Melba Nuzen **Problem Set 3** CS 155 February 7, 2020

1 Deep Learning Principles

Problem A:
The first network is initialized with nonzero weights, and after every layer its weights are updated. As there are more iterations, we see the errors generally decrease.

On the other hand, we first see that the second network only has weights initialized to be 0. After about 250 iterations in both networks, we see that the first network (the one initialized with nonzero weights) performs significantly better than the second network. This makes sense since there can be no learning with weights of 0 in back propagation, since the back propagation algorithm uses the derivative of our weights to adjust properly. Moreover, we know $(x) = \max(x, 0)$, and after the first layer in our second network, the inputs will always be 0, meaning the gradient will always be 0, meaning the weights will always stay as 0.

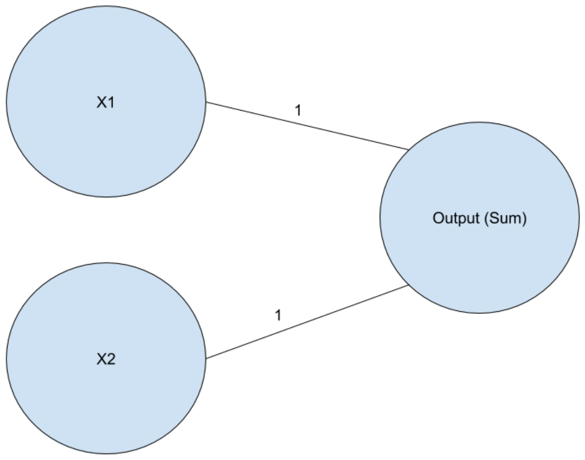
Problem B:
The first difference between using ReLU versus the sigmoid function is that the second network's weights were updated even though they were initialized as 0, so nonzero values are generated even when initial values are 0. So the previous problem with weights initialized as 0 does not apply.

What we do see is that since all of the weights in the second network were initialized with the same value, the weights end up around the same as well. Contrastly, some weights in the first network are updated differently and end up varying much more drastically than the second network.

In general we see that the first network is essential stagnant until about 500 iterations, which is when the weights start to update significantly and when the error begins to decrease. On the other hand, the second model remains stagnant until about 3200 iterations in, which is a much larger gap. This is most likely because, since the initial weights are 0, backpropagation takes much longer for the weights to become significant. Conversely, the first network takes less time since the weights take less time to update and become significant with backpropagation.

Problem C:
If we loop through all of the negative samples instead of selecting them randomly, then the model would adjust to correctly classify negative points, meaning having a large negative bias because of the ReLU function. Then all of the positive samples fed in afterwards wouldn't help learn since ReLU would have nodes become zero, or dies.

Problem D:



Problem E:
The minimum number of total layers would be three, since we need an input, a hidden, and an output layer. XOR is **not** linearly separable so using just one input layer and one output layer wouldn't be able to implement XOR.

2 Depth vs Width on the MNIST Dataset

Problem A:
After installing PyTorch, I get the following confirmation: **Successfully installed pillow-7.0.0 torch-1.4.0 torchvision-0.5.0**.

Problem B:
The images in the training set at 28 by 28 pixels and each value in the array (the values for which range from 0 to 255) stand for the RGB color value. The training set has 60000 images and the testing set has 10000 images.

Problem C:
My neural net had two hidden layers, one of 70 neurons and one of 30 neurons with ReLU between both, and a Softmax activation. This generated an accuracy of .976 with 10 epochs.

Problem D:
My neural net had three hidden layers, one of 100 neurons, one of 60 neurons, and one of 40 neurons with ReLU between each one, and a Softmax activation. This generated an accuracy of .9813 with 20 epochs.

Problem E:
My neural net had four hidden layers, one of 500 neurons, one of 250 neurons, one of 150, and one of 100 neurons with ReLU between each one, and a Softmax activation. This generated an accuracy of .9840 with 30 epochs.

3 Convolution Neural Networks

Problem A:
Adding zeros to pad would help conserve the image's original size and maintain as much about the original image as possible. However, adding zeros to pad might also distort the weights of different parts of the image, e.g. the original image's edges could be weighed differently than the center of the image since they're grouped with zeros-pads.

