Deep Learning Fall 2023

Name: Santosh Srinivas Ravichandran
NetID: sr6411

Problem 1:

A)

$f(x) = ||x||_2^2$

Assume the components of x are $x_1, x_2, \ldots . x_d$

$f(x) = x_1^2 + x_2^2 + \ldots . + x_d^2$

$\frac{\partial f(x)}{\partial (x)} = 2x$

B)

Assume D = $\sum_{i=1}^{i=n} ||(x_i - \mu)||_2^2$

=> $\frac{\partial D}{\partial (\mu)} = -2 * \sum_{i=1}^{i=n} (x_i - \mu)$

Setting $\frac{\partial D}{\partial \mu} = 0$ for finding the minima, also $\frac{\partial^2 D}{\partial^2 \mu} > 0$, thus its minima and not maxima.

$\frac{\partial D}{\partial (\mu)} = -2 * \sum_{i=1}^{i=n} (x_i - \mu) = 0$

=> $\sum_{i=1}^{i=n} (x_i - \mu) = 0$

$$=> \mu = \frac{\sum_{i=1}^{i=n} x_i}{n}$$

2

A)

L1 Loss function:

$$L = \sum_{i=1}^{i=n} |(y_i - \hat{y}_i|$$

$$L = ||XW - Y||_1$$

Size of X is n*d where n is the number of examples and d is the dimension of a X_i

W is d * 1

Y is n* 1

B)

From L, we get

$$\frac{\partial L}{\partial W} = \sum_{i=1}^{i=n} sign(< x_i, w > - y_i).x_i$$

$$\nabla L(w) = X^T.\, sign(XW - Y)$$

To get the closed form , we set $\nabla L(w) = 0$, and solve for optimal w*

$$X^T.\, sign(XW^* - Y) = 0$$

Since $sign(x)$ is a piecewise function that behaves differently for different inputs, it is NOT possible to get W^* in a closed form equation. In order to solve for it , we need to look at specific intervals for its sign and solve accordingly. Thus a general closed form expression for parameters is not possible.

Thus it is not possible to write the optimal linear model in a closed form equation.

C)

According to the 3 step ML-recipe,

Step 1: Representation of the system: In the question we were asked to find the “optimal linear model” so a linear model or linear representation was chosen.

Step 2: Measure of goodness: According the question, $L1$ loss function was used to measure the goodness.

Step 3: Method to optimize: We tried to see if finding the optimal parameters as a closed form expression was possible.

For this, we first find the gradient of the loss function from step 2, and set it to 0 and see if W can be derived from there. If so we have found a closed form expression of W to get the optimal parameters as a way of optimization.

But in our case, closed form expression of W wasn’t possible because of the nature of the gradient of the loss function. So another method of optimization is using gradient descent. Where we just need to know the gradient value of the loss function wrt to the parameters at a specific point in the parameter space. Then we keep updating it iteratively to reach the optimal w.

3

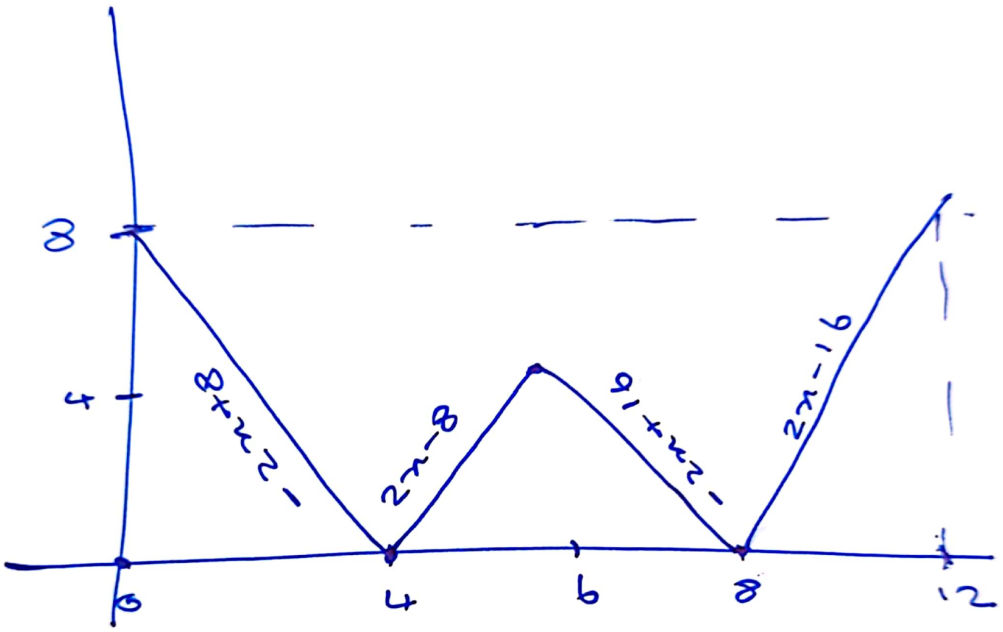
A)

Dataset 1: (x, y) = (0, 8), (1, 6), (2, 4), (3, 2), (4, 0), (5, 2), (6, 4), (7, 2), (8, 0), (9, 2), (10, 4), (11, 6), (12, 8)

Dataset 2: (x,y) = (0, 0), (1, 1), (2, 2), (3, 3), (4, 2), (5, 1), (6, 1.5), (7, 2), (8, 2.5), (9, 3), (10, 3.5)

By individually examining the trends and patterns in the data, we may find the weights and biases that exactly fit the provided datasets for this issue.

Dataset 1:



The following are assumed weights and biases that can achieve this:, satisfying the equation of lines. Whenever the line crosses below the x axis, that neuron becomes inactive. In certain intervals several of the lines are above y=0, meaning several neurons are active in that region.

Hidden Layer Weights and Biases

1. W_{H1} = —2, b_{H1} = 8

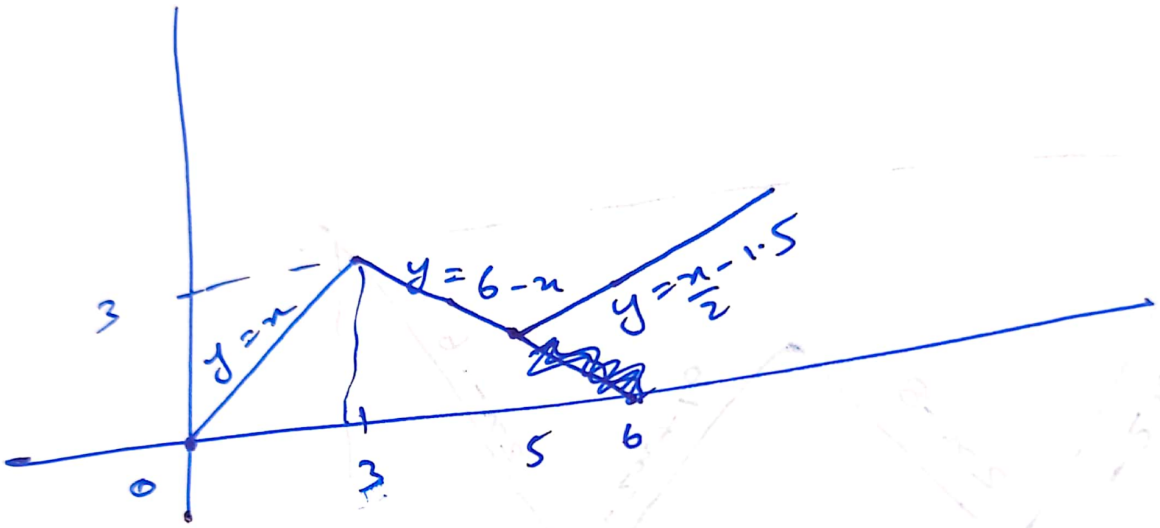
2. W_{H2} = 2, b_{H2} = -8

3. W_{H3} = -2, b_{H3} = 24

4. W_{H4} = 2, b_{H4} = -16

Output Layer Weights and Bias
1. W_{O1} =1
2. W_{O2} =1
3. W_{O3} =1
4. W_{O4} =1
5. Bias: b_0 =0

Dataset 2:



1. $W_{H1} = 1$, $b_{H1} = 0$

2. $W_{H2} = -1$, $b_{H2} = 6$

3. $W_{H3} = 0.5$, $b_{H3} = -1.5$
cycle.

4. $W_{H4} = 0$, $b_{H4} = 0$

Output Layer Weights and Bias
1. $W_{O1} = 1$

2. W_{O2} =1
3. W_{O3} =1
4. W_{O4} =1
5. Bias: b_0 =0

3.

