**Karan Vora (kv2154)**
**ECE-GY 7143 Introduction to Deep Learning, Assignment 1**

**Problem 1):**

**Solution a):**
The vector expression for $l_1$ – norm is

$$L(w)=\|Xw-Y\|_1$$

where, X is a $n\times d$ matrix, w is $d\times 1$ vector and Y is a $n\times 1$ vector.

**Solution b):**
Unlike, $l_2$ – norm, There's no closed form expression for $l_1$ – norm as it is non differentiable. We cannot solve for the gradient being zero.

**Solution c):**
Referred the provided notes for lecture 1, topic Warmup: Linear models. In three step recipe, in the 2nd step where we measure the quality of model, losses are an important metric. The lower the loss, the lower the overall penalty for an incorrect prediction.

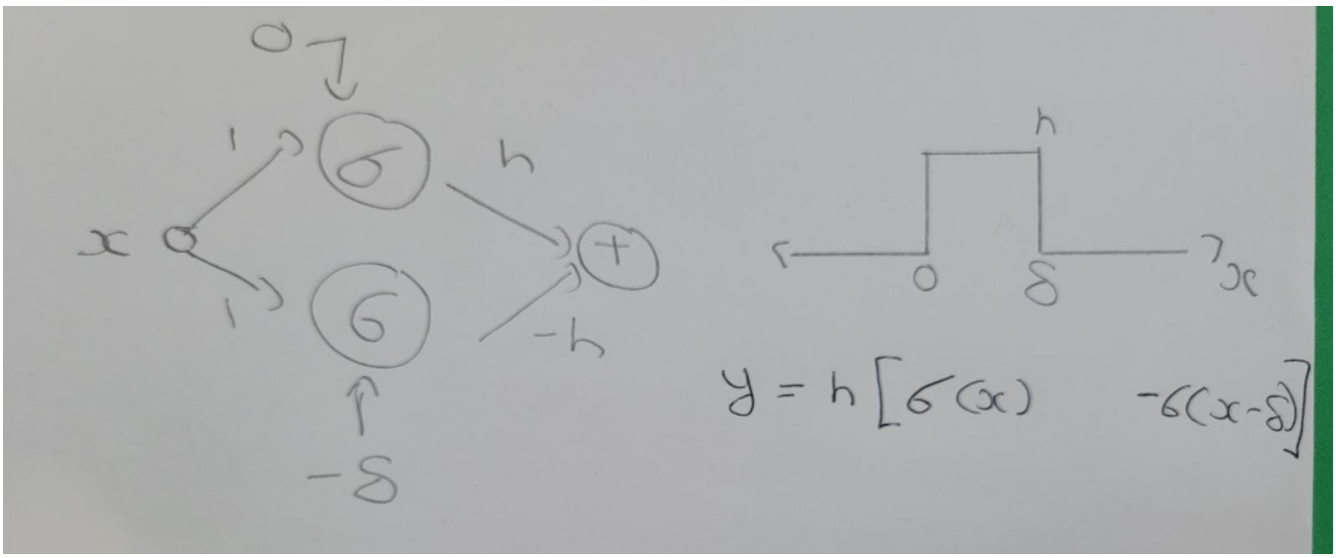
==========

**Problem 2):**

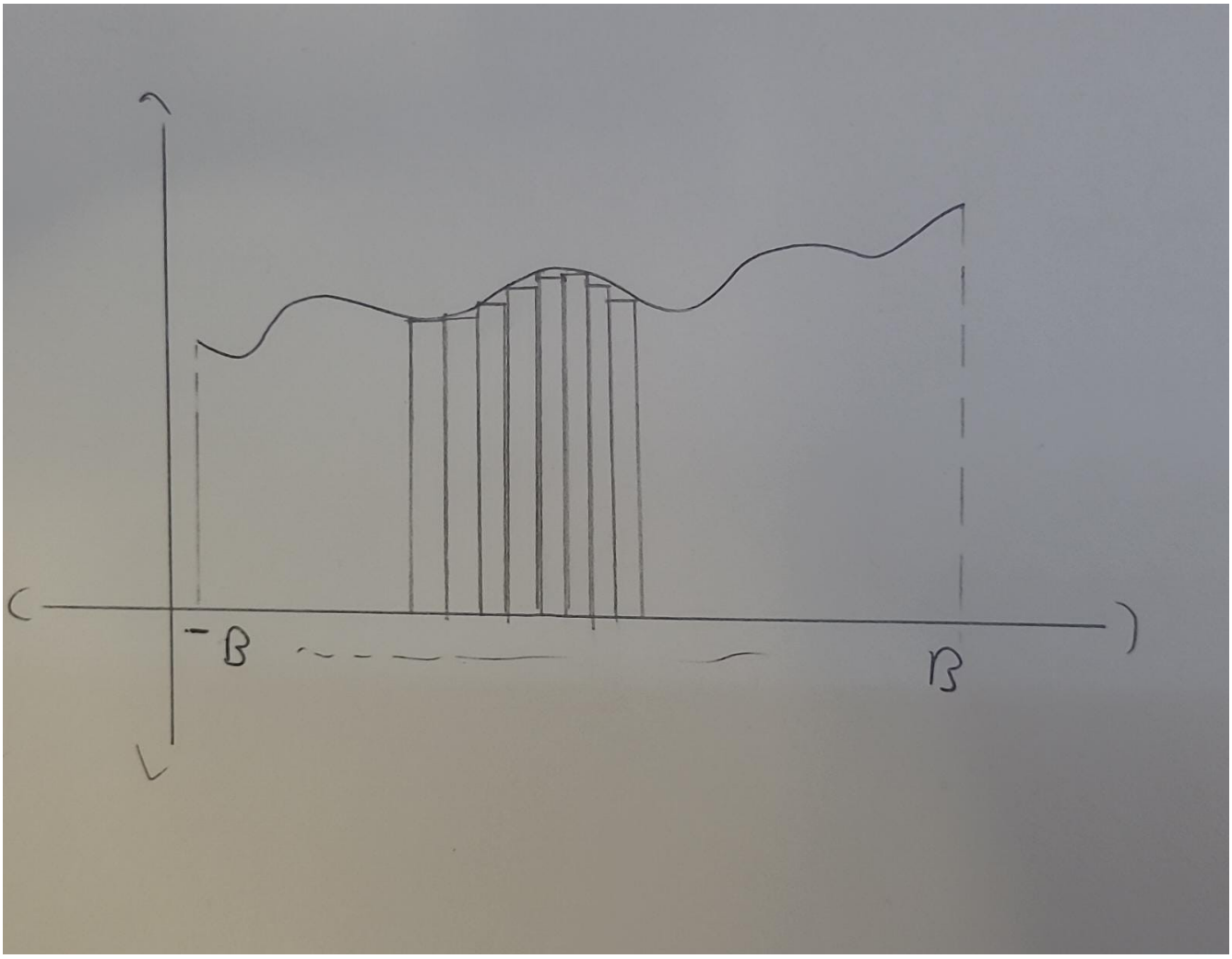
**Solution a):**
A box function can be realized by using the difference between two step function:

$$f(x)=h\,\sigma(x)-h\,\sigma(x-\delta)$$

This produces the following graph.



**Solution b):**
The approximate plot of smoothening a curve is given below. We can separate any arbitrary function distribution into an approximate superposition of box function. Each chunk can be plotted using 2 hidden neurons. To get the plot, we change the bias of first layer and value of second layer.




**Solution c):**
We can apply the same principle to a different dataset types aswell. As the dimensionality of the data increases so is the dimensions of the plotted boxes, thus we require more boxes to plot. Consider a domain,

$D=[-B,B]$

the approximate relation between dimensionality *d* and number of boxes required is given by

$N=(\frac{B}{\delta})^{d}$

With increase in *d* the number of trainable parameters also explodes requiring a more complex network. The network architecture will always be a combination of depth and width to sufficiently

approximate the said function.

**Solution d):**
From the above mentioned discussion, we can conclude that any sufficiently wide and deep neural network can approximate any arbitrary function with a great degree of accuracy considering the data provided as input is sufficiently good enough in terms of quality and the other parameters or an NN architecture are also optimal.