Problem B:
We end up with 608 parameters since we have 8 filters times 5 times 5 times 3 plus our 8 biases.

Problem C:
We have the shape of the output as 8 filters by 28 by 28 by 3.

Problem D:

$$\begin{bmatrix} 1 & .5 \\ .5 & .25 \end{bmatrix}, \begin{bmatrix} .5 & 1 \\ .25 & .5 \end{bmatrix}, \begin{bmatrix} .25 & .5 \\ .5 & 1 \end{bmatrix}, \begin{bmatrix} .5 & .25 \\ 1 & .5 \end{bmatrix}$$

Problem E:

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Problem F:
Pool would group pixels into regions, then calculate given those regions. If a pixel were missing, then pooling could make up for that lost pixel since the pixels around it have similar effects. Additionally, groups of pixels could most likely absorb some noise. Thus, pooling would be advantageous.

Problem G:
After referring to several convolutional architectures online, it seemed that capturing the model as a class would be easier to implement (rather than just creating a model as you are using it), so when writing this model, I initialized a ConvNet class and created two layers.

The first sequential layer has a Conv2d module that takes in our data and has 16 output channels. We used 5 by 5 convolution filters and a padding of 2 and stride of 1. We know this since the output dimension is equivalent to 1 plus our input width plus 2 times the padding minus our filter size. Then we have a ReLU activation, followed by a max pooling operation, which has a pooling size of 2 by 2 and a stride of 2. This effectively reduces the image size by a factor of 2.

This means we go into our second sequential layer with an input of 16 channels of 14 by 14 images. This second layer uses a similar structure to the first and reduces the image size again by 2, to end up with 7 by 7 images.

Finally, the last three layers are a dropout layer (to prevent overfitting) and two linear connected layers. The first has 7 by 7 by 32 nodes and connects to a second layer of 600 nodes. The forward function then overrides a given forward function in nn.Module and passes the data into the first layer, then the second, then flattened before being pushed through the dropout and final two fully connected layers. I used Adam as my optimizer with an initial learning rate of .001. This model has a total of 960754 parameters.

This architecture, after running through 10 epochs, generated an accuracy of .986.

Problem 2 Sample Code

This sample code is meant as a guide on how to use PyTorch and how to use the relevant model layers. This not a guide on how to design a network and the network in this example is intentionally designed to have poor performace.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
```

Loading MNIST

The `torchvision` module contains links to many standard datasets. We can load the MNIST dataset into a `Dataset` object as follows:

In [2]:
```
train_dataset = datasets.MNIST('./data', train=True, download=True,  # Downloads i
nto a directory ../data
                               transform=transforms.ToTensor())
test_dataset = datasets.MNIST('./data', train=False, download=False,  # No need to
download again
                              transform=transforms.ToTensor())
```

```
0.0%

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./dat
a/MNIST/raw/train-images-idx3-ubyte.gz

100.1%

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

28.4%

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./dat
a/MNIST/raw/train-labels-idx1-ubyte.gz

0.5%5%

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data
/MNIST/raw/t10k-images-idx3-ubyte.gz

180.4%

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data
/MNIST/raw/t10k-labels-idx1-ubyte.gz
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
Processing...
Done!
```

The `Dataset` object is an iterable where each element is a tuple of (input `Tensor` , target):

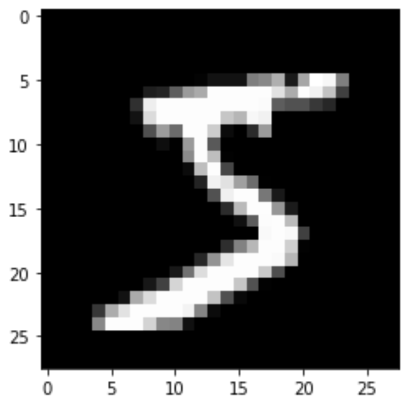
In [3]:
```
print(len(train_dataset), type(train_dataset[0][0]), type(train_dataset[0][1]))
```