B)

$$L(\vec{\theta}) = \sum_{i=1}^{i=n} (y_i - f(x_i, \vec{\theta}))^2$$

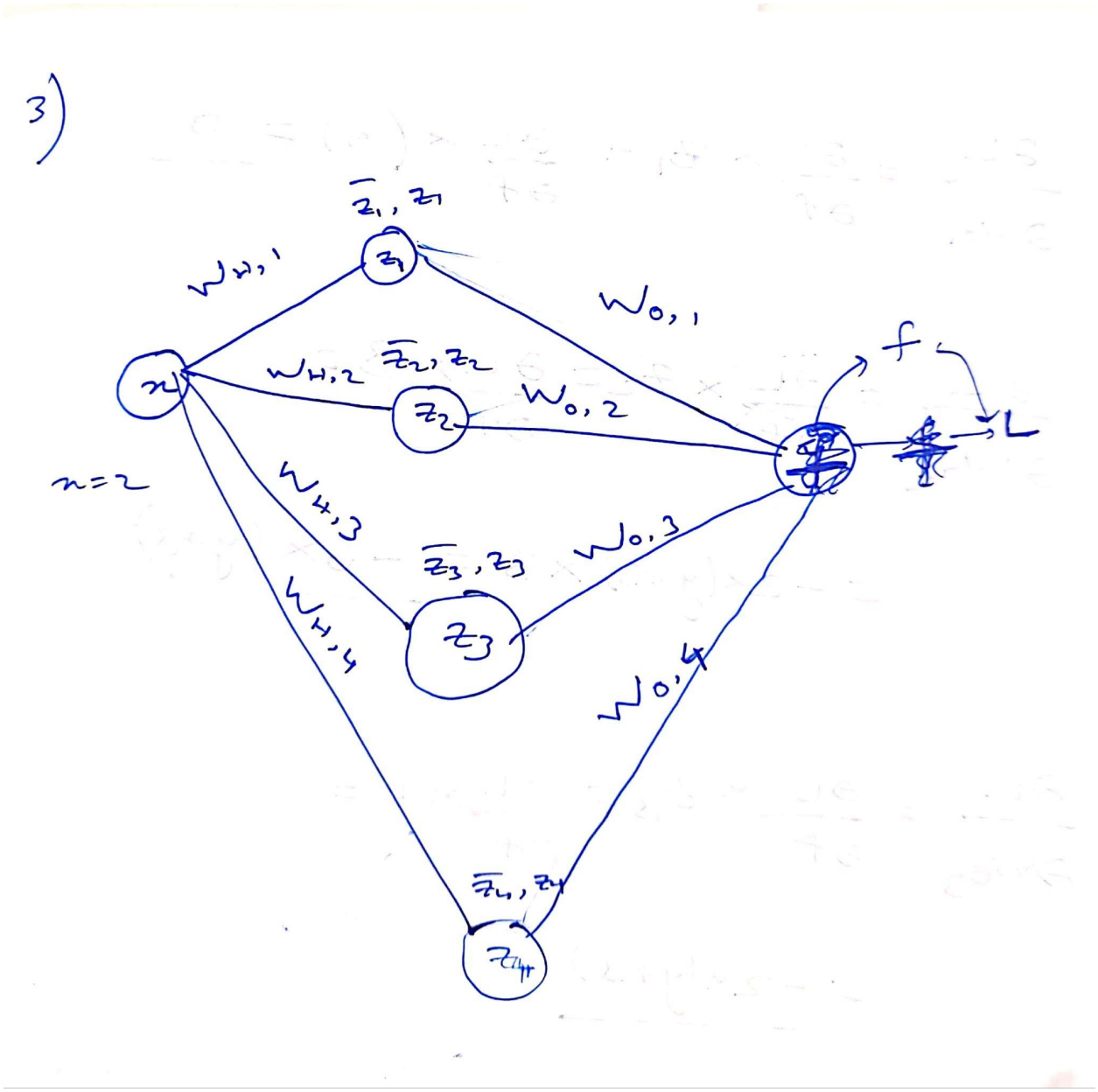
$$\frac{\partial L(\vec{\theta})}{\partial \vec{\theta}} = 2 * \sum_{i=1}^{i=n} (y_i - f(x_i, \vec{\theta}) * (0 - \frac{\partial f(x_i, \vec{\theta})}{\partial \vec{\theta}})$$

$$\frac{\partial L(\vec{\theta})}{\partial \vec{\theta}} = -2 * \sum_{i=1}^{i=n} (y_i - f(x_i, \vec{\theta}) * (\frac{\partial f(x_i, \vec{\theta})}{\partial \vec{\theta}})$$

$$\nabla L(\vec{\theta}) = -2 * \sum_{i=1}^{i=n} (y_i - f(x_i, \vec{\theta}) * \nabla f(x_i, \vec{\theta})$$

C)

3)



C)

$$\overline{z_1} = W_{H1} * x + b_{H1} = -1 * 2 + 1 = -1$$

$$\overline{z_2} = W_{H2} * x + b_{H2} = 1 * 2 + 1 = 3$$

$$\overline{z_3} = W_{H3} * x + b_{H3} = 1 * 2 - 1 = 1$$

$$\overline{z_4} = W_{H4} * x + b_{H4} = -1 * 2 + 1 = -1$$

$$z_1 = max(0, -1) = 0$$

$$z_2 = max(0, 3) = 3$$

$$z_3 = max(0, 1) = 1$$

$$z_4 = max(0, -1) = 0$$

$$f = W_{O1} * Z_1 + W_{O2} * Z_2 + W_{O3} * Z_3 + W_{O4} * Z_4 + b_O = 0 * -1 + 3 * -1 + 1 * -1 + 0 * 1 + 1$$

$$f = -3$$

3.

D)

We know that,

$$f = W_{O1} * Z_1 + W_{O2} * Z_2 + W_{O3} * Z_3 + W_{O4} * Z_4 + b_O$$

$$\frac{\partial f}{\partial W_{O1}} = z1 = 0$$

$$\frac{\partial f}{\partial W_{O2}} = z2 = 3$$

$$\frac{\partial f}{\partial W_{O3}} = z3 = 1$$

$$\frac{\partial f}{\partial W_{O4}} = z4 = 0$$

$$\frac{\partial f}{\partial B_0} = 1$$

$$\frac{\partial f}{\partial W_{H1}} = \frac{\partial f}{\partial Z_1} * \frac{\partial Z_1}{\partial \bar{Z}_1} * \frac{\partial \bar{Z}_1}{\partial W_{H1}}$$

We know from $f = W_{O1} * Z_1 + W_{O2} * Z_2 + W_{O3} * Z_3 + W_{O4} * Z_4 + b_O$ that

$$\frac{\partial f}{\partial Z_1} = W_{O1}$$

$\frac{\partial Z_1}{\partial \bar{Z}_1}$ = 1 if $Z_1 > 0$, else $\frac{\partial Z_1}{\partial \bar{Z}_1} = 0$

$$\frac{\partial \bar{Z}_1}{\partial W_{H1}} = x = 2$$

Substituting for all the below calculations,

$$\frac{\partial f}{\partial W_{H1}} = W_{O1} * 0 * 2 = 0$$

Similarly,

$$\frac{\partial f}{\partial W_{H2}} = \frac{\partial f}{\partial Z_2} * \frac{\partial Z_2}{\partial \bar{Z}_2} * \frac{\partial \bar{Z}_2}{\partial W_{H2}}$$

$$\frac{\partial f}{\partial W_{H2}} = W_{O2} * 1 * 2 = -2$$

$$\frac{\partial f}{\partial W_{H3}} = \frac{\partial f}{\partial Z_3} * \frac{\partial Z_3}{\partial \bar{Z}_3} * \frac{\partial \bar{Z}_3}{\partial W_{H3}}$$

$$\frac{\partial f}{\partial W_{H3}} = W_{O3} * 1 * 2 = -2$$

$$\frac{\partial f}{\partial W_{H4}} = \frac{\partial f}{\partial Z_4} * \frac{\partial Z_4}{\partial \bar{Z}_4} * \frac{\partial \bar{Z}_4}{\partial W_{H4}}$$

$$\frac{\partial f}{\partial W_{H4}} = W_{O4} * 0 * 2 = 0$$

$$\frac{\partial f}{\partial b_{H1}} = \frac{\partial f}{\partial Z_1} * \frac{\partial Z_1}{\partial \bar{Z}_1} * \frac{\partial \bar{Z}_1}{\partial B_{H1}}$$

We know, $\frac{\partial f}{\partial Z_1}$ and $\frac{\partial Z_1}{\partial \bar{Z}_1}$ from above,

$$\frac{\partial \bar{Z}_1}{\partial b_{H1}} = 1 \; as \; \overline{z}_1 = W_{H1} * x + b_{H1}$$

$$\frac{\partial f}{\partial b_{H1}} = W_{O1} * 0 * 1 = 0$$

Similarly,

$$\frac{\partial f}{\partial b_{H2}} = \frac{\partial f}{\partial Z_2} * \frac{\partial Z_2}{\partial \bar{Z}_2} * \frac{\partial \bar{Z}_2}{\partial b_{H2}} = (-1).(1).(1) = -1$$

$$\frac{\partial f}{\partial b_{H3}} = \frac{\partial f}{\partial Z_3} * \frac{\partial Z_3}{\partial \bar{Z}_3} * \frac{\partial \bar{Z}_3}{\partial b_{H3}} = (-1).(1).(1) = -1$$

$$\frac{\partial f}{\partial b_{H4}} = \frac{\partial f}{\partial Z_4} * \frac{\partial Z_4}{\partial \bar{Z}_4} * \frac{\partial \bar{Z}_4}{\partial b_{H4}} = (1).(0).(1) = 0$$