==========

**Problem 3):**

The softmax function is given by,

$$y_i = \frac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}}$$

Taking log,

$$\log(y_i) = z_i - \log\left(\sum_{k=1}^{n} e^{z_k}\right)$$

Now taking partial derivative,

$$\frac{\partial \log(y_i)}{\partial z_j} = \frac{1}{y_i}\frac{\partial y_i}{\partial z_j}$$

When j = i,

$$\frac{\partial y_i}{\partial z_i} = 1 - \frac{\partial}{\partial z_i}\log\left(\sum_{k=1}^{n} e^{z_k}\right)$$

$$= 1 - \frac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}}$$

$$= 1 - y_i$$

and when j ≠ i,

$$\frac{\partial y_i}{\partial z_j} = -\frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}} = -y_j$$

Therefore, putting it together using Dirac Delta as indicator

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

==========

Karan Vora (kv2154)

ECE-GY 7143 Introduction to Deep Learning, Assignment 1, Question 4

Disclosure:

I have discussed this particular problem with Rithviik Srinivasan (rs8385) and Charmee Mehta (cm6389)

I have also used online resources like StackOverflow, GitHub repos, Kaggle Competition entries, ChatGPT and official pytorch, numpy, matplotlib documentation.

In [ ]:
```
#Import all required libraries
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

In [ ]:
```
# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [ ]:
```
# Load FashionMNIST dataset
TrainDataset = torchvision.datasets.FashionMNIST(root='./FashionMNIST/', train=True, transform=torchvision.transforms.ToTensor(), download=True)
TestDataset = torchvision.datasets.FashionMNIST(root='./FashionMNIST/', train=False, transform=torchvision.transforms.ToTensor(), download=True)

# Define dataloaders
TrainDataLoader = torch.utils.data.DataLoader(TrainDataset, batch_size=64, shuffle=True)
TestDataLoader = torch.utils.data.DataLoader(TestDataset, batch_size=64, shuffle=False)
```

In [ ]:
```
# Define model
# As input dimension is 28 x 28, the input layer will have 784 nodes
# The architecture is a 3 layer dense NN with 256, 128, 64 neurons in the hidden layers
# The output is a 10 class classification thus 10 nodes in output.
# We are using ReLU activation for each layer
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = x.view(-1, 784)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.fc4(x)
        return x

model = NeuralNetwork().to(device)
```

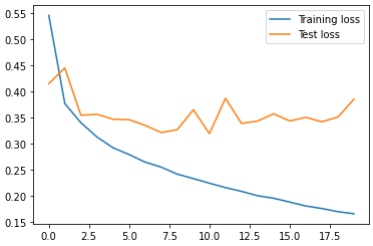
In [ ]:
```
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [ ]:
```
# Train model
train_losses = []
test_losses = []
num_epochs = 20
```

In [ ]:
```
# Running the epochs and training the network, Calculating the loss.

for epoch in range(num_epochs):
    train_loss = 0.0
    test_loss = 0.0
    for images, labels in TrainDataLoader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        output = model(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * images.size(0)
    train_loss = train_loss / len(TrainDataLoader.dataset)
    train_losses.append(train_loss)

    with torch.no_grad():
        for images, labels in TestDataLoader:
            images, labels = images.to(device), labels.to(device)
            output = model(images)
            loss = criterion(output, labels)
            test_loss += loss.item() * images.size(0)
        test_loss = test_loss / len(TestDataLoader.dataset)
        test_losses.append(test_loss)

    print('Epoch: {} \tTraining Loss: {:.6f} \tTest Loss: {:.6f}'.format(epoch+1, train_loss, test_loss))
```

```
Epoch: 1        Training Loss: 0.546322         Test Loss: 0.415558
Epoch: 2        Training Loss: 0.377236         Test Loss: 0.445570
Epoch: 3        Training Loss: 0.340729         Test Loss: 0.355067
Epoch: 4        Training Loss: 0.313457         Test Loss: 0.356937
Epoch: 5        Training Loss: 0.292774         Test Loss: 0.347378
Epoch: 6        Training Loss: 0.279877         Test Loss: 0.346601
Epoch: 7        Training Loss: 0.265311         Test Loss: 0.335643
Epoch: 8        Training Loss: 0.255733         Test Loss: 0.321906
Epoch: 9        Training Loss: 0.242337         Test Loss: 0.327425
Epoch: 10       Training Loss: 0.233854         Test Loss: 0.365484
Epoch: 11       Training Loss: 0.224996         Test Loss: 0.319899
Epoch: 12       Training Loss: 0.216542         Test Loss: 0.387408
Epoch: 13       Training Loss: 0.209370         Test Loss: 0.339271
Epoch: 14       Training Loss: 0.201021         Test Loss: 0.343956
Epoch: 15       Training Loss: 0.196199         Test Loss: 0.357778
Epoch: 16       Training Loss: 0.188847         Test Loss: 0.344070
Epoch: 17       Training Loss: 0.181367         Test Loss: 0.351023
Epoch: 18       Training Loss: 0.176534         Test Loss: 0.342519
Epoch: 19       Training Loss: 0.170533         Test Loss: 0.351566
Epoch: 20       Training Loss: 0.166449         Test Loss: 0.385731
```

In [ ]:
```
# Plot loss curves
plt.plot(train_losses, label='Training loss')
plt.plot(test_losses, label='Test loss')
plt.legend()
plt.show()
```




In [ ]:
```
# Evaluate model
correct = 0
total = 0
with torch.no_grad():
    model.eval()
    for images, labels in TestDataLoader:
        images, labels = images.to(device), labels.to(device)
        output = model(images)
        _, predicted = torch.max(output.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    print('Test Accuracy: {:.2f}%'.format(correct / total * 100))
    model.train()
```

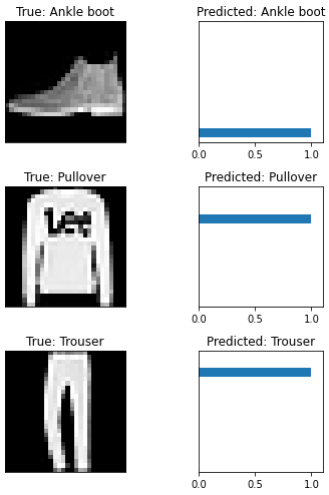
```
Test Accuracy: 89.28%
```

In [ ]:
```
# Define class names
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

In [ ]:
```
# Choose 3 random images from the test dataset
num_images = 3
images, labels = next(iter(TestDataLoader))
images, labels = images[:num_images], labels[:num_images]
images, labels = images.to(device), labels.to(device)
```

In [ ]:
```
# Make predictions and get predicted class probabilities
model.eval()
outputs = model(images)
probs = torch.softmax(outputs, dim=1)
predicted_classes = torch.argmax(probs, dim=1)
```

In [ ]:
```
# Visualize images and predicted class probabilities
with torch.no_grad():
  fig, axs = plt.subplots(nrows=num_images, ncols=2, figsize=(6, 7))
  for i in range(num_images):
      img = np.transpose(images[i].cpu().numpy(), (1, 2, 0)).squeeze()
      axs[i, 0].imshow(img, cmap='gray')
      axs[i, 0].set_xticks([])
      axs[i, 0].set_yticks([])
      axs[i, 0].set_title('True: {}'.format(class_names[labels[i]]))
      axs[i, 1].barh(np.arange(len(class_names)), probs[i].cpu().numpy())
      axs[i, 1].set_aspect(0.1)
      axs[i, 1].set_yticks([])
      axs[i, 1].set_xlim([0, 1.1])
      axs[i, 1].invert_yaxis()
      axs[i, 1].set_title('Predicted: {}'.format(class_names[predicted_classes[i]]))
  plt.tight_layout()
  plt.show()
```



Karan Vora (kv2154)

ECE-GY 7143 Introduction to Deep Learning, Assignment 1, Question 5

Disclosure:

I have discussed this particular problem with Rithviik Srinivasan (rs8385), Ratik Vig (rv2292) and Charmee Mehta (cm6389)

I have also used online resources like StackOverflow, GitHub repos, Kaggle Competition entries, ChatGPT, YouTube video on back-propagation by prof. Chinmay Hegde and official pytorch, numpy, matplotlib documentation.

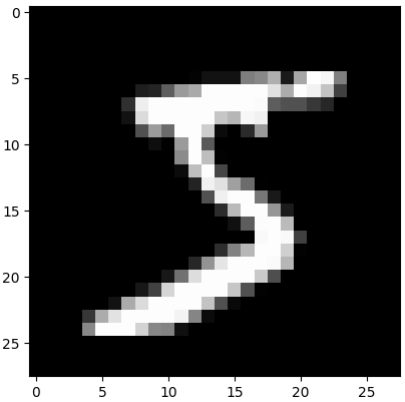
In this problem we will train a neural network from scratch using numpy. In practice, you will never need to do this (you'd just use TensorFlow or PyTorch). But hopefully this will give us a sense of what's happening under the hood.

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.

In [ ]:
```
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')
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7f0ef26f0640>
```




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.

In [ ]:
```
import numpy as np
from math import exp

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 $1/\max(n_{in}, n_{out})$.

In [ ]:
```
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 =
# biases =
InputLayerSize = 784
FirstHiddenLayer = 32
SecondHiddenLayer = 32
OutputLayer = 10

Weight1 = np.random.randn(InputLayerSize, FirstHiddenLayer) * np.sqrt(1 / max(InputLayerSize, FirstHiddenLayer))
Bias1 = np.zeros((1, FirstHiddenLayer))
Weight2 = np.random.randn(FirstHiddenLayer, SecondHiddenLayer) * np.sqrt(1 / max(FirstHiddenLayer, SecondHiddenLayer))
Bias2 = np.zeros((1, SecondHiddenLayer))
Weight3 = np.random.rand(SecondHiddenLayer, OutputLayer) * np.sqrt(1 / max(SecondHiddenLayer, OutputLayer))
Bias3 = np.zeros((1, OutputLayer))

# max_n_m = max(InputLayerSize, FirstHiddenLayer)
# Weight1 = rng.normal(loc=0, scale=1/math.sqrt(max_n_m), size=(InputLayerSize, FirstHiddenLayer))
# Bias1 = np.zeros(FirstHiddenLayer)

# max_n_m = max(FirstHiddenLayer, SecondHiddenLayer)
# Weight2 = rng.normal(loc=0, scale=1/math.sqrt(max_n_m), size=(FirstHiddenLayer, SecondHiddenLayer))
# Bias2 = np.zeros(SecondHiddenLayer)

# max_n_m = max(SecondHiddenLayer, OutputLayer)
# Weight3 = rng.normal(loc=0, scale=1/math.sqrt(max_n_m), size=(SecondHiddenLayer, OutputLayer))
# Bias3 = np.zeros(OutputLayer)

weights = [Weight1, Weight2, Weight3]
biases = [Bias1, Bias2, Bias3]
```

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).

In [ ]:

```
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.
  # ...

  sample = [sample]

  for i in range(len(weights)):
    z = np.dot(sample[-1].flatten().T, weights[i])
    z = z + biases[i]
    if(i < len(weights) - 1):
      a = sigmoid(z)
      sample.append(a)
    else:
      a = softmax(z)
      sample.append(a)

  # Compute cross entropy loss
  y_one_hot = np.zeros((1, 10))
  y_one_hot[0, y] = 1

  output = sample[-1]
  loss = cross_entropy_loss(output, y_one_hot)

  # Compute most likely class
  one_hot_guess = np.zeros((1, 10))
  one_hot_guess[0, np.argmax(output)] = 1

  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
  # ...

  for i in range(x.shape[0]):
    sample = x[i, :]
    label = y[i]

    loss, one_hot_guess = feed_forward_sample(sample, label)

    losses[i] = loss
    one_hot_guesses[i, :] = one_hot_guess


  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("")

feed_forward_test_data()
```

```
Feeding forward all test data...
/tmp/ipykernel_9962/1909522419.py:26: RuntimeWarning: divide by zero encountered in log
  return -np.sum(y * np.log(yHat))
Average loss: inf
Accuracy (# of correct guesses): 1025.0 / 10000 ( 10.25 %)
```

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.

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

#   # We will store each layer's activations to calculate gradient
#   activations = []
#   #activations = np.array(a)

#   # Forward pass
#   for i in range(len(weights)):
#     z = np.dot(a, weights[i])
#     z = z + biases[i]
#     if(i < len(weights) - 1):
#       a = sigmoid(z)
#       activations.append(a)
#     else:
#       a = softmax(z)
#       activations.append(a)

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

#   # Backward pass
#   delta = -(y - activations[-1]) * dsigmoid(activations[-1])
#   #print(delta.shape)
#   weight_gradients = []
#   bias_gradients = []
#   for i in reversed(range(len(weights))):
#     # print("Shape of activation: {}".format(activations[i].reshape(-1, 1).shape))
#     # print("Shape of Delta: {}".format(delta.reshape(1, -1).shape))
#     weight_gradients.insert(0, np.outer(delta, activations[i]))
#     bias_gradients.insert(0, delta)
#     if i > 0:
#       delta = (np.dot(weights[i], delta.T)) * (dsigmoid(activations[i]))

#   # 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

#   for i in range(len(weights)):
#     weights[i] -= weight_gradients[i] * learning_rate
#     biases[i] -= bias_gradients[i] * learning_rate
```