```
60000 <class 'torch.Tensor'> <class 'int'>
```

We can convert images to numpy arrays and plot them with matplotlib:

In [4]:
```
plt.imshow(train_dataset[0][0][0].numpy(), cmap='gray')
```

Out[4]:
```
<matplotlib.image.AxesImage at 0x10787c910>
```



Network Definition

Let's instantiate a model and take a look at the layers.

In [42]:
```
model = nn.Sequential(
    # In problem 2, we don't use the 2D structure of an image at all. Our network
    # takes in a flat vector of the pixel values as input.
    nn.Flatten(),
    nn.Linear(784, 70),
    nn.ReLU(),
    nn.Linear(70, 30),
    nn.ReLU(),
    #nn.Dropout(0.5),
    nn.Linear(30, 10),
    nn.LogSoftmax(dim=1)
)
print(model)
```

```
Sequential(
  (0): Flatten()
  (1): Linear(in_features=784, out_features=70, bias=True)
  (2): ReLU()
  (3): Linear(in_features=70, out_features=30, bias=True)
  (4): ReLU()
  (5): Linear(in_features=30, out_features=10, bias=True)
  (6): LogSoftmax()
)
```

Training

We also choose an optimizer and a loss function.

In [43]:
```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()
```

We could write our training procedure manually and directly index the `Dataset` objects, but the `DataLoader` object conveniently creates an iterable for automatically creating random minibatches:

In [46]:
```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=True)
```

We now write our backpropagation loop, training for 10 epochs.

In [47]:
```
# Some layers, such as Dropout, behave differently during training
model.train()

for epoch in range(10):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Erase accumulated gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Calculate loss
        loss = loss_fn(output, target)

        # Backward pass
        loss.backward()

        # Weight update
        optimizer.step()

    # Track loss each epoch
    print('Train Epoch: %d  Loss: %.4f' % (epoch + 1,  loss.item()))
```

```
Train Epoch: 1  Loss: 0.0428
Train Epoch: 2  Loss: 0.0086
Train Epoch: 3  Loss: 0.2078
Train Epoch: 4  Loss: 0.0245
Train Epoch: 5  Loss: 0.0209
Train Epoch: 6  Loss: 0.0536
Train Epoch: 7  Loss: 0.0033
Train Epoch: 8  Loss: 0.0004
Train Epoch: 9  Loss: 0.0876
Train Epoch: 10  Loss: 0.0046
```

Testing

We can perform forward passes through the network without saving gradients.

In [48]:
```
# Putting layers like Dropout into evaluation mode
model.eval()

test_loss = 0
correct = 0

# Turning off automatic differentiation
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        test_loss += loss_fn(output, target).item()  # Sum up batch loss
        pred = output.argmax(dim=1, keepdim=True)  # Get the index of the max clas
s score
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)

print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
      (test_loss, correct, len(test_loader.dataset),
       100. * correct / len(test_loader.dataset)))
```

```
Test set: Average loss: 0.0030, Accuracy: 9761/10000 (97.6100)
```

In [ ]:

In [ ]:

Problem 2 Sample Code

This sample code is meant as a guide on how to use PyTorch and how to use the relevant model layers. This not a guide on how to design a network and the network in this example is intentionally designed to have poor performace.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
```

Loading MNIST

The `torchvision` module contains links to many standard datasets. We can load the MNIST dataset into a `Dataset` object as follows:

In [2]:
```
train_dataset = datasets.MNIST('./data', train=True, download=True,  # Downloads i
nto a directory ../data
                               transform=transforms.ToTensor())
test_dataset = datasets.MNIST('./data', train=False, download=False,  # No need to
download again
                              transform=transforms.ToTensor())
```

```
0.0%

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./dat
a/MNIST/raw/train-images-idx3-ubyte.gz

100.1%

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

28.4%

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./dat
a/MNIST/raw/train-labels-idx1-ubyte.gz

0.5%5%

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data
/MNIST/raw/t10k-images-idx3-ubyte.gz

180.4%

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data
/MNIST/raw/t10k-labels-idx1-ubyte.gz
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
Processing...
Done!
```

The `Dataset` object is an iterable where each element is a tuple of (input `Tensor`, target):