Demo 1 using Fashion MNIST Dataset

```
# We'll start by importing some libraries
import numpy as np # Very helpful (and popular!) for mathematical
computation
import torch # The main deep learning library we'll use (there are
others...)
import torchvision # Specialized for vision tasks such as recognizing
digits
import matplotlib.pyplot as plt # Helpful for plotting things (like
our images!)
```

Loading fashion MNIST dataset

```
# Load train and test data

trainingdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',
train=True,download=True,transform=torchvision.transforms.ToTensor())
testdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',
train=False,download=True,transform=torchvision.transforms.ToTensor())
```

```
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-images-idx3-ubyte.gz to
./FashionMNIST/FashionMNIST/raw/train-images-idx3-ubyte.gz

100%|██████████| 26421880/26421880 [00:01<00:00, 17034793.15it/s]

Extracting ./FashionMNIST/FashionMNIST/raw/train-images-idx3-ubyte.gz
to ./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/train-labels-idx1-ubyte.gz to
./FashionMNIST/FashionMNIST/raw/train-labels-idx1-ubyte.gz

100%|██████████| 29515/29515 [00:00<00:00, 274445.34it/s]

Extracting ./FashionMNIST/FashionMNIST/raw/train-labels-idx1-ubyte.gz
to ./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-images-idx3-ubyte.gz to
./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-ubyte.gz

100%|██████████| 4422102/4422102 [00:00<00:00, 5084219.67it/s]
```

```
Extracting ./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
to ./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-
1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz

100%|██████████| 5148/5148 [00:00<00:00, 19540522.16it/s]

Extracting ./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
to ./FashionMNIST/FashionMNIST/raw
```

Let's take a look at how much training and test data we have.

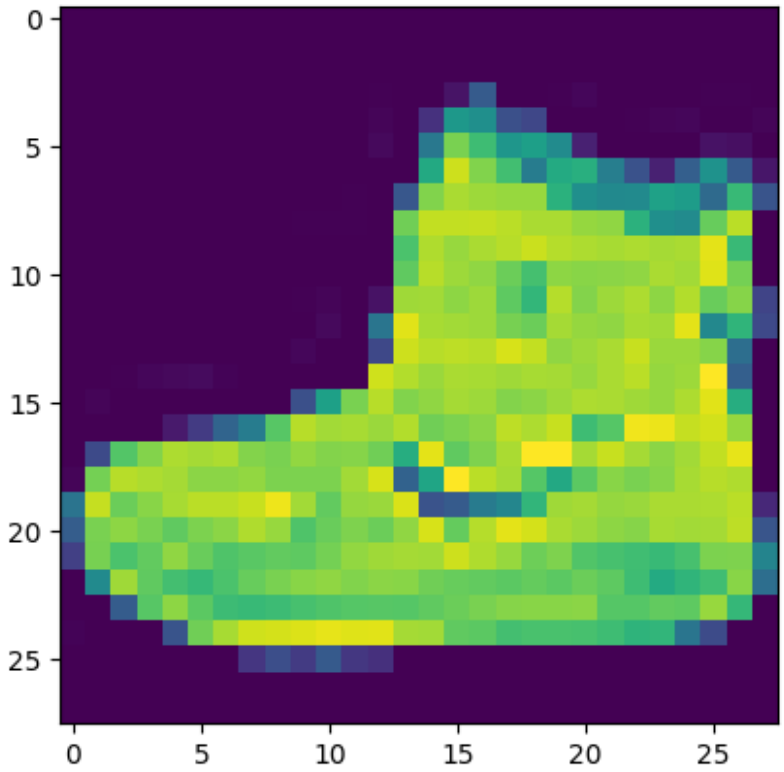
```
print(f'There are {len(trainingdata)} training images.')
print(f'There are {len(testdata)} test images.')

There are 60000 training images.
There are 10000 test images.
```

That's a lot! But perfect for our favorite big data, deep learning setting. Let's look inside to see one of these examples.

```
image, label = trainingdata[0]
print(image.shape) # A 1x28x28 image. This means one color and on a
grid of 28x28 pixels.
print(label) # A single number corresponding to the label of this
image.

torch.Size([1, 28, 28])
9
```

We can't directly plot the image because of formatting constraints that the first dimension can't have a size of 1. Instead, we'll use the `squeeze` function to turn it into a 28x28 image that we can plot.

```
plt.imshow(image.squeeze()) # Plot the 28x28 image
plt.show()
```



Let's define a handy way of loading the data.

```
# The DataLoader is a nifty class for wrapping the data so we can load batches easily
trainDataLoader = torch.utils.data.DataLoader(trainingdata,batch_size=64,shuffle=True)
testDataLoader = torch.utils.data.DataLoader(testdata,batch_size=64,shuffle=False)

images, labels = next(iter(trainDataLoader)) # Let's take a look at an example batch
print(images.shape) # There are 64 images
print(labels.shape) # And the corresponding 64 labels
```

```
torch.Size([64, 1, 28, 28])
torch.Size([64])
```

Now that we have our data, let's define our model, loss function, and optimizer using our three step recipe for machine learning.

```
import torch
import torch.nn as nn

class Model(nn.Module):
    def __init__(self):
```

```
        super(Model, self).__init__()
        self.fc1 = nn.Linear(28*28, 256)  # First hidden layer with
256 neurons
        self.fc2 = nn.Linear(256, 128)    # Second hidden layer with
128 neurons
        self.fc3 = nn.Linear(128, 64)     # Third hidden layer with 64
neurons
        self.fc4 = nn.Linear(64, 10)      # Output layer with 10
neurons for classification

    def forward(self, x):
        x = x.view(-1, 28*28)  # Flatten the input from 28x28 to 784
        x = torch.relu(self.fc1(x))  # Apply ReLU activation to the
first hidden layer
        x = torch.relu(self.fc2(x))  # Apply ReLU activation to the
second hidden layer
        x = torch.relu(self.fc3(x))  # Apply ReLU activation to the
third hidden layer
        x = self.fc4(x)  # Output layer
        return x

# Step 1: Create the model
model = Model()

# Step 2: Define the loss function
loss = nn.CrossEntropyLoss()

# Step 3: Define the optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

We also want to run everything quickly on a GPU. A GPU enables us to (basically) to matrix multiplication fast.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'We\'re using the {device}.')
# Put the model on the device
model = model.to(device)
```

```
We're using the cuda.
```

Now let's do some training!! We'll treat the training process as a bit of a black box for now.

As we go, we'll keep track of the training and test loss so we can visualize how they change later.

```
train_losses = []
test_losses = []

for epoch in range(25): # We'll train for 5 "epochs"
```

