Deep Learning Homework_1

Jinrui Zhang

February 17, 2024

Question 1

a.

The simple neural network with 2 hidden neurons is shown as followed. Assuming wights of each edge are w_1, w_2, w_3 and w_4, the hidden neurons are neuron A and neuron B with biases b_A and b_B. And as the question stated, the function $f(x)$ is: $f(x) = \begin{cases} h, & 0 < x < \delta \\ 0, & else \end{cases}$.

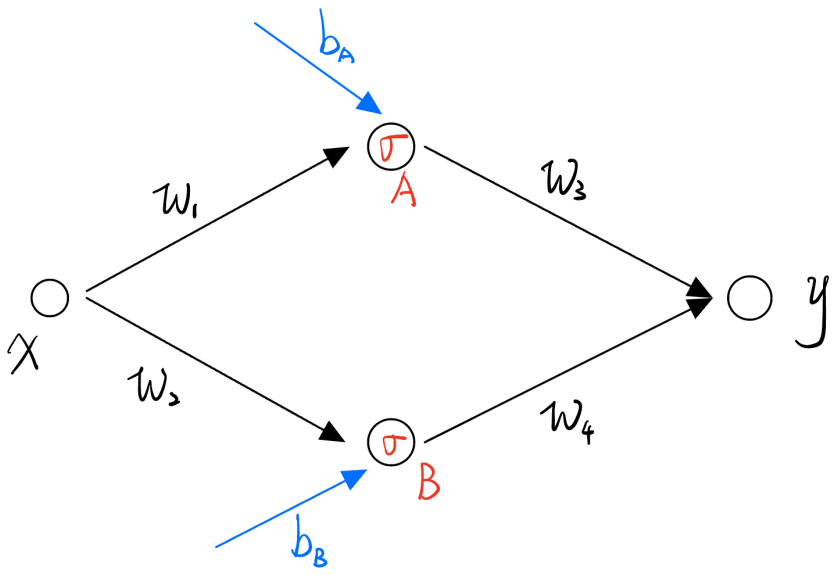


Figure 1: Simple Nuron Network

From the neuron network, we could deduce y, in other words, $f(x)$, as follows.

$$A = \sigma(w_1 + b_A)$$
$$B = \sigma(w_2 + b_B)$$

$$f(x) = w_3 A + w_4 B \tag{1}$$
$$= w_3\sigma(w_1 + b_A) + w_4\sigma(w_2 + b_B) \tag{2}$$

To make sure $f(x)$ can be expressed linearly, the $f(x)$ needs to be expressed as the difference between two step functions. In this case,

$$f(x) = h\sigma(x) - h\sigma(x - \delta) \tag{3}$$

where, $\sigma(x) = \begin{cases} 1, & 0 < x < \delta \\ 0, & else \end{cases}$.

In this case, all parameters in the equation (2) can be solved easily. Therefore,

$$\begin{cases} w_1 = 1 \\ w_2 = 1 \\ w_3 = h \\ w_4 = -h \\ b_A = 0 \\ b_B = -\delta \end{cases}$$

b.