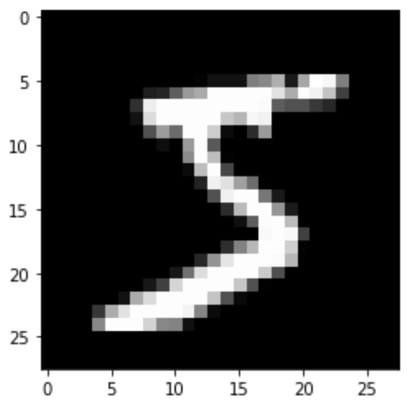
In [3]:
```
print(len(train_dataset), type(train_dataset[0][0]), type(train_dataset[0][1]))
```

```
60000 <class 'torch.Tensor'> <class 'int'>
```

We can convert images to numpy arrays and plot them with matplotlib:

In [4]:
```
plt.imshow(train_dataset[0][0][0].numpy(), cmap='gray')
```

Out[4]:
```
<matplotlib.image.AxesImage at 0x10787c910>
```



Network Definition

Let's instantiate a model and take a look at the layers.

In [64]:
```
model = nn.Sequential(
    # In problem 2, we don't use the 2D structure of an image at all. Our network
    # takes in a flat vector of the pixel values as input.
    nn.Flatten(),
    nn.Linear(784, 100),
    nn.ReLU(),
    nn.Linear(100, 60),
    nn.ReLU(),
    nn.Linear(60, 40),
    nn.ReLU(),
    nn.Linear(40, 10),
    nn.LogSoftmax(dim=1)
)
print(model)
```

```
Sequential(
  (0): Flatten()
  (1): Linear(in_features=784, out_features=100, bias=True)
  (2): ReLU()
  (3): Linear(in_features=100, out_features=60, bias=True)
  (4): ReLU()
  (5): Linear(in_features=60, out_features=40, bias=True)
  (6): ReLU()
  (7): Linear(in_features=40, out_features=10, bias=True)
  (8): LogSoftmax()
)
```

Training

We also choose an optimizer and a loss function.

```
In [65]: optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
         loss_fn = nn.CrossEntropyLoss()
```

We could write our training procedure manually and directly index the `Dataset` objects, but the `DataLoader` object conveniently creates an iterable for automatically creating random minibatches:

```
In [66]: train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=T
         rue)
         test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=Tru
         e)
```

We now write our backpropagation loop, training for 10 epochs.

In [72]:
```
# Some layers, such as Dropout, behave differently during training
model.train()

for epoch in range(20):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Erase accumulated gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Calculate loss
        loss = loss_fn(output, target)

        # Backward pass
        loss.backward()

        # Weight update
        optimizer.step()

    # Track loss each epoch
    print('Train Epoch: %d  Loss: %.4f' % (epoch + 1,  loss.item()))
```

```
Train Epoch: 1  Loss: 0.0000
Train Epoch: 2  Loss: 0.0258
Train Epoch: 3  Loss: 0.0001
Train Epoch: 4  Loss: 0.0004
Train Epoch: 5  Loss: 0.0078
Train Epoch: 6  Loss: 0.0007
Train Epoch: 7  Loss: 0.0000
Train Epoch: 8  Loss: 0.0000
Train Epoch: 9  Loss: 0.0001
Train Epoch: 10  Loss: 0.0863
Train Epoch: 11  Loss: 0.0000
Train Epoch: 12  Loss: 0.0066
Train Epoch: 13  Loss: 0.0001
Train Epoch: 14  Loss: 0.0001
Train Epoch: 15  Loss: 0.0003
Train Epoch: 16  Loss: 0.0000
Train Epoch: 17  Loss: 0.0002
Train Epoch: 18  Loss: 0.0048
Train Epoch: 19  Loss: 0.0000
Train Epoch: 20  Loss: 0.0000
```

Testing

We can perform forward passes through the network without saving gradients.