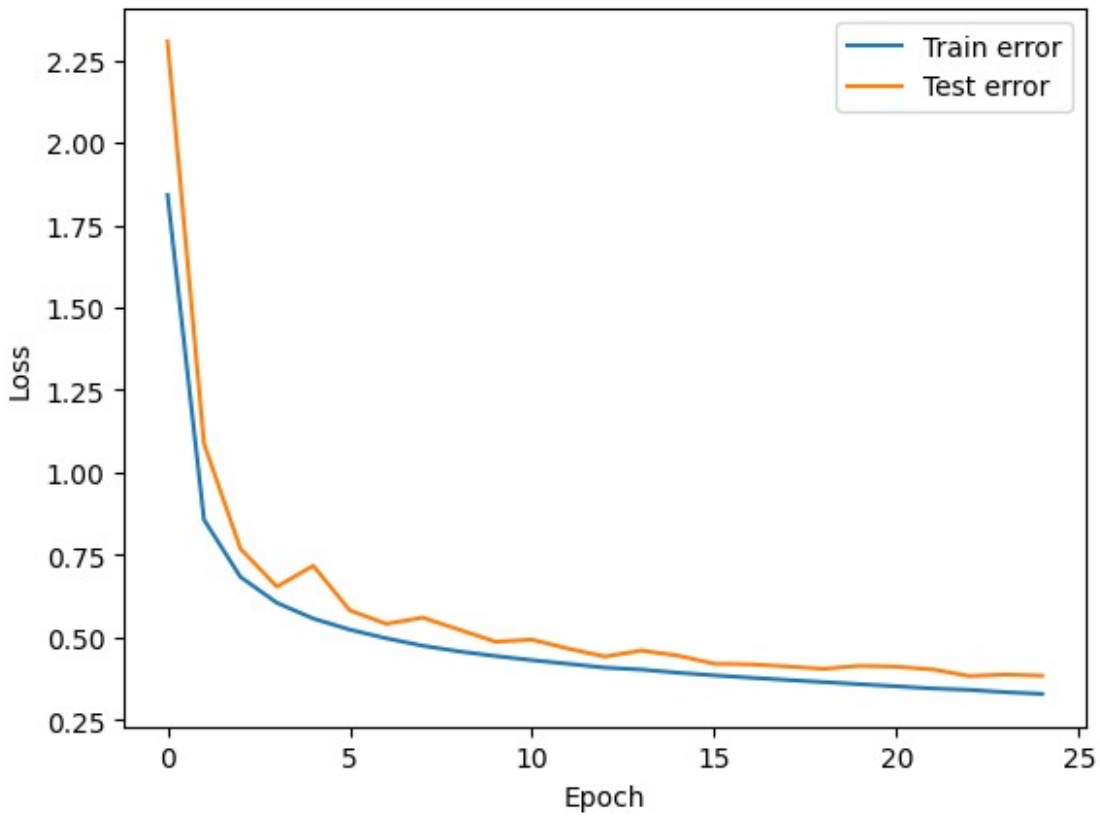
```
  train_loss = 0
  test_loss = 0

  # Evaluation process
  for i, data in enumerate(testDataLoader):
    images, labels = data # Unpack the data into the images and labels
    images, labels = images.to(device), labels.to(device) # Put on
device
    predicted_output = model(images) # Apply our network to the images
    fit = loss(predicted_output, labels) # Measure how well the
predicted output matches the labels
    test_loss += fit.item() # Add the fit to the loss for tracking
purposes

  # Training process
  for i, data in enumerate(trainDataLoader):
    images, labels = data # Unpack the data into the images and labels
    images, labels = images.to(device), labels.to(device) # Put on
device
    optimizer.zero_grad() # Zero out the gradient values
    predicted_output = model(images) # Apply our network to the images
    fit = loss(predicted_output, labels) # Measure how well the
predicted output matches the labels
    fit.backward() # Compute the gradient of the fit with respect to
the model parameters
    optimizer.step() # Update the weights in the model using gradient
descent
    train_loss += fit.item() # Add the fit to the loss for tracking
purposes

  # Add the current losses to our tracking lists
  train_losses += [train_loss/len(trainDataLoader)]
  test_losses += [test_loss/len(testDataLoader)]

  # Print the current loss
  print(f'Epoch {epoch}, Train loss {train_loss}, Test loss
{test_loss}')
```

```
Epoch 0, Train loss 1727.3545330166817, Test loss 362.1974289417267
Epoch 1, Train loss 803.6327338814735, Test loss 170.73893213272095
Epoch 2, Train loss 640.9106209874153, Test loss 120.69214868545532
Epoch 3, Train loss 567.1938232183456, Test loss 102.614536434412
Epoch 4, Train loss 522.3575262725353, Test loss 112.55262008309364
Epoch 5, Train loss 490.9789659976959, Test loss 91.36244958639145
Epoch 6, Train loss 466.2319046407938, Test loss 84.91656944155693
Epoch 7, Train loss 445.09929390251637, Test loss 87.89059269428253
Epoch 8, Train loss 428.94404873251915, Test loss 82.1755399107933
Epoch 9, Train loss 415.8034529387951, Test loss 76.40750846266747
Epoch 10, Train loss 404.05214984714985, Test loss 77.41713863611221
Epoch 11, Train loss 393.6788412630558, Test loss 73.10791577398777
```

```
Epoch 12, Train loss 382.90175135433674, Test loss 69.37548460066319
Epoch 13, Train loss 377.4120763093233, Test loss 72.21525743603706
Epoch 14, Train loss 368.36826483905315, Test loss 69.80670140683651
Epoch 15, Train loss 360.87860859930515, Test loss 65.94491311907768
Epoch 16, Train loss 354.16249062120914, Test loss 65.63785474002361
Epoch 17, Train loss 347.90206388384104, Test loss 64.6513214558363
Epoch 18, Train loss 341.803239941597, Test loss 63.52241359651089
Epoch 19, Train loss 335.713591337204, Test loss 64.93441192805767
Epoch 20, Train loss 329.76467844098806, Test loss 64.53870524466038
Epoch 21, Train loss 323.54288825392723, Test loss 63.232231855392456
Epoch 22, Train loss 319.68698064237833, Test loss 60.076358050107956
Epoch 23, Train loss 312.9808259680867, Test loss 60.80336445569992
Epoch 24, Train loss 308.2194154113531, Test loss 60.23711909353733
```

Plotting train and test Loss vs epochs

```
plt.plot(range(25),train_losses, label='Train error')
plt.plot(range(25),test_losses, label='Test error')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



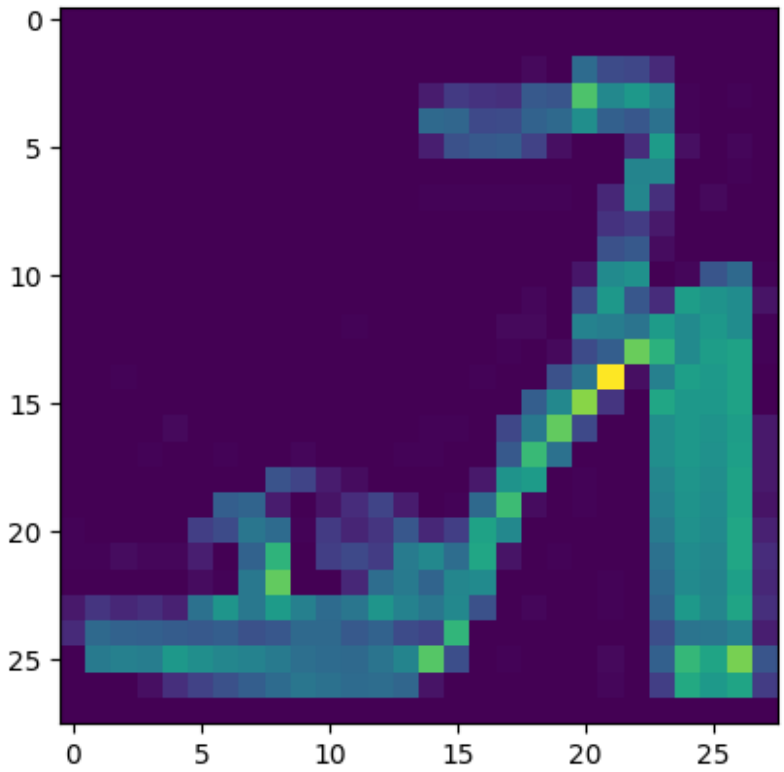
Let's look at an example to see how well the model does. We'll use the variables from the last pass of the training process.

```
predicted_classes = torch.max(predicted_output, 1)[1]
print('Predicted:', predicted_classes)
print('Labels:', labels)
```

```
Predicted: tensor([2, 7, 8, 5, 6, 5, 6, 5, 9, 2, 3, 3, 5, 3, 6, 1, 3,
6, 6, 3, 6, 0, 6, 2,
        5, 1, 3, 2, 0, 7, 9, 4], device='cuda:0')
Labels: tensor([2, 7, 8, 5, 0, 5, 6, 5, 9, 2, 4, 3, 5, 3, 6, 1, 6, 6,
6, 3, 4, 0, 6, 2,
        5, 1, 3, 2, 0, 7, 9, 4], device='cuda:0')
```

```
i = 5
print('Predicted:', predicted_classes[i].item())
print('Labels:', labels[i].item())
plt.imshow(images[i].squeeze().cpu()) # Visualize iamge
plt.show()
```