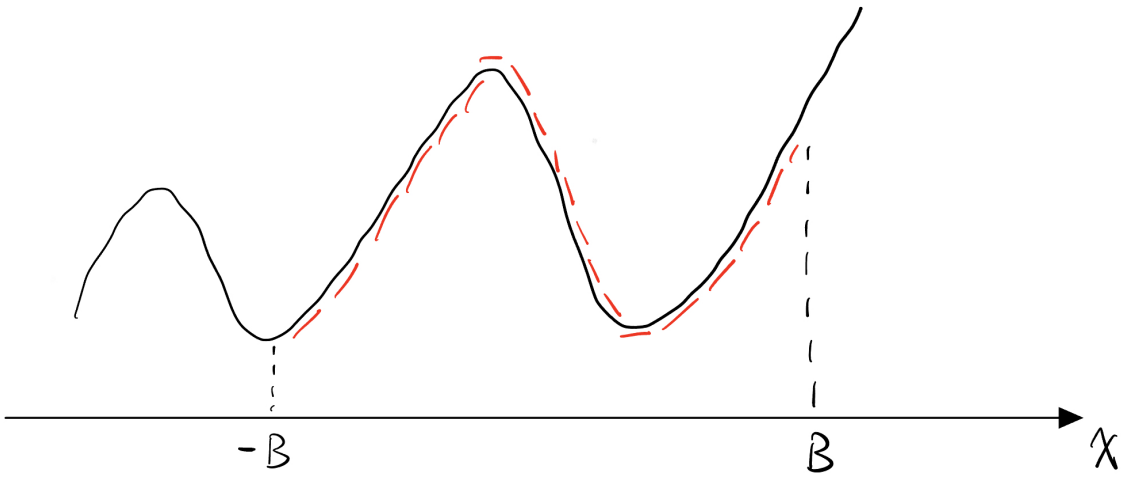


Figure 2: Approximate linearly

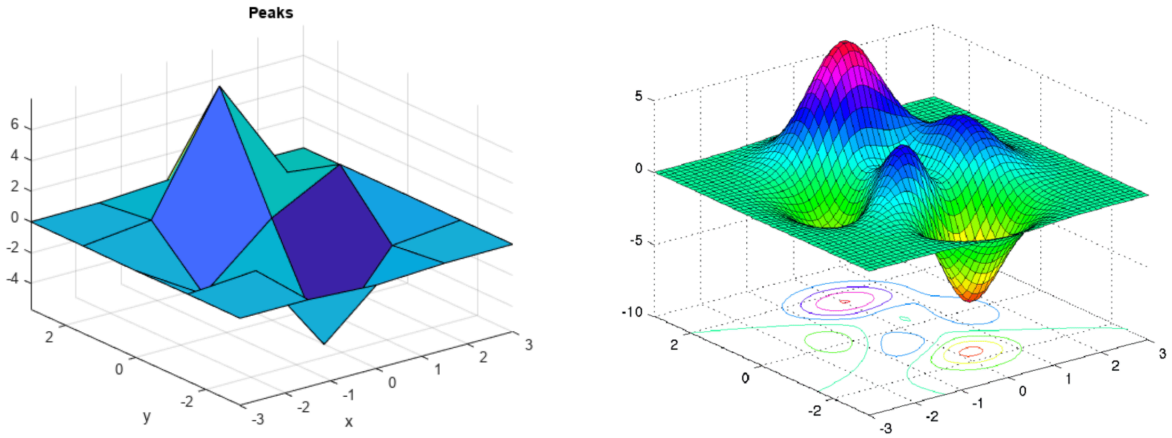
As part a indicated, function $f(x)$ can be expressed as:

$$f(x) = \sum_{i=1}^{n} \sigma_i(w_i x + b_i)$$

Function $f(x)$ could be chunked up into very small linear pieces, and each piece could be expressed by two different hidden neurons. As the figure 2 shows, if the number of red linear functions is large enough, the black nonlinear function can be approximated. Therefore, as long as the number of linear function is large enough, which means the width of the hidden layer is large enough, an arbitrary, smooth, bounded function can be approximated.

c.

The argument in part b can be extended to the case of multi-dimensional inputs.



(a) (b)

Figure 3: Different amount of division

We now assume d-dimensional inputs as two dimensions, which could express a curved surface. We chose to discuss this argument with the most common function in Matlab. From the figure above, as we divide the curved surface small enough, the approximate accuracy is getting better. In a neuron network that has two-dimensional inputs, every four hidden neurons can form a plane. As long as the network is wide enough, it can approximate a curved surface. However, in practice, the number of hidden neurons is too large to compute, so we need to balance the width and depth of a network.

Question 2

As the definition of softmax function:

$$y_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

taking logarithms o y_i, we have:

$$\log y_i = z_i - \log \sum_{j=1}^{n} e^{z_j}$$

if we take partial derivation of $\log y_i$, we have:

$$\frac{\partial \log y_i}{\partial z_i} = \frac{1}{y_i} \frac{\partial y_i}{\partial z_i}$$

Therefore, we have derivation of y_i respect to z_i, while $i = j$ we have:

$$\begin{aligned}\frac{\partial y_i}{\partial z_i} &= y_i \frac{\partial \log y_i}{\partial z_i} \\ &= y_i(1 - \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}) \\ &= y_i(1 - y_i)\end{aligned}$$

Likewise, while $i \neq j$, we have:

$$\begin{aligned}\frac{\partial y_i}{\partial z_j} &= y_i \frac{\partial \log y_i}{\partial z_j} \\ &= y_i(\frac{e^{z_j}}{\sum_{j=1}^{n} e^{z_j}}) \\ &= y_i y_j\end{aligned}$$