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

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

#     # Forward pass
#     for i in range(len(weights)):
#         z = np.dot(activations[-1], weights[i]) + biases[i]
#         if i == len(weights) - 1:
#             a = softmax(z)
#         else:
#             a = sigmoid(z)
#         activations.append(a)

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

#     # Backward pass
#     weight_gradients = []
#     bias_gradients = []
#     for i in reversed(range(len(weights))):
#         if i == len(weights) - 1:
#             delta *= dsigmoid(activations[i])
#         else:
#             delta = np.dot(delta, weights[i+1].T) * dsigmoid(activations[i])
#         weight_gradients.insert(0, np.dot(activations[i].reshape(-1, 1), delta))
#         bias_gradients.insert(0, delta)

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

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

    # We will store each layer's activations to calculate gradient
    activations = []
    #activations = np.array(a)

    # Forward pass
    FirstLayer = np.dot(a, weights[0]) + biases[0]
    FirstLayerActivation = sigmoid(FirstLayer)
    SecondLayer = np.dot(FirstLayerActivation, weights[1]) + biases[1]
    SecondLayerActivation = sigmoid(SecondLayer)
```

```
    ThirdLayer = np.dot(SecondLayerActivation, weights[2]) + biases[2]
    ThirdLayerActivation = softmax(ThirdLayer)

    # Q3. This should be the same as what you did in feed_forward_sample above.
    # ...
    prediction = np.argmax(ThirdLayerActivation)
    one_hot_guess = integer_to_one_hot(prediction,10)
    loss = cross_entropy_loss(integer_to_one_hot(y, 10), ThirdLayerActivation)

    # 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.
    # ...

    LastLayer = ThirdLayerActivation - integer_to_one_hot(y,10)
    LastLayerWeights = SecondLayerActivation.T.dot(LastLayer)
    LastLayerBiases = LastLayer

    SecondLastLayer = LastLayer.dot(weights[2].T)
    SecondLastLayerUpdate = np.multiply(SecondLastLayer, dsigmoid(SecondLayer))
    SecondLastLayerWeights = FirstLayerActivation.T.dot(SecondLastLayerUpdate)
    SecondLayerBiases = SecondLastLayer

    FirstLastLayer = SecondLastLayer.dot(weights[1].T)
    FirstLayerUpdate = np.multiply(FirstLastLayer, dsigmoid(FirstLayer))
    FirstLayerWeights = a.T.dot(FirstLayerUpdate)
    FirstLayerBiases = FirstLastLayer

    weight_gradients = [FirstLayerWeights, SecondLastLayerWeights, LastLayerWeights]
    bias_gradients = [FirstLayerBiases, SecondLayerBiases, LastLayerBiases]

    for i in range(3):
        weights[i] = weights[i] - learning_rate * weight_gradients[i]
        biases[i] = biases[i] - learning_rate * bias_gradients[i]

    return loss
```

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

Q4. Train your model for 3 epochs.

In [ ]:
```
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 values in range(x_train.shape[0]):
    train_one_sample(x_train[values], y_train[values], learning_rate)
  print("Finished training.\n")


feed_forward_test_data()

def test_and_train():
  train_one_epoch()
  feed_forward_test_data()

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

```
Feeding forward all test data...
/tmp/ipykernel_9962/1909522419.py:26: RuntimeWarning: divide by zero encountered in log
  return -np.sum(y * np.log(yHat))
Average loss: inf
Accuracy (# of correct guesses): 1025.0 / 10000 ( 10.25 %)

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

Feeding forward all test data...

Average loss: inf
Accuracy (# of correct guesses): 4175.0 / 10000 ( 41.75 %)

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

Feeding forward all test data...

Average loss: inf
Accuracy (# of correct guesses): 4934.0 / 10000 ( 49.34 %)

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

Feeding forward all test data...

Average loss: inf
Accuracy (# of correct guesses): 5664.0 / 10000 ( 56.64 %)
```

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.