```
Predicted: 5
Labels: 5
```

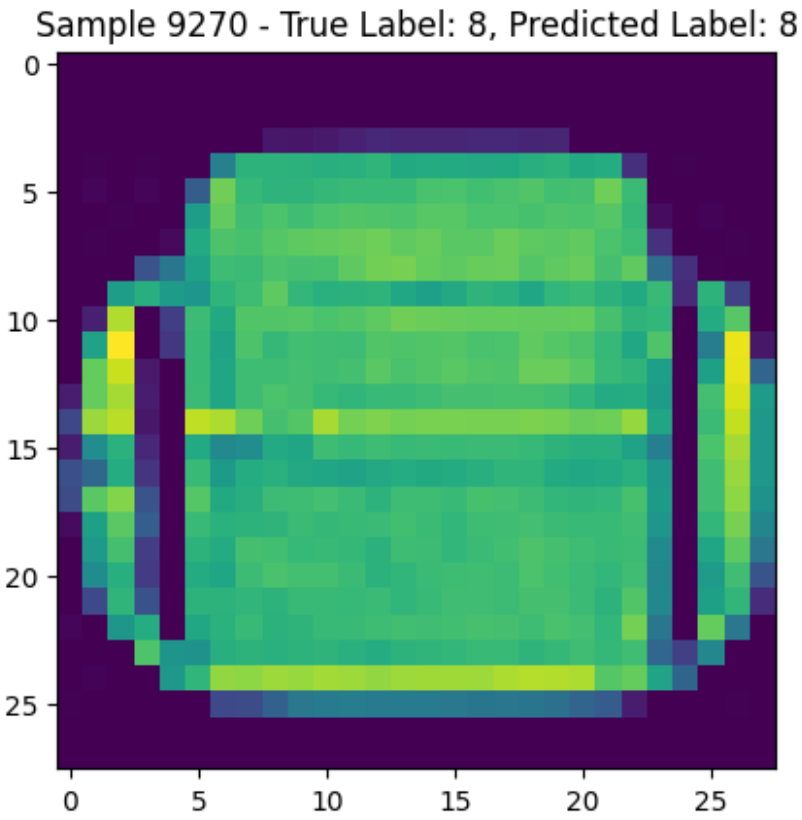


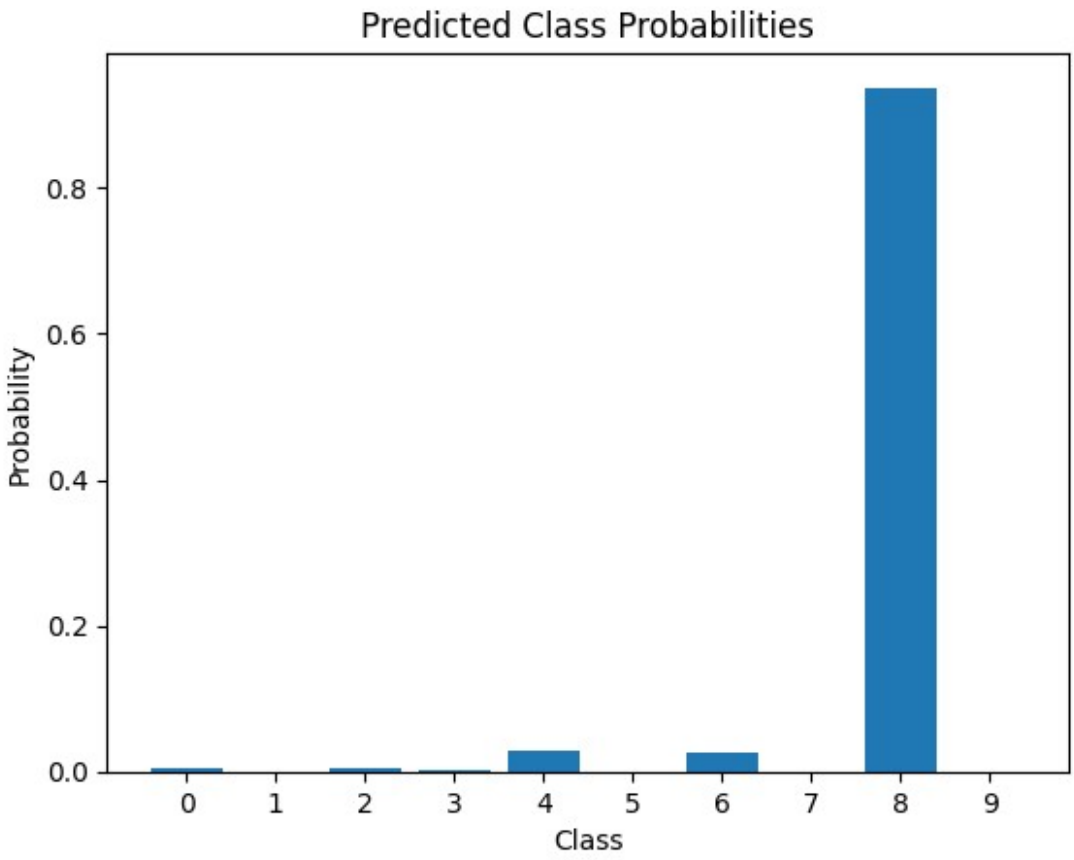
Random sampling from test data of 3 datapoints and visualizing probability distrubution for each sample

```
import torch
import torchvision
import torchvision.transforms as transforms
import random
import matplotlib.pyplot as plt

# Define the number of samples you want to randomly select
num_samples = 3

# Load FashionMNIST test dataset
testdata = torchvision.datasets.FashionMNIST(
    './FashionMNIST/', train=False, download=True,
transform=transforms.ToTensor())

# Create a list of indices from 0 to len(testdata) - 1
indices = list(range(len(testdata)))

# Randomly shuffle the indices
random.shuffle(indices)

# Select the first 'num_samples' indices from the shuffled list
sampled_indices = indices[:num_samples]

# Loop over the sampled indices to process the images
for i in sampled_indices:
    image, label = testdata[i]
    image = image.unsqueeze(0)  # Add batch dimension (1, C, H, W)

    # Transfer the image to the device if needed
    image = image.to(device)

    # Get predicted output from your model
    predicted_output = model(image)

    # Calculate class probabilities
    predicted_probs = torch.softmax(predicted_output, dim=1)

    print(" Probability values:", predicted_probs)


    # Get the predicted class
    predicted_class = torch.argmax(predicted_output, dim=1).item()

    # Display the input image
    plt.imshow(image.squeeze().cpu()) # Visualize iamge
    plt.title(f"Sample {i + 1} - True Label: {label}, Predicted Label:
{predicted_class}")

    # Create a bar graph for class probabilities
    classes = list(range(10))  # Assuming 10 classes for FashionMNIST
    plt.figure()
```

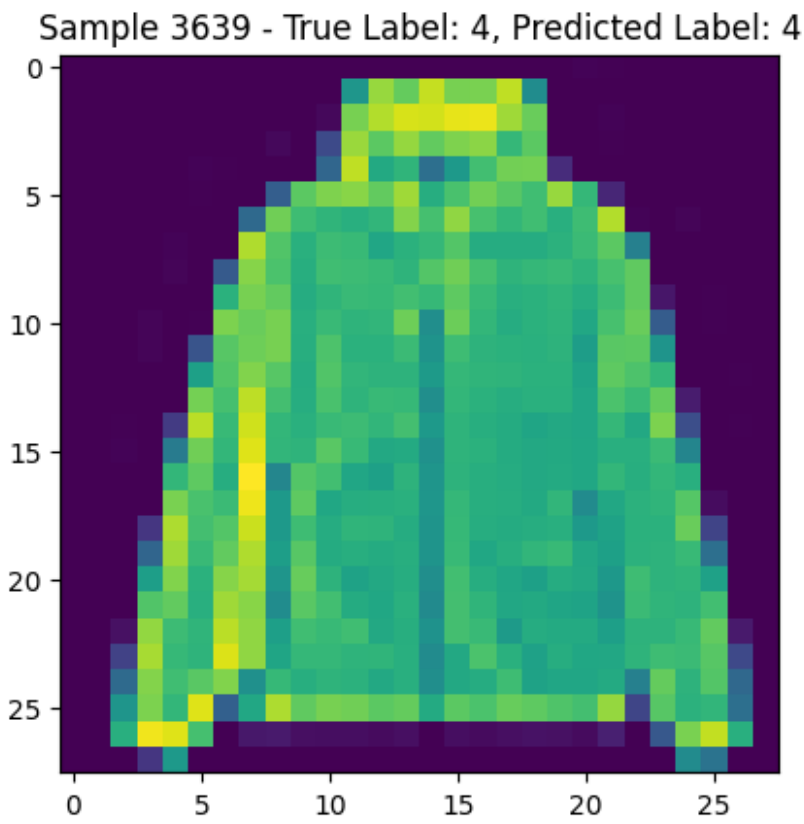
```
    plt.bar(classes, predicted_probs.squeeze().tolist())
    plt.xticks(classes)
    plt.xlabel('Class')
    plt.ylabel('Probability')
    plt.title('Predicted Class Probabilities')

    plt.show()


print('------------------------------------------------------------'*2)
```

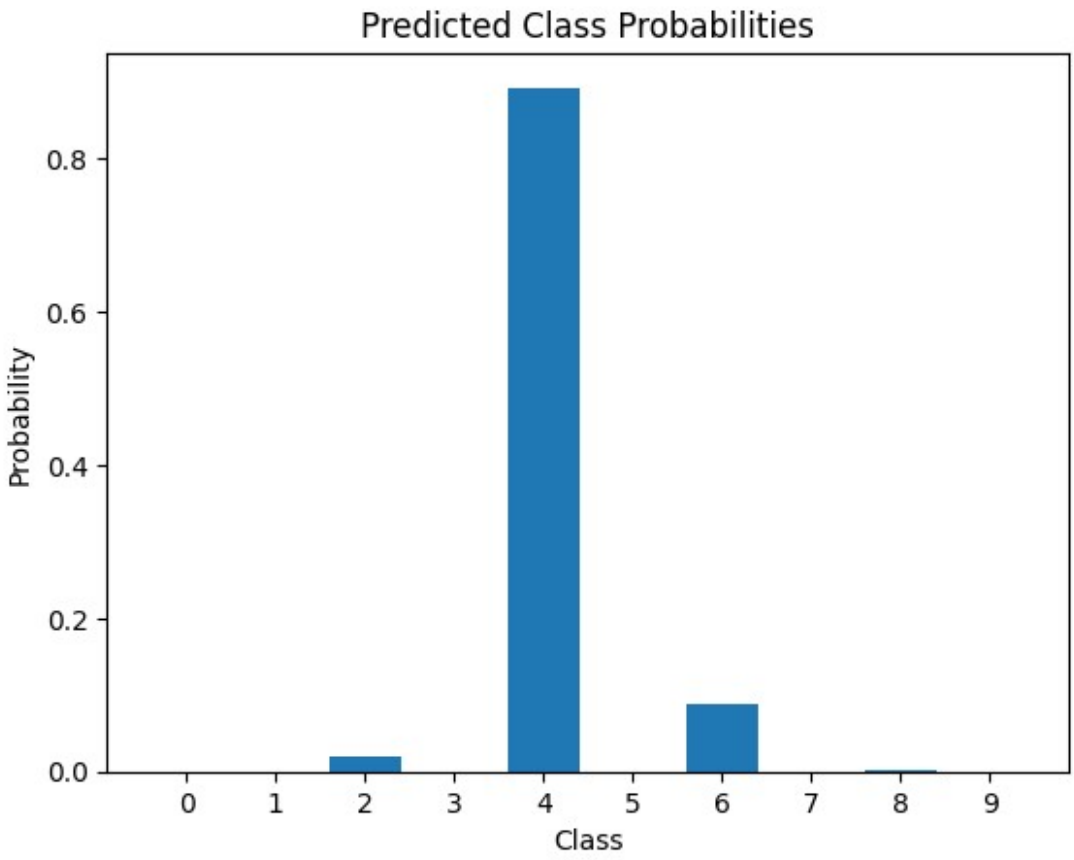
```
 Probability values: tensor([[4.2715e-03, 8.6439e-05, 4.0099e-03,
8.0646e-04, 2.8411e-02, 2.6288e-06,
         2.5030e-02, 9.6806e-06, 9.3736e-01, 1.0682e-05]],
device='cuda:0',
       grad_fn=<SoftmaxBackward0>)
```



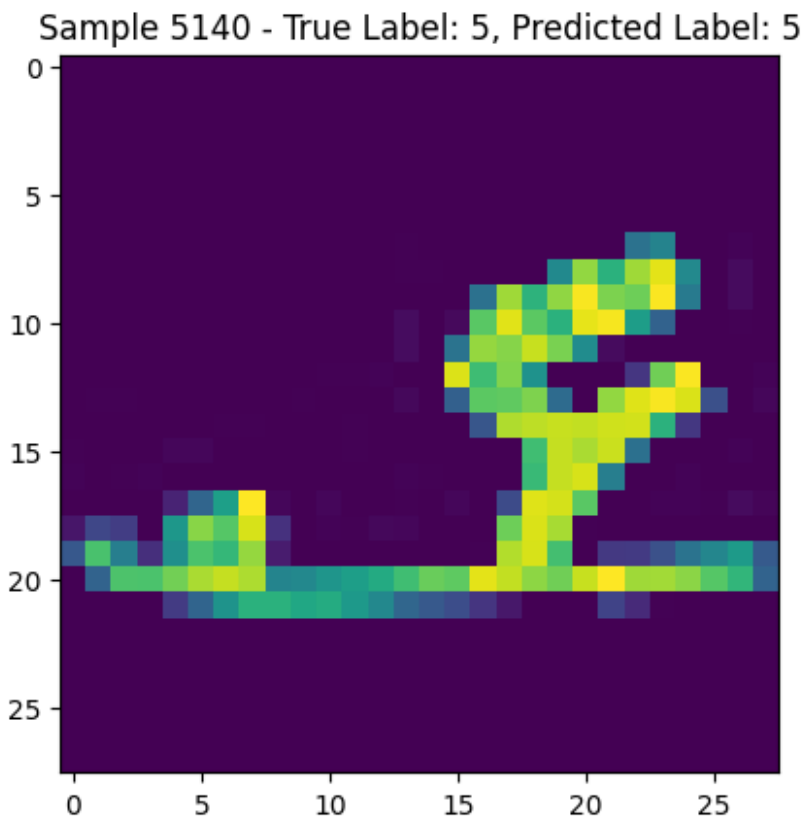


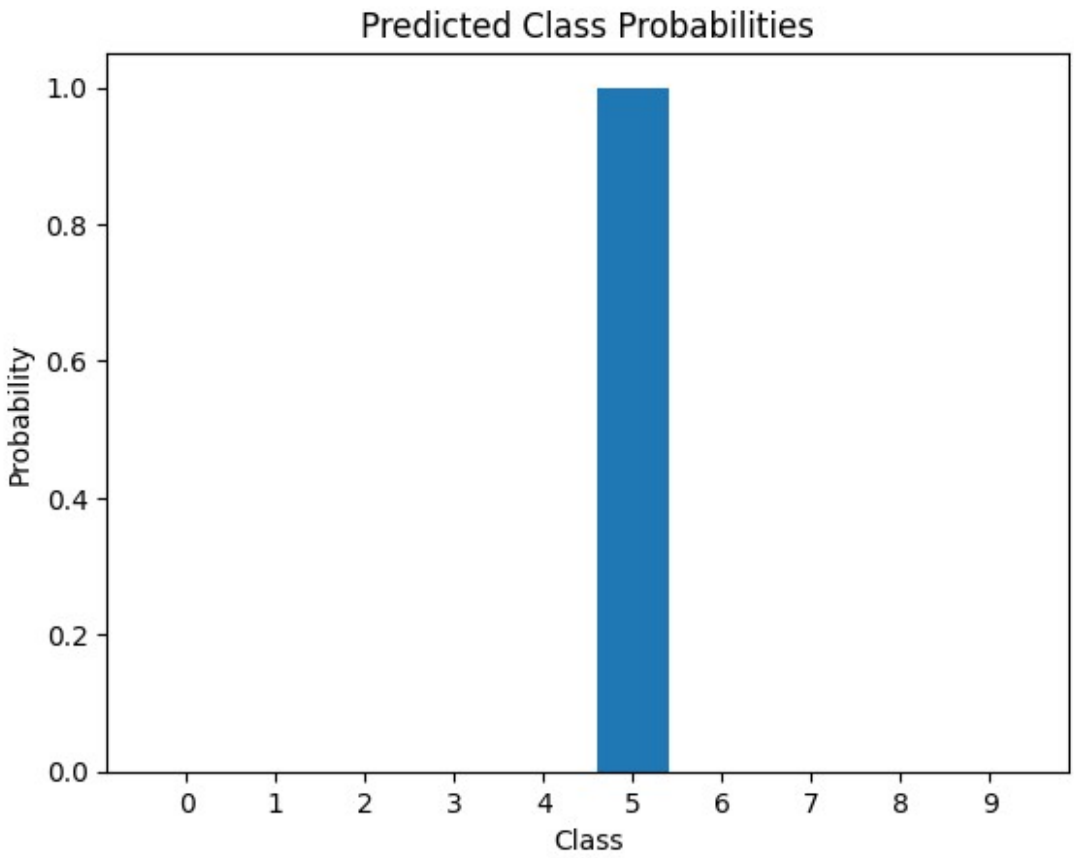

```
--------------------------------------------------------------------------------
 Probability values: tensor([[6.3994e-06, 3.2346e-06, 1.8729e-02,
9.5252e-06, 8.9233e-01, 1.7414e-06,
         8.8100e-02, 6.4457e-10, 8.2018e-04, 5.5695e-12]],
device='cuda:0',
       grad_fn=<SoftmaxBackward0>)
```


Sample 3639 - True Label: 4, Predicted Label: 4
0
5
10
15
20
25
0
5
10
15
20
25




```
--------------------------------------------------------------------------------
 Probability values: tensor([[4.0163e-05, 1.4879e-06, 1.0759e-05,
4.8009e-06, 3.4188e-06, 9.9946e-01,
         1.1498e-04, 7.7425e-05, 2.3312e-04, 5.7643e-05]],
device='cuda:0',
       grad_fn=<SoftmaxBackward0>)
```


Sample 5140 - True Label: 5, Predicted Label: 5
0
5
10
15
20
25
0
5
10
15
20
25




```
----------------------------------------------------------------------------------------------------
```

Comment:

In sample 9270, class 8 clearly has a higher probability prediction than rest of the classes

In sample 3639, class 4 clearly has a higher probability prediction than rest of the classes

In sample 5140, class 5 clearly has a higher probability prediction than rest of the classes

This is true as when one probability is high, other classes must be low since they add upto 1.

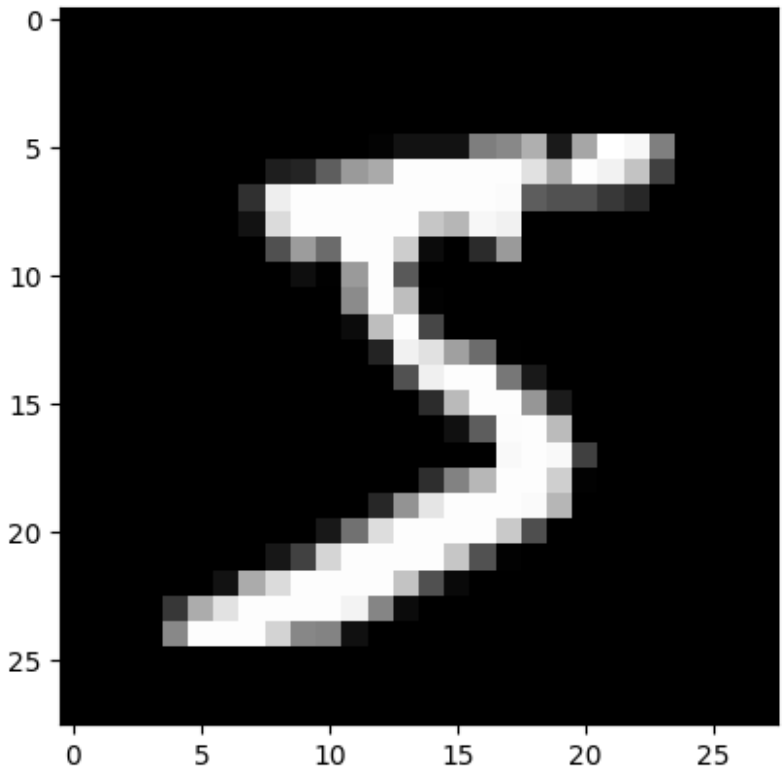
In our sample space, there is no confusion between classifying classes. clearly one class dominates others. which can be a good sign since this means test loss is low atleast for this sample space.

In this problem we will train a neural network from scratch using numpy. In practice, you will never need to do this (you'd just use TensorFlow or PyTorch). But hopefully this will give us a sense of what's happening under the hood.

For training/testing, we will use the standard MNIST benchmark consisting of images of handwritten images.

In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations directly. Instead of using autodiff, you will manually compute the gradient of the loss function for each parameter.

```
import tensorflow as tf
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.mnist.load_data(path="mnist.npz")

plt.imshow(x_train[0],cmap='gray');
```



Loading MNIST is the only place where we will use TensorFlow; the rest of the code will be pure numpy.

Let us now set up a few helper functions. We will use sigmoid activations for neurons, the softmax activation for the last layer, and the cross entropy loss.

```
import numpy as np

def sigmoid(x):
  # Numerically stable sigmoid function based on
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-
trick/

  x = np.clip(x, -500, 500) # We get an overflow warning without this

  return np.where(
    x >= 0,
    1 / (1 + np.exp(-x)),
    np.exp(x) / (1 + np.exp(x))
  )

def dsigmoid(x): # Derivative of sigmoid
  return sigmoid(x) * (1 - sigmoid(x))

def softmax(x):
  # Numerically stable softmax based on (same source as sigmoid)
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-
trick/
  b = x.max()
  y = np.exp(x - b)
  return y / y.sum()

def cross_entropy_loss(y, yHat):
  return -np.sum(y * np.log(yHat))

def integer_to_one_hot(x, max):
  # x: integer to convert to one hot encoding
  # max: the size of the one hot encoded array
  result = np.zeros(10)
  result[x] = 1
  return result
```

OK, we are now ready to build and train our model. The input is an image of size 28x28, and the output is one of 10 classes. So, first:

Q1. Initialize a 2-hidden layer neural network with 32 neurons in each hidden layer, i.e., your layer sizes should be:

784 -> 32 -> 32 -> 10

If the layer is $n_{in} \times n_{out}$ your layer weights should be initialized by sampling from a normal distribution with mean zero and variance $1/max\left(n_{in}, n_{out}\right)$.

```
import math

# Initialize weights of each layer with a normal distribution of mean
```

```
0 and
# standard deviation 1/sqrt(n), where n is the number of inputs.
# This means the weighted input will be a random variable itself with
mean
# 0 and standard deviation close to 1 (if biases are initialized as 0,
standard
# deviation will be exactly 1)

from numpy.random import default_rng

rng = default_rng(80085)

# Q1. Fill initialization code here.
# ...


# weights are initialised as per the question and biases to all zero
weights = [
  rng.normal(0, 1/math.sqrt(max(784,32)), (784, 32)),
  rng.normal(0, 1/math.sqrt(max(32,32)), (32, 32)),
  rng.normal(0, 1/math.sqrt(max(32,10)), (32, 10)),

]
biases = [np.zeros((1,32)), np.zeros((1,32)), np.zeros((1,10))]

# extra

print(weights)
print(biases)
```

```
[array([[-0.00826663,  0.01740251, -0.01155122, ...,  0.03339308,
        -0.01628035,  0.00304262],
       [-0.01750653,  0.00246254,  0.0052245 , ...,  0.03231726,
        -0.05517854, -0.02622419],
       [ 0.01513449,  0.02976356,  0.02213578, ...,  0.03511572,
         0.01412661, -0.0195296 ],
       ...,
       [-0.02997337,  0.01828714, -0.04305112, ..., -0.01901253,
        -0.01224845, -0.00538314],
       [-0.07465027,  0.03689759, -0.00464901, ..., -0.05838515,
         0.07805863, -0.00754196],
       [ 0.0029482 ,  0.05713184, -0.0343841 , ...,  0.03585718,
         0.00693053, -0.01106129]]), array([[ 0.20411041,  0.16573658,
-0.01178314, ...,  0.0506476 ,
         0.13878533, -0.05569114],
       [-0.17210972, -0.19314677, -0.18917497, ...,  0.01669375,
        -0.0446299 , -0.22262393],
       [-0.1362278 ,  0.08886137,  0.18602144, ..., -0.01772237,
         0.01482264,  0.05075596],
       ...,
```

```
       [ 0.0061864 ,  0.21454234,  0.07365931, ...,  0.13785818,
        -0.15618541, -0.0315416 ],
       [-0.23872604,  0.10417256,  0.29570148, ...,  0.20105108,
         0.0150497 ,  0.41545658],
       [ 0.0661201 , -0.02127139,  0.05551189, ...,  0.08884615,
        -0.21223365, -0.10568208]]), array([[ 0.06667852, -0.12305605,
0.1886475 ,  0.00129612, -0.01248619,
         0.06938735,  0.0006548 , -0.09539658, -0.21513325, -
0.11880947],
       [-0.03150972,  0.1428765 ,  0.14386447,  0.06791716, -
0.46984052,
         0.08087086,  0.11044014,  0.11470577,  0.02369094, -
0.03322626],
       [-0.12950165,  0.12804389,  0.00992893,  0.15368709,
0.05053986,
        -0.21175006, -0.01365356,  0.30065459,  0.29706225,
0.12195563],
       [-0.03517588,  0.05204323, -0.17820966,  0.1904392 ,
0.05568819,
         0.17588379,  0.16729606, -0.49713414, -0.05137462,
0.14421643],
       [ 0.0190138 ,  0.04386751, -0.0591114 , -0.34516762, -
0.04214091,
         0.47931078,  0.09757416,  0.14901733, -0.29683775,
0.06980003],
       [-0.22122515, -0.04274778, -0.07324473, -0.00412583,
0.07565026,
         0.06422773,  0.12428442, -0.02562716, -0.05343189,
0.06286264],
       [ 0.21755729, -0.19539098, -0.13418914,  0.57551781,
0.14912714,
        -0.02069389,  0.09745102,  0.09071709, -0.09368606, -
0.11661075],
       [-0.07946047,  0.27900972,  0.06323323,  0.05687957, -0.1803402
,
        -0.07027253, -0.10312629,  0.01861472,  0.07507861, -0.1712809
],
       [-0.12383892,  0.16296936,  0.11235332,  0.389831  ,  0.2192455
,
         0.16942017,  0.10787054, -0.06422827, -0.02611312,
0.25100181],
       [ 0.21419595,  0.1642632 ,  0.09889733,  0.06572262, -0.4550455
,
        -0.08885739,  0.42808745, -0.32657372,  0.06634819,
0.38408785],
       [-0.07920064,  0.55774737, -0.02327088, -0.2353093 ,  0.2178017
,
         0.06516138, -0.21550099,  0.44608056, -0.07284695,
0.26202064],
```

```
       [-0.11981522,  0.08680361,  0.11865212, -0.15551864,
0.16616824,
        -0.08697369,  0.13603236,  0.03151822, -0.17562069, -
0.02746763],
       [ 0.04249061,  0.18391693, -0.02861405,  0.06432146,
0.15210413,
        -0.09234865, -0.2327168 ,  0.17337497,  0.17606313,
0.11718014],
       [-0.15020967, -0.08902679, -0.01283324, -0.1411443 ,
0.20373036,
        -0.02708593, -0.05258924, -0.04193949, -0.12212182, -
0.18983982],
       [ 0.09475102, -0.18793577,  0.22798587,  0.28897089,
0.06678324,
        -0.26330719,  0.2019765 ,  0.12038335,  0.05447484,
0.13283333],
       [-0.15222092, -0.04532919, -0.43390911,  0.05589933,
0.28320827,
         0.13982774,  0.1443502 ,  0.25509992,  0.27587453, -
0.04304269],
       [-0.25413207, -0.19616038,  0.11790309, -0.11917507,
0.04009574,
        -0.03038689, -0.1285437 ,  0.20112257,  0.14133733,
0.31882622],
       [-0.22426951,  0.3453804 ,  0.06623497, -0.17190061, -
0.08199583,
        -0.08945022,  0.163226  , -0.10709577, -0.04348015,
0.21528719],
       [-0.39423226,  0.04963487,  0.10029627, -0.0547637 ,
0.22034237,
         0.02794828,  0.13637415, -0.02711563,  0.00114306,
0.09214163],
       [ 0.05284844, -0.11997135,  0.31899406,  0.00865162,
0.30733793,
        -0.15324912, -0.10787428,  0.01340299, -0.04583321, -0.2012835
],
       [-0.35282274, -0.0870429 ,  0.14243413, -0.23054039,
0.02997779,
         0.2710754 ,  0.0043214 , -0.24409125,  0.12418991,
0.21496296],
       [ 0.11591245,  0.043367  , -0.13048866, -0.05952365, -
0.08192474,
         0.14061309,  0.33365244, -0.03177119, -0.1038939 ,
0.11877615],
       [ 0.11767792,  0.01964855,  0.27063737,  0.09315677, -
0.15960862,
         0.07963308,  0.04805784,  0.06811963, -0.14280632,  0.092552
],
       [ 0.12697075, -0.09061633, -0.04260626, -0.13169074,
```