Therefore, the Jacobian of y with respect to z should be:

$$J_{ij} = y_i(\delta_{ij} - y_j)$$

where, $\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & else \end{cases}$

Question 3

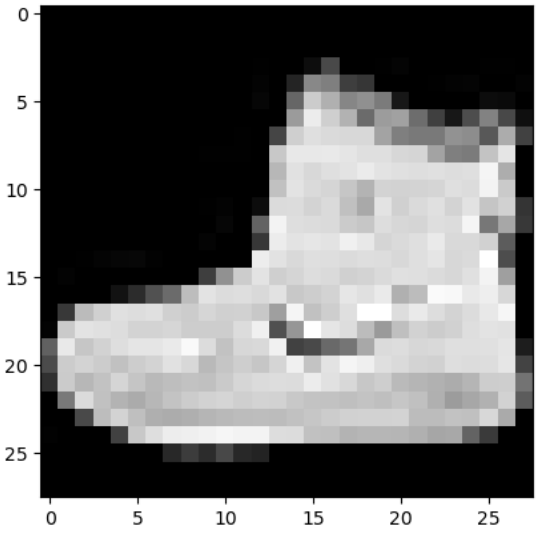
The code for question 3 is followed:

In [2]:
```
import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
import random
```

In [4]:
```
## download the data
trainingdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=True,download=True,transform=torchvi
testdata = torchvision.datasets.FashionMNIST('./FashionMNIST/',train=False,download=True,transform=torchvisio

## print length for checking if data downloaded
print(len(trainingdata))
print(len(testdata))

image, label = trainingdata[0]
plt.imshow(image.squeeze(), cmap=plt.cm.gray)
```

```
60000
10000
```

Out[4]:
```
<matplotlib.image.AxesImage at 0x1415da410>
```



In [5]:
```
# adding the training and test loader making it easy for us to iterate over the data repetitively in batches

trainDataLoader = torch.utils.data.DataLoader(trainingdata,batch_size=64,shuffle=True)
testDataLoader = torch.utils.data.DataLoader(testdata,batch_size=64,shuffle=False)
```

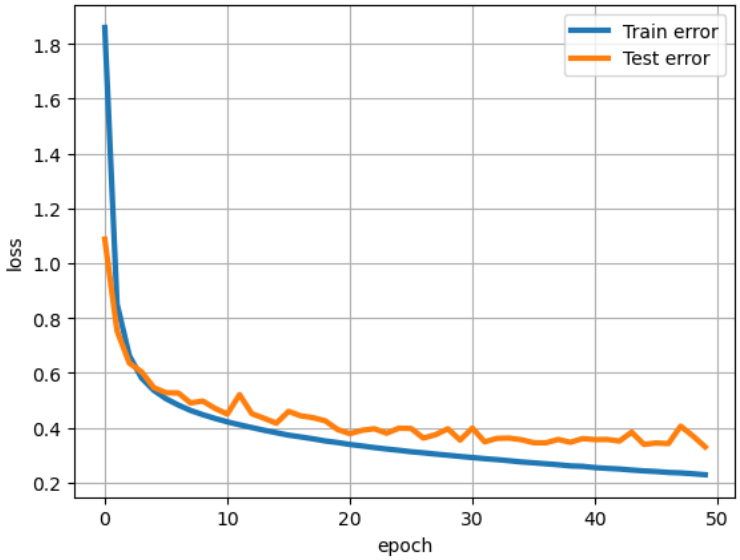
In [14]:
```
# Seting up model
class LogisticRegression(torch.nn.Module):
    def __init__(self):
        super(LogisticRegression, self).__init__()
        self.layer1 = torch.nn.Linear(784,256)  # input layer
        self.layer2 = torch.nn.Linear(256, 128)  # First hidden layer
        self.layer3 = torch.nn.Linear(128, 64)  # Second hidden layer
        self.layer4 = torch.nn.Linear(64, 10)  # fully connected
        self.ReLU = torch.nn.ReLU()  # ReLU activation


    def forward(self, x):
        x = x.view(-1, 784)
        x = self.ReLU(self.layer1(x))  # ReLU activation
        x = self.ReLU(self.layer2(x))  # ReLU activation
        x = self.ReLU(self.layer3(x))  # ReLU activation
        return self.layer4(x)  # output layer
```