In [73]:
```
# Putting layers like Dropout into evaluation mode
model.eval()

test_loss = 0
correct = 0

# Turning off automatic differentiation
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        test_loss += loss_fn(output, target).item()  # Sum up batch loss
        pred = output.argmax(dim=1, keepdim=True)  # Get the index of the max clas
s score
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)

print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
      (test_loss, correct, len(test_loader.dataset),
       100. * correct / len(test_loader.dataset)))
```

```
Test set: Average loss: 0.0044, Accuracy: 9813/10000 (98.1300)
```

In [ ]:

In [ ]:

Problem 2 Sample Code

This sample code is meant as a guide on how to use PyTorch and how to use the relevant model layers. This not a guide on how to design a network and the network in this example is intentionally designed to have poor performace.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
```

Loading MNIST

The `torchvision` module contains links to many standard datasets. We can load the MNIST dataset into a `Dataset` object as follows:

In [2]:
```
train_dataset = datasets.MNIST('./data', train=True, download=True,  # Downloads i
nto a directory ../data
                               transform=transforms.ToTensor())
test_dataset = datasets.MNIST('./data', train=False, download=False,  # No need to
download again
                              transform=transforms.ToTensor())
```

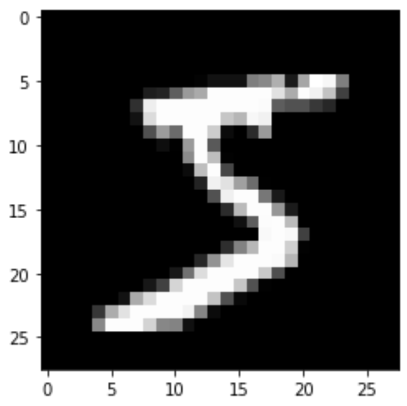
```
0.0%

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./dat
a/MNIST/raw/train-images-idx3-ubyte.gz

100.1%

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

28.4%

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./dat
a/MNIST/raw/train-labels-idx1-ubyte.gz

0.5%5%

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data
/MNIST/raw/t10k-images-idx3-ubyte.gz

180.4%

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data
/MNIST/raw/t10k-labels-idx1-ubyte.gz
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
Processing...
Done!
```

The `Dataset` object is an iterable where each element is a tuple of (input `Tensor` , target):

In [3]:
```
print(len(train_dataset), type(train_dataset[0][0]), type(train_dataset[0][1]))
```
```
60000 <class 'torch.Tensor'> <class 'int'>
```

We can convert images to numpy arrays and plot them with matplotlib:

In [4]:
```
plt.imshow(train_dataset[0][0][0].numpy(), cmap='gray')
```
Out[4]:
```
<matplotlib.image.AxesImage at 0x10787c910>
```



Network Definition

Let's instantiate a model and take a look at the layers.

In [74]:
```
model = nn.Sequential(
    # In problem 2, we don't use the 2D structure of an image at all. Our network
    # takes in a flat vector of the pixel values as input.
    nn.Flatten(),
    nn.Linear(784, 500),
    nn.ReLU(),
    nn.Linear(500, 250),
    nn.ReLU(),
    nn.Linear(250, 150),
    nn.ReLU(),
    nn.Linear(150, 100),
    nn.ReLU(),
    nn.Linear(100, 10),
    nn.LogSoftmax(dim=1)
)
print(model)
```

```
Sequential(
  (0): Flatten()
  (1): Linear(in_features=784, out_features=500, bias=True)
  (2): ReLU()
  (3): Linear(in_features=500, out_features=250, bias=True)
  (4): ReLU()
  (5): Linear(in_features=250, out_features=150, bias=True)
  (6): ReLU()
  (7): Linear(in_features=150, out_features=100, bias=True)
  (8): ReLU()
  (9): Linear(in_features=100, out_features=10, bias=True)
  (10): LogSoftmax()
)
```

Training

We also choose an optimizer and a loss function.

In [75]:
```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()
```

We could write our training procedure manually and directly index the `Dataset` objects, but the `DataLoader` object conveniently creates an iterable for automatically creating random minibatches:

In [76]:
```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=True)
```

We now write our backpropagation loop, training for 10 epochs.

In [79]:
```
# Some layers, such as Dropout, behave differently during training
model.train()

for epoch in range(30):
    for batch_idx, (data, target) in enumerate(train_loader):
        # Erase accumulated gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Calculate loss
        loss = loss_fn(output, target)

