminative** model. It is trying to understand the distribution of the images, and try to generate a fake image with the **generator** where the **discriminative** model trying to distinguish whether its fake or real. By going back and forth, you can create many similar images/signals/text or restore loss data. It is not only limited to images by the way.
2. **Transfer Learning**: training data is a lengthy process. Transfer learning is a technique related to understanding how we can use some pre-trained model and apply it to another totally different distribution *without* training again. Mostly, it has to deal with shifting and transferring the *distribution*
3. **Autoencoder**: neural network is not only limited to supervised learning. We can also do unsupervised learning by creating a **encoder** which squeeze things into some middle form and a **decoder** that transform the middle form back to the original one. Since the middle form is the best way to represent the data in a diminished form, we can say that the middle form actually holds **most important** information about the data so that it can be decoded to data that is as close to the original. Autoencoder has been useful for noise cancelation, outlier detection, and even prediction task (just like how PCA can improve prediction accuracy)
4. **Neural style transfer**: Given a picture of Chaky, and another picture of some comic, let's say Harry Potter, would it be cool to draw Chaky in a Harry Potter way? This is called **style transfer** in which the network tries to learn the style from the comic and apply to Chaky image.
5. **Transformers**. This is a very recent trend. Back then, if we are talking about sequential data, we would talk about LSTM, GRU, or TCN. But right now, Transformers which is based on Attention-Mechanisms have shown great results and has been used to improve accuracy in NLP, and signal processing. Some NLP pre-trained model like BERT or GBT2 using Transformers can perform really well in detecting fake news, fake reviews, or even write own reviews.

Next week onward, some of my Ph.D./Master students will be coming to give their talk on some of these topics and how they apply to their brain/data science research. This is a good time to take a glimpse how thesis looks like. Please spare them some slack because they are also learning just like you, but I am sure they have plenty good things to share with you.

[ ]: