Assignment 1

Jagennath Hari

NYUID: jh7454

1 Linear regression with non-standard losses

1.1 Loss function in terms of the L1-norm

The given input is a matrix X.

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \text{ X} \in R^{n \times d}$$

The given output is a matrix y.

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \text{ y} \in R^{n \times 1}$$

L1-norm Loss is given as :-
$Loss(L1) = \sum_{i=1}^{n} ((X_i W_i + b_i) - y_i)$,
where X_i is the input vector, W_i , b_i is the corresponding weight and bias vector and y_i is the label.
This is also commonly written as $\mathbf{L1_{norm}} = \Big|\big|(\mathbf{Xw} + \mathbf{b}) - \mathbf{y}\big|\Big|_{\mathbf{1}}$
Where $\mathbf{X} \in \mathbf{R^{n \times d}}$, $\mathbf{y} \in \mathbf{R^{n \times 1}}$, $\mathbf{w} \in \mathbf{R^{d \times 1}}$ and $\mathbf{b} \in \mathbf{R^{n \times 1}}$

1.2 Optimal linear model in closed form

L1-norm is not differentiable, so a solution for gradient equal to 0 does not exist.

$$\nabla(L1) = \begin{cases} \text{-1} & \text{if } L1 < 0 \\ \infty & \text{if } L1 = 0 \\ 1 & \text{if } L1 > 0 \end{cases} \tag{1}$$

So, there is no closed form solution for L1-norm.

2 Expressivity of neural networks

2.1 Simple neural network realizing box function

The box function can expressed as shown below :-

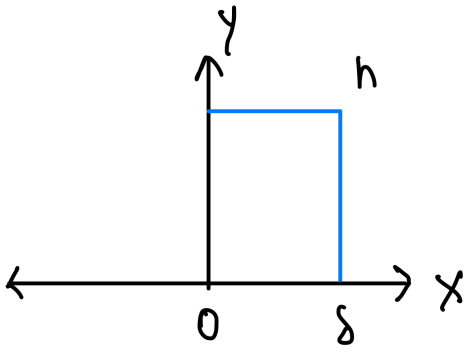


Figure 1:- Box function

A box function can be realized by a pair of unit step function :-
$y = W_3\sigma_1\left(W_1x + b_1\right) + W_4\sigma_2\left(W_2x + b_2\right)$
This can be represented as :-

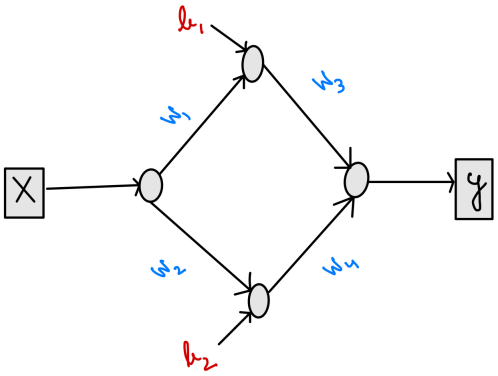


Figure 2:- Neural network with two hidden neurons

One such weights which can realize this function is given below :-

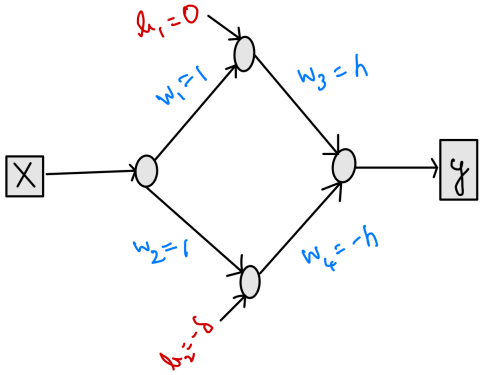


Figure 3:- Neural network with weights and biases

We can then write y in terms of h and δ as:-
$\mathbf{y = h\sigma_1\left(x\right) - h\sigma_2\left(x - \delta\right)}$

2.2 Approximating function using hidden layer of neurons

We can represent the function by subdividing it into multiple box functions. By adjusting the the weights and biases of each neuron, will cause each of boxes to change in height and have an offset. By carefully controlling these weights and biases the neural network can be represented in the interval $\mathbf{f}(\mathbf{x}) \in [-\mathbf{B}, \mathbf{B}]$.

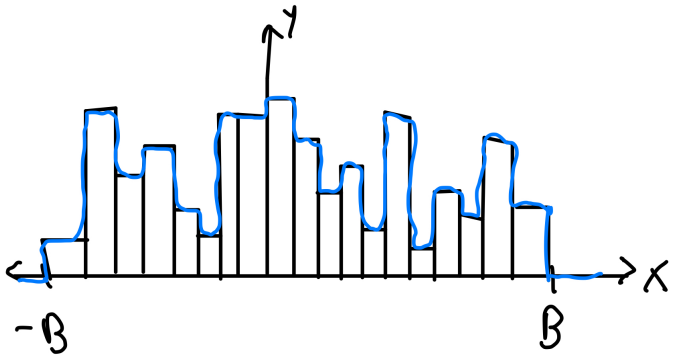


Figure 4:- Recreating the function

This function will have $\mathbf{O}(\frac{\mathbf{B}}{\delta})$ neurons.

2.3 Extended to the case of d-dimensional inputs

Yes, a d-dimensional input can be used to express the function. As the dimension of inputs increase so do the number of weights and biases. The neural network needs to learn a lot of weights and biases if d increase, this causes an overall speed reduction or slower time complexity. When there are d dimensions and the number of neurons are $\mathbf{O}((\frac{\mathbf{B}}{\delta})^{\mathbf{d}})$.

3 Calculating gradients

We know $y = softmax(z)$, $y \in R^{n \times m}$, $z \in R^{n \times m}$

$y_i = \frac{e^{z_i}}{\sum_{i=1}^{k} e^{z_j}}$

Computing $\delta y_i / \delta z_j$
Let us assume for case i=j=1

$\frac{\delta y_1}{\delta z_1} = \frac{\delta y_1}{\delta y_1} . \frac{\delta y_1}{\delta z_1}$

$\frac{\delta y_1}{\delta z_1} = y_1 \times (1 - y_1)$

Let us assume for case i=1, j=2

$\frac{\delta y_1}{\delta z_2} = \frac{\delta y_1}{\delta y_1} . \frac{\delta y_1}{\delta z_2}$

$\frac{\delta y_1}{\delta z_1} = -y_1 \times y_2$

Let us assume for case i=2, j=1

$\frac{\delta y_2}{\delta z_1} = \frac{\delta y_2}{\delta y_2} . \frac{\delta y_2}{\delta z_1}$

$\frac{\delta y_2}{\delta z_1} = -y_2 \times y_1$

Let us assume for case i=j=2

$\frac{\delta y_2}{\delta z_2} = \frac{\delta y_2}{\delta y_2} . \frac{\delta y_2}{\delta z_2}$

$\frac{\delta y_2}{\delta z_2} = y_2 \times (1 - y_2)$

If we compute for all cases until i=j=n we get a general solution in the form of :-

$$\frac{\delta y_i}{\delta z_j} = \begin{cases} y_i \times (1 - y_j) & \text{if i = j} \\ -y_i \times y_j & \text{else} \end{cases} \tag{2}$$

$J_{ij} = \delta y_i / \delta z_j = y_i \times (\delta_{ij} - y_j)$
J is the Jacobian matrix and δ is the Dirac delta, $\delta = 1$ if $i = j$ else 0.

4 Improving the FashionMNIST classifier

Import statements

```
#Imports of Libraries

import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torchvision
%matplotlib inline
```

Figure 5:- Import statements

Setting device(I used the M1 GPU of my mac book)

```
#Setting devices for training

if torch.backends.mps.is_built and torch.backends.mps.is_available and torch.has_mps:
    print("Device set as MPS")
    device = "mps:0"
elif torch.has_cuda:
    print("Device set as CUDA")
    device = "cuda:0"
else:
    print("Using CPU instead of GPU")
    device = "cpu"
```

```
Device set as MPS
```

Figure 6:- Setting device

Loading data into batches

```
#Loading FashionMNIST Dataset

traingData = 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())
```

Figure 7:- Getting data

```
#Loading the data

trainDataLoader = DataLoader(traingData, batch_size = 64, shuffle = True, num_workers = 0)
testDataLoader = DataLoader(testData, batch_size = 64, shuffle = False, num_workers = 0)
```

Figure 8:- Loading into batches of 64

```python
#Classifier Model
####USING FLOAT32 AS M1 GPU DOES NOT SUPPORT FLOAT64##########

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 256, dtype = torch.float32)
        self.fc2 = nn.Linear(256, 128, dtype = torch.float32)
        self.fc3 = nn.Linear(128, 64, dtype = torch.float32)
        self.fc4 = nn.Linear(64, 10, dtype = torch.float32)
    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        return self.fc4(x)
```

Figure 9:- Model architecture

Sending model to GPU, using stochastic gradient descent as the optimizer and cross entropy loss.

```python
#Creating model object and setting loss function and optimizers

net = Net().to(device)
print("Model Architecture:-")
print(net)
Loss = nn.CrossEntropyLoss()
print("Loss is:-", Loss)
optimizer = torch.optim.SGD(net.parameters(), lr = 1e-01)
#optimizer = torch.optim.Adam(net.parameters(), lr= 1e-03)
print("Optimizer is:-", optimizer)
```

```
Model Architecture:-
Net(
  (fc1): Linear(in_features=784, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=64, bias=True)
  (fc4): Linear(in_features=64, out_features=10, bias=True)
)
Loss is:- CrossEntropyLoss()
Optimizer is:- SGD (
Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    lr: 0.1
    maximize: False
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

Figure 10:- Optimizer and loss

Training code is given in the next page, where we will see how the loss changes, when trained for 100 epochs.