        # Backward pass
        loss.backward()

        # Weight update
        optimizer.step()

    # Track loss each epoch
    print('Train Epoch: %d  Loss: %.4f' % (epoch + 1,  loss.item()))
```

```
Train Epoch: 1  Loss: 0.0016
Train Epoch: 2  Loss: 0.0074
Train Epoch: 3  Loss: 0.0018
Train Epoch: 4  Loss: 0.0002
Train Epoch: 5  Loss: 0.0000
Train Epoch: 6  Loss: 0.0000
Train Epoch: 7  Loss: 0.0002
Train Epoch: 8  Loss: 0.0663
Train Epoch: 9  Loss: 0.0002
Train Epoch: 10  Loss: 0.0637
Train Epoch: 11  Loss: 0.0004
Train Epoch: 12  Loss: 0.0003
Train Epoch: 13  Loss: 0.0049
Train Epoch: 14  Loss: 0.0010
Train Epoch: 15  Loss: 0.0000
Train Epoch: 16  Loss: 0.0004
Train Epoch: 17  Loss: 0.0000
Train Epoch: 18  Loss: 0.0001
Train Epoch: 19  Loss: 0.2435
Train Epoch: 20  Loss: 0.0000
Train Epoch: 21  Loss: 0.0081
Train Epoch: 22  Loss: 0.0000
Train Epoch: 23  Loss: 0.0071
Train Epoch: 24  Loss: 0.0100
Train Epoch: 25  Loss: 0.0000
Train Epoch: 26  Loss: 0.0002
Train Epoch: 27  Loss: 0.0000
Train Epoch: 28  Loss: 0.0000
Train Epoch: 29  Loss: 0.0000
Train Epoch: 30  Loss: 0.0000
```

Testing

We can perform forward passes through the network without saving gradients.

In [80]:
```
# Putting layers like Dropout into evaluation mode
model.eval()

test_loss = 0
correct = 0

# Turning off automatic differentiation
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        test_loss += loss_fn(output, target).item()  # Sum up batch loss
        pred = output.argmax(dim=1, keepdim=True)  # Get the index of the max clas
s score
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)

print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
      (test_loss, correct, len(test_loader.dataset),
       100. * correct / len(test_loader.dataset)))
```

```
Test set: Average loss: 0.0074, Accuracy: 9840/10000 (98.4000)
```

In [ ]:

In [ ]:

Problem 3

Use this notebook to write your code for problem 3.

In [4]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

3D - Convolutional network

As in problem 2, we have conveniently provided for your use code that loads and preprocesses the MNIST data.

In [5]:
```
# load MNIST data into PyTorch format
import torch
import torchvision
import torchvision.transforms as transforms

# set batch size
batch_size = 32

# load training data downloaded into data/ folder
mnist_training_data = torchvision.datasets.MNIST('data/', train=True, download=Tru
e,
                                                 transform=transforms.ToTensor())
# transforms.ToTensor() converts batch of images to 4-D tensor and normalizes 0-25
5 to 0-1.0
training_data_loader = torch.utils.data.DataLoader(mnist_training_data,
                                                   batch_size=batch_size,
                                                   shuffle=True)

# load test data
mnist_test_data = torchvision.datasets.MNIST('data/', train=False, download=True,
                                                 transform=transforms.ToTensor())
test_data_loader = torch.utils.data.DataLoader(mnist_test_data,
                                                   batch_size=batch_size,
                                                   shuffle=False)
```

In [6]:
```
# look at the number of batches per epoch for training and validation
print(f'{len(training_data_loader)} training batches')
print(f'{len(training_data_loader) * batch_size} training samples')
print(f'{len(test_data_loader)} validation batches')
```

```
1875 training batches
60000 training samples
313 validation batches
```

In [65]:
```
# sample model
import torch.nn as nn

class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.first = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.second = 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))
        self.Dropout = nn.Dropout()
        # 7 * 7 * 32
        self.fc1 = nn.Linear(7*7*32, 600)
        self.fc2 = nn.Linear(600, 10)

    def forward(self, data):
        out = self.first(data)
        out = self.second(out)
        # reshape
        out = out.reshape(out.size(0), -1)
        out = self.Dropout(out)
        out = self.fc1(out)
        out = self.fc2(out)
        return out

model = ConvNet()
```

In [66]:
```
# why don't we take a look at the shape of the weights for each layer
for p in model.parameters():
    print(p.data.shape)
```