In [52]:
```
# Perparation

# using 'cpu' since .cuda() is not available in Macbook.
device = torch.device("cpu")
model = LogisticRegression()  # Step 1: architecture
model.to(device)  # add specific device to model
loss = torch.nn.CrossEntropyLoss()  # Step 2: loss
loss.to(device)  # add specific device to loss
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)  # Step 3: training method
```

In [55]:
```
train_loss_history = []
test_loss_history = []

for epoch in range(50):
    train_loss = 0.0
    test_loss = 0.0

    # Training model
    model.train()
    for i, data in enumerate(trainDataLoader):
        images, labels = data
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()  # zero out any gradient values from the previous iteration
        predicted_output = model(images)  # forward propagation
        fit = loss(predicted_output, labels)  # calculate our measure of goodness
        fit.backward()  # backpropagation
        optimizer.step()  # update the weights of our trainable parameters
        train_loss += fit.item()

    # Testing model
    model.eval()
    for i, data in enumerate(testDataLoader):
        with torch.no_grad():
            images, labels = data
            images = images.to(device)
            labels = labels.to(device)
            predicted_output = model(images)
            fit = loss(predicted_output, labels)
            test_loss += fit.item()
    train_loss = train_loss / len(trainDataLoader)
    test_loss = test_loss / len(testDataLoader)
    train_loss_history += [train_loss]
    test_loss_history += [test_loss]
    print(f'Epoch {epoch}, Train loss {train_loss}, Test loss {test_loss}')
```

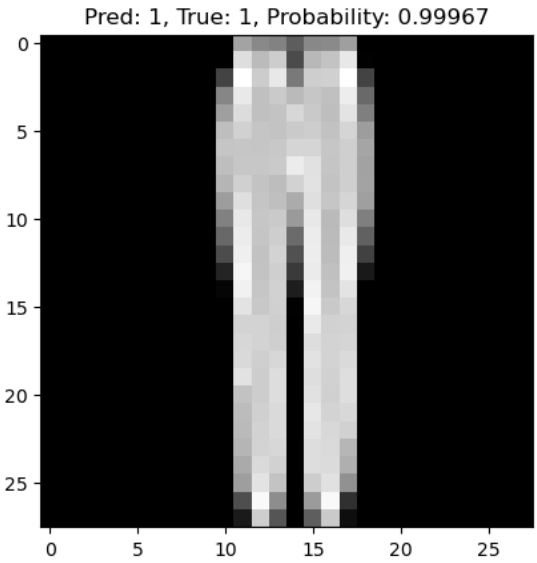
```
Epoch 0, Train loss 1.8599737423188143, Test loss 1.0879819894292553
Epoch 1, Train loss 0.8520451883263171, Test loss 0.7534975705632738
Epoch 2, Train loss 0.6633171050914569, Test loss 0.6357466899285651
Epoch 3, Train loss 0.5812871634070553, Test loss 0.6042404184295873
Epoch 4, Train loss 0.5361825011368754, Test loss 0.5463287737339165
Epoch 5, Train loss 0.5055548595403557, Test loss 0.5274235890929106
Epoch 6, Train loss 0.4826679655483791, Test loss 0.5267461208021564
Epoch 7, Train loss 0.4628188446767803, Test loss 0.4902096181918102
Epoch 8, Train loss 0.44742317486609984, Test loss 0.4972275843855682
Epoch 9, Train loss 0.4336468667935715, Test loss 0.47040568947032757
Epoch 10, Train loss 0.42122781155968525, Test loss 0.4498191614439533
Epoch 11, Train loss 0.41063440762666753, Test loss 0.5207532534174099
Epoch 12, Train loss 0.40048549778616505, Test loss 0.45100189935249885
Epoch 13, Train loss 0.3906617711729078, Test loss 0.4342965388753612
Epoch 14, Train loss 0.3816815967848306, Test loss 0.41644957785014136
Epoch 15, Train loss 0.37273259751641674, Test loss 0.45956142806703115
Epoch 16, Train loss 0.3661950723226391, Test loss 0.4431108928220287
Epoch 17, Train loss 0.35934858364082856, Test loss 0.43658396658624055
Epoch 18, Train loss 0.3515248740199151, Test loss 0.42496422645013043
Epoch 19, Train loss 0.34583768783919594, Test loss 0.39331954138673797
Epoch 20, Train loss 0.3387878672130454, Test loss 0.3781756559374985
Epoch 21, Train loss 0.333388367226955, Test loss 0.39043747657423566
Epoch 22, Train loss 0.32742547050817433, Test loss 0.39611930338440426
Epoch 23, Train loss 0.3222412084703888, Test loss 0.37989413776215475
Epoch 24, Train loss 0.31763139715009153, Test loss 0.3981907711287213
Epoch 25, Train loss 0.31227544124033657, Test loss 0.39708956003568735
Epoch 26, Train loss 0.3083479481973628, Test loss 0.3621746803734713
Epoch 27, Train loss 0.30367927116625854, Test loss 0.3744443989103767
Epoch 28, Train loss 0.29962832230462955, Test loss 0.3960111701184777
Epoch 29, Train loss 0.29510541759860287, Test loss 0.35465516966239663
Epoch 30, Train loss 0.2911869402029621, Test loss 0.3985324072989689
Epoch 31, Train loss 0.28701897167257157, Test loss 0.3484337536772345
Epoch 32, Train loss 0.28356763806297325, Test loss 0.3609428795849442
Epoch 33, Train loss 0.2794342605289874, Test loss 0.3620459948945197
Epoch 34, Train loss 0.275004328464839, Test loss 0.3561281811469679
Epoch 35, Train loss 0.2719160064038183, Test loss 0.3452547054951358
Epoch 36, Train loss 0.2682970358984176, Test loss 0.34479837053141016
Epoch 37, Train loss 0.2648664895913748, Test loss 0.35714260858904784
Epoch 38, Train loss 0.2606170200573991, Test loss 0.34679327848230956
Epoch 39, Train loss 0.2590377589088005, Test loss 0.36026558090167443
Epoch 40, Train loss 0.2544194388904297, Test loss 0.35656583347138326
Epoch 41, Train loss 0.2515182216633866, Test loss 0.35714597520744723
Epoch 42, Train loss 0.2493063440915745, Test loss 0.3511483248821489
Epoch 43, Train loss 0.24552003657226878, Test loss 0.3836745161349606
Epoch 44, Train loss 0.24245951761171888, Test loss 0.3389835833648967
Epoch 45, Train loss 0.24013529041174378, Test loss 0.34417330858054435
Epoch 46, Train loss 0.2366783776755399, Test loss 0.34200436427335074
Epoch 47, Train loss 0.23514254758162284, Test loss 0.4056032252539495
Epoch 48, Train loss 0.23186070791312627, Test loss 0.36982799691569274
Epoch 49, Train loss 0.2284418091114396, Test loss 0.3299709448389187
```

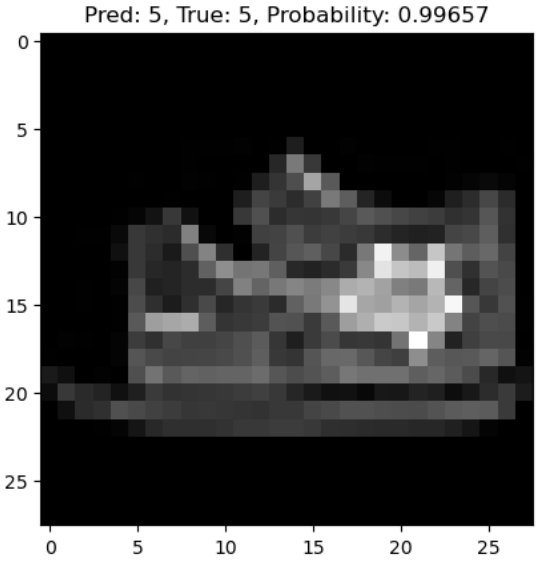
In [56]:
```
# Plot Train and Test error curves
plt.plot(train_loss_history,'-',linewidth=3,label='Train error')
plt.plot(test_loss_history,'-',linewidth=3,label='Test error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
plt.show()
```