```
#Training and Testing

train_loss_history = []
test_loss_history = []

for epoch in range(100):
  train_loss = 0.0
  test_loss = 0.0
  for i, data in enumerate(trainDataLoader):
    net.train()
    images, labels = data
    images = images.to(device)
    labels = labels.to(device)
    optimizer.zero_grad()
    predicted_output = net(images)
    fit = Loss(predicted_output,labels)
    fit.backward()
    optimizer.step()
    train_loss += fit.item()
  for i, data in enumerate(testDataLoader):
    with torch.no_grad():
      net.eval()
      images, labels = data
      images = images.to(device)
      labels = labels.to(device)
      predicted_output = net(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.append(train_loss)
  test_loss_history.append(test_loss)
  print('Epoch %s, Train loss %s, Test loss %s'%(epoch, train_loss, test_loss))
```

Figure 11:- Training

The loss curves are :-

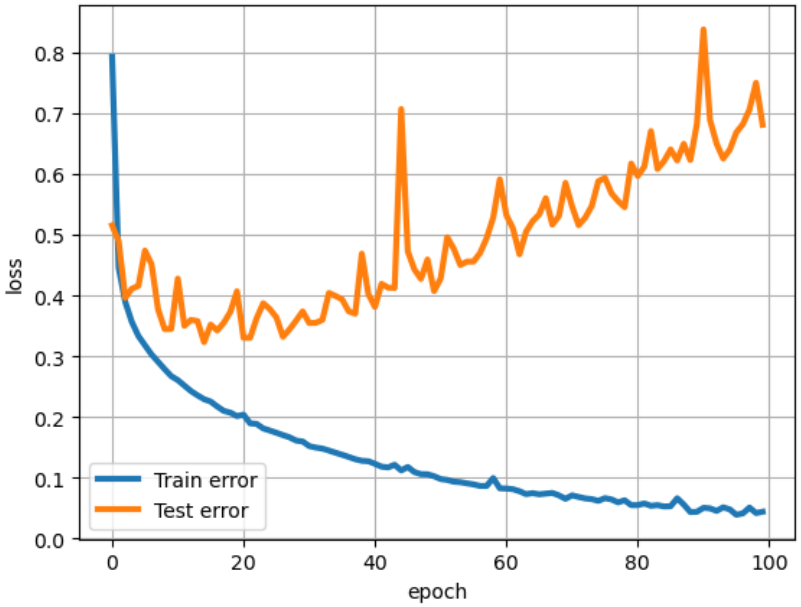


Figure 12:- Loss curves

As time increases train error decreases but after t=30 test error starts to increase.

Prediction vs truth :-

```
#Prdicted class vs truth

torch.no_grad()
net.eval()
predicted_output = net(images)
print(torch.max(predicted_output.cpu(), 1))
fit = Loss(predicted_output.cpu(), labels.cpu())
print(labels.cpu())
print(fit)
```

```
torch.return_types.max(
values=tensor([25.9558, 27.8129, 37.4549, 89.7835, 38.1162, 35.5386, 90.9389, 23.0583,
        78.3789, 54.6357, 50.7011, 49.9981, 51.5491, 39.9888, 51.7689, 33.3923],
       grad_fn=<MaxBackward0>),
indices=tensor([3, 2, 7, 5, 8, 4, 5, 0, 8, 9, 1, 9, 1, 8, 1, 5]))
tensor([3, 2, 7, 5, 8, 4, 5, 6, 8, 9, 1, 9, 1, 8, 1, 5])
tensor(0.2525, grad_fn=<NllLossBackward0>)
```

Figure 13:- Checking predictions using test set

Looks like it got one incorrect, given below are the images.

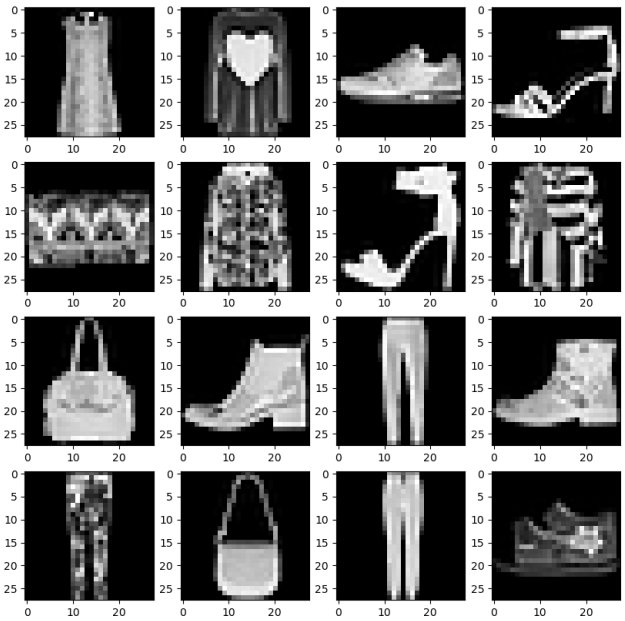


Figure 14:- Visualizing images

Now lets get the final accuracy

```
# Calculating Accuracy

correct = 0
total = 0

with torch.no_grad():
     for data in testDataLoader:
            net.eval()
            X, y = data
            X = X.to(device)
            y = y.to(device)
            output = net(X)
            for idx, i in enumerate(output):
                if torch.argmax(i) == y[idx]:
                    correct += 1
                total +=1

print("Accuracy: ", 100 * round(correct/total, 5))
```

```
Accuracy:  89.21
```

Figure 15:- Final accuracy

Final accuracy is 89.21%

5 Implementing back-propagation in Python from scratch

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.

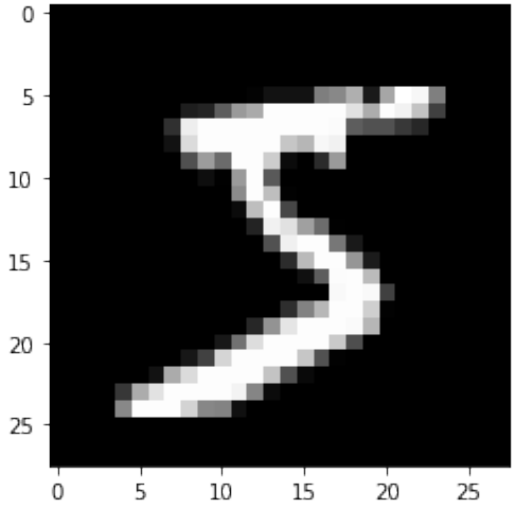
In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations 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="mnis

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datas
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```



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

```
from numpy.ma.core import sqrt
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 = [np.random.normal(0, 1/sqrt(784), (784, 32)),
           np.random.normal(0, 1/sqrt(32), (32, 32)),
           np.random.normal(0, 1/sqrt(32), (32, 10))]

biases = [np.random.normal(0, 1, (1, 32)), np.random.normal(0, 1, (1, 32)), np.ranc
```

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).

```
from re import X
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.
  a = np.reshape(sample, (1, 784))
  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)

  pred_class = np.argmax(yhat)
  one_hot_guess = integer_to_one_hot(pred_class, 10)
  yvector = integer_to_one_hot(y, 10)
  loss = cross_entropy_loss(yvector, yhat)
  # ...
  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 = np.reshape(x[i], (1, 784))
    losses[i], one_hot_guesses[i] = feed_forward_sample(sample, y[i])
  # ...

  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], "(",

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: 3.03
    Accuracy (# of correct guesses): 958.0 / 10000 ( 9.58 %)
```

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 = 0.001):
  a = np.reshape(sample, (1, 784))

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

  # Forward pass

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

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

# 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 sh
# At the end, you should get two vectors: weight_gradients and bias_gradients.
# ...


#Jacobian matrix

#Cross entropy diffrentiation is yhat - yvector
du3 = (yhat - yvector)
dw3 = z2.T.dot(du3)
db3 = du3

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

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] = 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.

```
def train_one_epoch(learning_rate = 0.0008):
  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], 0.0008)

  print("Finished training.\n")



def test_and_train():
  train_one_epoch()
  feed_forward_test_data()

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

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

Feeding forward all test data...

Average loss: 0.79
Accuracy (# of correct guesses): 8260.0 / 10000 ( 82.60 %)

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

Feeding forward all test data...

Average loss: 0.56
Accuracy (# of correct guesses): 8507.0 / 10000 ( 85.07 %)

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

Feeding forward all test data...

Average loss: 0.53
Accuracy (# of correct guesses): 8510.0 / 10000 ( 85.10 %)
```

Double-click (or enter) to edit