```
torch.Size([16, 1, 5, 5])
torch.Size([16])
torch.Size([16])
torch.Size([16])
torch.Size([32, 16, 5, 5])
torch.Size([32])
torch.Size([32])
torch.Size([32])
torch.Size([600, 1568])
torch.Size([600])
torch.Size([10, 600])
torch.Size([10])
```

In [67]:
```
# our model has some # of parameters:
count = 0
for p in model.parameters():
    n_params = np.prod(list(p.data.shape)).item()
    count += n_params
print(f'total params: {count}')
```

```
total params: 960754
```

In [68]:
```
# For a multi-class classification problem
#import torch.optim as optim
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=.001)
```

In [69]:
```
# Train the model for 10 epochs, iterating on the data in batches
n_epochs = 10

# store metrics
training_accuracy_history = np.zeros([n_epochs, 1])
training_loss_history = np.zeros([n_epochs, 1])
validation_accuracy_history = np.zeros([n_epochs, 1])
validation_loss_history = np.zeros([n_epochs, 1])

for epoch in range(n_epochs):
    print(f'Epoch {epoch+1}/10:', end='')
    train_total = 0
    train_correct = 0
    # train
    model.train()
    for i, data in enumerate(training_data_loader):
        images, labels = data
        optimizer.zero_grad()
        # forward pass
        output = model(images)
        # calculate categorical cross entropy loss
        loss = criterion(output, labels)
        # backward pass
        loss.backward()
        optimizer.step()

        # track training accuracy
        _, predicted = torch.max(output.data, 1)
        train_total += labels.size(0)
        train_correct += (predicted == labels).sum().item()
        # track training loss
        training_loss_history[epoch] += loss.item()
        # progress update after 180 batches (~1/10 epoch for batch size 32)
        if i % 180 == 0: print('.',end='')
    training_loss_history[epoch] /= len(training_data_loader)
    training_accuracy_history[epoch] = train_correct / train_total
    print(f'\n\tloss: {training_loss_history[epoch,0]:0.4f}, acc: {training_accura
cy_history[epoch,0]:0.4f}',end='')

    # validate
    test_total = 0
    test_correct = 0
    with torch.no_grad():
        model.eval()
        for i, data in enumerate(test_data_loader):
            images, labels = data
            # forward pass
            output = model(images)
            # find accuracy
            _, predicted = torch.max(output.data, 1)
            test_total += labels.size(0)
            test_correct += (predicted == labels).sum().item()
            # find loss
            loss = criterion(output, labels)
            validation_loss_history[epoch] += loss.item()
        validation_loss_history[epoch] /= len(test_data_loader)
        validation_accuracy_history[epoch] = test_correct / test_total
    print(f', val loss: {validation_loss_history[epoch,0]:0.4f}, val acc: {validat
ion_accuracy_history[epoch,0]:0.4f}')
```

```
Epoch 1/10:..........
        loss: 0.2103, acc: 0.9369, val loss: 0.0465, val acc: 0.9843
Epoch 2/10:..........
        loss: 0.0968, acc: 0.9709, val loss: 0.0462, val acc: 0.9848
Epoch 3/10:..........
        loss: 0.0722, acc: 0.9785, val loss: 0.0308, val acc: 0.9896
Epoch 4/10:..........
        loss: 0.0634, acc: 0.9807, val loss: 0.0355, val acc: 0.9889
Epoch 5/10:..........
        loss: 0.0582, acc: 0.9825, val loss: 0.0327, val acc: 0.9886
Epoch 6/10:..........
        loss: 0.0504, acc: 0.9847, val loss: 0.0305, val acc: 0.9905
Epoch 7/10:..........
        loss: 0.0477, acc: 0.9854, val loss: 0.0318, val acc: 0.9896
Epoch 8/10:..........
        loss: 0.0441, acc: 0.9859, val loss: 0.0301, val acc: 0.9909
Epoch 9/10:..........
        loss: 0.0431, acc: 0.9870, val loss: 0.0314, val acc: 0.9902
Epoch 10/10:..........
        loss: 0.0390, acc: 0.9881, val loss: 0.0232, val acc: 0.9925
```

Above, we output the training loss/accuracy as well as the validation loss and accuracy. Not bad! Let's see if you can do better.