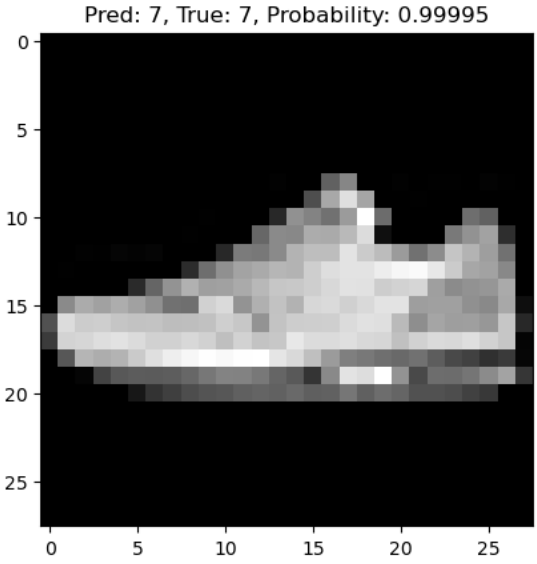


In [64]:
```
# Visualize samples and its prdicted probabilities
predicted_outputs = model(images)
predicted_classes = torch.max(predicted_outputs, 1)[1]
print('Predicted:', predicted_classes)
fit = loss(predicted_output, labels)
print('True labels:', labels)
print(fit.item())
probs = torch.nn.functional.softmax(predicted_output.cpu(), dim=1).detach().numpy()

for i in range(3):  # randomly select theree images to verify
    j = random.randint(0, images.shape[0] - 1)
    plt.imshow(images[j].squeeze().cpu(), cmap=plt.cm.gray)
    plt.title('Pred: {}, True: {}, Probability: {:.5f}'.format(predicted_classes[j].item(),labels[j].item(),p
    plt.show()
```

```
Predicted: tensor([3, 2, 7, 5, 8, 4, 5, 0, 8, 9, 1, 9, 1, 8, 1, 5])
True labels: tensor([3, 2, 7, 5, 8, 4, 5, 6, 8, 9, 1, 9, 1, 8, 1, 5])
0.12704485654830933
```



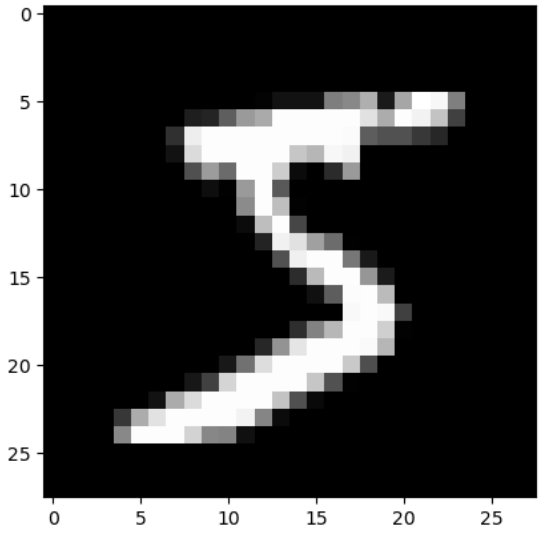



Pred: 7, True: 7, Probability: 0.99995
0
5
10
15
20
25
0
5
10
15
20
25

Question 4

The code for question 4 is presented in the following pages:

In [1]:
```
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');
```



In [2]:
```
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
```

In [3]:
```
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.
# ...

layers = [784, 32, 32, 10]
weights = []
biases = []

for i in range(3):

    n_in, n_out = layers[i], layers[i + 1]
```

```
    w = rng.normal(0, (1 / math.sqrt(n_in)), (n_in, n_out))
    weights.append(w)

    b = np.zeros((1,n_out))
    biases.append(b)
```

In [4]:
```
def feed_forward_sample(sample, label):
    """
    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.
    # ...

    activation = sample.flatten()

    for index, weight in enumerate(weights):

        z = np.matmul(activation,weight)+biases[index] # z = wx+b
        if index < len(weights)-1:
            activation = sigmoid(z)  # z' = sigmoid(z)
        else:
            activation = softmax(z)  # z' = softmax(z)

    y = integer_to_one_hot(label,10)  # True label

    loss = cross_entropy_loss(y,activation)

    prediction = np.argmax(activation)
    one_hot_prediction = integer_to_one_hot(prediction,10)

    return loss, one_hot_prediction
```