```
0.26042976,
        -0.09505302, -0.29423987, -0.17541236,  0.01323028,  0.202825
],
       [-0.26698639,  0.19375272,  0.24618989, -0.17930884, -
0.06294202,
        -0.25996971, -0.17185415,  0.27107755,  0.22329456, -
0.16929151],
       [ 0.07150038,  0.05370818, -0.01522907,  0.04310103, -
0.16551037,
         0.11958714, -0.17701419, -0.12047302, -0.00476727,
0.21405513],
       [-0.21134573,  0.10839846, -0.04097207,  0.15521059,
0.06128219,
         0.03076064,  0.07616357,  0.11394959,  0.1206088 , -
0.25885276],
       [ 0.28686911,  0.05755687,  0.29144817,  0.04653277, -
0.12068839,
        -0.13755049,  0.03147542,  0.14373089,  0.19995382, -
0.04554261],
       [-0.11037302,  0.04056933, -0.02599068,  0.02035556, -0.1193677
,
         0.19599552, -0.01143023,  0.00397499,  0.17450331, -
0.03771648],
       [ 0.14572867, -0.15455643, -0.25175454,  0.04806895, -
0.06879288,
         0.04905893, -0.04063888, -0.02072746, -0.23275414, -
0.12584788],
       [ 0.07958356, -0.21616138, -0.25841157,  0.12665762, -
0.04654309,
        -0.01293728,  0.08694383,  0.31498051,  0.08496313,
0.06760091],
       [ 0.10435065,  0.17053191,  0.0053084 , -0.01121967,
0.07429579,
        -0.21791466, -0.05414794,  0.40976908, -0.06890102, -
0.03831953]])]
[array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.]]), array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.]]), array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])]
```

Next, we will set up the forward pass. We will implement this by looping over the layers and successively computing the activations of each layer.

Q2. Implement the forward pass for a single sample, and for the entire dataset.

Right now, your network weights should be random, so doing a forward pass with the data should not give you any meaningful information. Therefore, in the last line, when you calculate test accuracy, it should be somewhere around 1/10 (i.e., a random guess).

```
def feed_forward_sample(sample, y):
  """ Forward pass through the neural network.
    Inputs:
      sample: 1D numpy array. The input sample (an MNIST digit).
      label: An integer from 0 to 9.

    Returns: the cross entropy loss, most likely class
  """
  # Q2. Fill code here.
  # ...

  # reshaping of a
  a = np.reshape(sample,(1,28*28))

  # u corresponding to linar combination and z to application of
activation function to u in all layers
  u1 = np.dot(a,weights[0])+biases[0]
  z1 = sigmoid(u1)

  u2 = np.dot(z1,weights[1])+biases[1]
  z2 = sigmoid(u2)

  u3 = np.dot(z2,weights[2])+biases[2]

  # yhat is final predicted output
  yhat = softmax(u3)

  # one hot encoded vector form of label
  yvector = integer_to_one_hot(y,10)

  loss = cross_entropy_loss(yvector,yhat)
  one_hot_guess = integer_to_one_hot(np.argmax(yhat),10)


  return loss, one_hot_guess


def feed_forward_dataset(x, y):
  losses = np.empty(x.shape[0])
  one_hot_guesses = np.empty((x.shape[0], 10))


  # ...
  # Q2. Fill code here to calculate losses, one_hot_guesses
  # ...

  # looping over the entire dataset and forwarding each sample
```

```
  for i in range(x.shape[0]):
    sample = np.reshape(x[i],(1,28*28))
    losses[i], one_hot_guesses[i] = feed_forward_sample(sample, y[i])


  # True label vector for all examples in one hot vector form
  y_one_hot = np.zeros((y.size, 10))
  y_one_hot[np.arange(y.size), y] = 1

  # Accuracy calculation

  correct_guesses = np.sum(y_one_hot * one_hot_guesses)
  correct_guess_percent = format((correct_guesses / y.shape[0]) * 100,
".2f")

  print("\nAverage loss:", np.round(np.average(losses), decimals=2))
  print("Accuracy (# of correct guesses):", correct_guesses, "/",
y.shape[0], "(", correct_guess_percent, "%)")

def feed_forward_training_data():
  print("Feeding forward all training data...")
  feed_forward_dataset(x_train, y_train)
  print("")

def feed_forward_test_data():
  print("Feeding forward all test data...")
  feed_forward_dataset(x_test, y_test)
  print("")

feed_forward_test_data()
```

```
Feeding forward all test data...

Average loss: 2.37
Accuracy (# of correct guesses): 880.0 / 10000 ( 8.80 %)
```

OK, now we will implement the backward pass using backpropagation. We will keep it simple and just do training sample-by-sample (no minibatching, no randomness).

Q3: Compute the gradient of all the weights and biases by backpropagating derivatives all the way from the output to the first layer.

```
def train_one_sample(sample, y, learning_rate=0.003):
  #a = sample.flatten()

  # We will store each layer's activations to calculate gradient
  activations = []

  # intermediate variables
```

```
  weight_gradients = []
  bias_gradients = []

  # Forward pass

  # Q3. This should be the same as what you did in feed_forward_sample
above.
  # ...

  # Forward pass same as Q2. All intermediary values here used for
backprop calculation

  a = np.reshape(sample,(1,28*28))

  u1 = np.dot(a,weights[0])+biases[0]
  z1 = sigmoid(u1)

  u2 = np.dot(z1,weights[1])+biases[1]
  z2 = sigmoid(u2)

  u3 = np.dot(z2,weights[2])+biases[2]
  yhat = softmax(u3)

  yvector = integer_to_one_hot(y,10)

  loss = cross_entropy_loss(yvector,yhat)
  one_hot_guess = integer_to_one_hot(np.argmax(yhat),10)


  # Backward pass

  # Q3. Implement backpropagation by backward-stepping gradients
through each layer.
  # You may need to be careful to make sure your Jacobian matrices are
the right shape.
  # At the end, you should get two vectors: weight_gradients and
bias_gradients.
  # ...


  # Gradient calculation of loss; du3 stands for dL/du3

  # Layer 3 gradients

  du3 = yhat - yvector
  dw3 = z2.T.dot(du3)
  db3 = du3

  # Layer 2 gradients
```

```
  dz2 = du3.dot(weights[2].T)
  du2 = np.multiply(dz2,dsigmoid(u2))
  dw2 = z1.T.dot(du2)
  db2 = du2

  # Layer 1 gradients


  dz1= du2.dot(weights[1].T)
  du1 = np.multiply(dz1,dsigmoid(u1))
  dw1 = a.T.dot(du1)
  db1 = du1

  weight_gradients.append(dw1)
  weight_gradients.append(dw2)
  weight_gradients.append(dw3)


  bias_gradients.append(db1)
  bias_gradients.append(db2)
  bias_gradients.append(db3)







  # Update weights & biases based on your calculated gradient
  num_layers = 3
  for i in range(num_layers):
    weights[i] -= weight_gradients[i] * learning_rate
    biases[i] -= bias_gradients[i] * learning_rate
```

Finally, train for 3 epochs by looping over the entire training dataset 3 times.

Q4. Train your model for 3 epochs.

```
def train_one_epoch(learning_rate=0.003):
  print("Training for one epoch over the training dataset...")

  # Q4. Write the training loop over the epoch here.
  # ...
  for i in range(x_train.shape[0]):
    train_one_sample(x_train[i], y_train[i], learning_rate)

  print("Finished training.\n")


feed_forward_test_data()
```

```python
def test_and_train():
  train_one_epoch()
  feed_forward_test_data()

for i in range(3):
  test_and_train()
```

```
Feeding forward all test data...

Average loss: 2.37
Accuracy (# of correct guesses): 880.0 / 10000 ( 8.80 %)

Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.96
Accuracy (# of correct guesses): 6653.0 / 10000 ( 66.53 %)

Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.96
Accuracy (# of correct guesses): 6596.0 / 10000 ( 65.96 %)

Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.84
Accuracy (# of correct guesses): 7257.0 / 10000 ( 72.57 %)
```

That's it!

Your code is probably very time- and memory-inefficient; that's ok. There is a ton of optimization under the hood in professional deep learning frameworks which we won't get into.

If everything is working well, you should be able to raise the accuracy from ~10% to ~70% accuracy after 3 epochs.