In [5]:
```
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
    # ...

    for i in range(x.shape[0]):
        sample = x[i]
        label = y[i]
        L, P = feed_forward_sample(sample, label)
        losses[i] = L
        one_hot_guesses[i] = P

    y_one_hot = np.zeros((y.size, 10))
    y_one_hot[np.arange(y.size), y] = 1

    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("")
```

In [6]:
```
feed_forward_test_data()
```

```
Feeding forward all test data...

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

In [7]:
```
def train_one_sample(sample, y, learning_rate=0.003):
    a = np.reshape(sample,(1,28*28))

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

    # Forward pass
    activations = []
    num_layers = 3
    weight_gradients = []
    bias_gradients = []
    numclasses = 10
```

```
    u1 = np.dot(a,weights[0]) + biases[0]
    z1 = sigmoid(u1)  # hidden layer1
    activations.append(z1)

    u2 = np.dot(z1,weights[1]) + biases[1]
    z2 = sigmoid(u2)  # hidden layer2
    activations.append(z2)

    u3 = np.dot(z2,weights[2]) + biases[2]
    yhat = softmax(u3)  # hidden layer3
    activations.append(yhat)

    #activation[i][j] indicate the activated result of j_th neuron in the i_th layer

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

    # Loss Calculation
    yvector = integer_to_one_hot(y,numclasses)
    loss = cross_entropy_loss(yvector,yhat)

    #Prediction
    pred = np.argmax(yhat)
    one_hot_guess = integer_to_one_hot(pred,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.
    # ...

    # Update weights & biases based on your calculated gradient

    du3 = activations[2]-yvector  # hidden layer3
    dW3 = activations[1].T.dot(du3)
    db3 = du3

    dz2 = du3.dot(weights[2].T)  # hidden layer2
    dW2 = activations[0].T.dot(np.multiply(dz2,dsigmoid(u2)))
    db2 = np.multiply(dz2,dsigmoid(u2))

    dz1 = np.multiply(dz2.dot(weights[1].T),dsigmoid(u2))  # hidden layer1
    dW1 = a.T.dot(np.multiply(dz1,dsigmoid(u1)))
    db1 = np.multiply(dz1,dsigmoid(u1))

    weight_gradients.append(dW1)
    weight_gradients.append(dW2)
    weight_gradients.append(dW3)
    bias_gradients.append(db1)
    bias_gradients.append(db2)
    bias_gradients.append(db3)

    for i in range(num_layers):
        weights[i] -= weight_gradients[i] * learning_rate
        biases[i] -= bias_gradients[i].flatten() * learning_rate
```

In [9]:

```
def train_one_epoch(x, y, learning_rate=0.0005):

    # Q4. Write the training loop over the epoch here.
    # ...
    for i in range(x.shape[0]):
        if i==0 or ((i+1)%10000 == 0):
            percentage = (i+1)/x_train.shape[0]*100
            print('Training Percentage: {:.3f}'.format(percentage))

        train_one_sample(x[i],y[i],learning_rate)
    print("")

def test_and_train():
    train_one_epoch(x_train,y_train)
    feed_forward_test_data()


for i in range(3):
    print('Epoch {} is running'.format(i+1))
    test_and_train()

print('{} epoches have been completed. '.format(i+1))
```

```
Epoch 1 is running
Training Percentage: 0.002
Training Percentage: 16.667
Training Percentage: 33.333
Training Percentage: 50.000
Training Percentage: 66.667
Training Percentage: 83.333
Training Percentage: 100.000

Feeding forward all test data...

Average loss: 0.49
Accuracy (# of correct guesses): 8686.0 / 10000 ( 86.86 %)

Epoch 2 is running
Training Percentage: 0.002
Training Percentage: 16.667
Training Percentage: 33.333
Training Percentage: 50.000
Training Percentage: 66.667
Training Percentage: 83.333
Training Percentage: 100.000

Feeding forward all test data...

Average loss: 0.45
Accuracy (# of correct guesses): 8684.0 / 10000 ( 86.84 %)

Epoch 3 is running
Training Percentage: 0.002
Training Percentage: 16.667
Training Percentage: 33.333
Training Percentage: 50.000
Training Percentage: 66.667
Training Percentage: 83.333
Training Percentage: 100.000

Feeding forward all test data...

Average loss: 0.46
Accuracy (# of correct guesses): 8627.0 / 10000 ( 86.27 %)

3 epoches have been completed.
```