Humans can easily recognize digits in handwriting. However, it is not an easy task for computers as handwriting varies from person to person. Handwritten digit recognition is the ability of the machine to train itself so that it can recognize human handwritten digits.
Our goal in this project is to build and train two systems for digit classification, which take images as input and return the correct numeric digit.
As the first step, we import the libraries and load the dataset. The first part of our code shows how to import the .mat data from google drive to Google Colab. We normalize input data by dividing each pixel value by 255.0; this places the pixel value within the range 0 and 1.

Problem 1

Sample digits.
Using the uploaded data, Figure 1 displays 10 images of each of the digits.

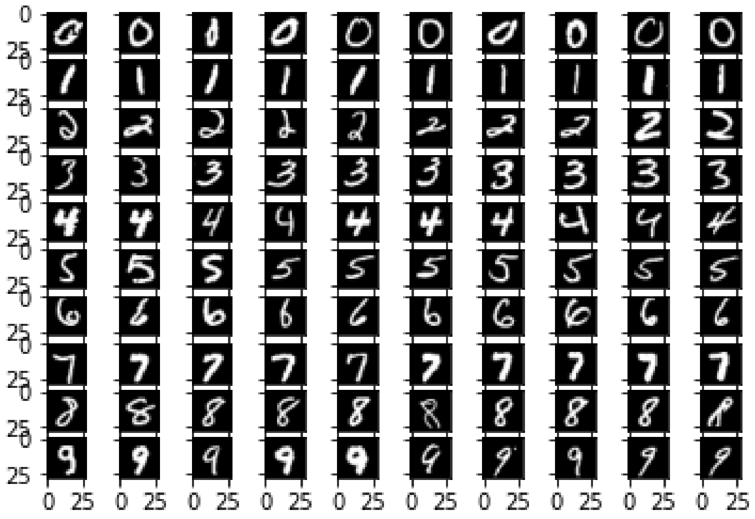


Figure 1: 10 images of each of the digits. Generated using the function `plt.subplots()`.

Part 1 code

```python
from pylab import *
from numpy import *
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import time
import matplotlib.image as mpimg
from scipy.ndimage import filters
import urllib
from numpy import random
import os
from scipy.io import loadmat

from google.colab import drive
drive.mount('/content/drive')

# Load the MNIST digit data
M=loadmat('/content/drive/My Drive/MAT5314C/mnist_all.mat')
```

```
#M=loadmat('/content/mnist_all.mat')
# Define the train Set List, which will be iterated over
trainSet = ["train0", "train1", "train2", "train3", "train4", "train5", "train6", "train7", "
    train8", "train9"]

# Divide M by 255.0
for i in trainSet:
  M[i]=M[i]/255.0

#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# Prepare subplots (10 by 10)
f, axarr = plt.subplots(10,10, sharey=True, sharex=True)

#Display the 140th to 150-th  of each digit from the training set
for i in range(len(trainSet)):
  for j in range(10):
    axarr[i,j].imshow(M[trainSet[i]][j+140].reshape((28,28)), cmap=cm.gray)
plt.show()
```

Problem 2

Initial exploration with defined functions.
We create a feed-forward network with 784 inputs ($= 28 \times 28$), and a Softmax function as the neural transfer operation in the last layer to generate the output.
The forward passes on a single example x and executes the following computation; in our code we compute this in two stages, `id_layer()` and then `Softmax()`:

$$L0 = \text{Softmax}(W0^T x + b0)$$

The Softmax function takes a vector of numbers as an input and turns these numbers into a vector that represents the probability distributions of a list of potential outcomes, each of the hidden layer units in the last layer outputs a number between 0 and 1. The entries of the output vector add up to one. The function is defined as:

$$a(y) = \frac{e^{y_i}}{\sum\limits_{i=1}^{k} e^{y_i}}, \qquad \text{where } y = \{y_1, \ldots, y_k\}, \quad k \text{ is the number of classes.}$$

The above Softmax function is not stable in implementation; in python, users will frequently get *nan* error due to floating-point limitation. To avoid this, we multiplied both the numerator and denominator by a small constant c and chose -max(y) as log(c). The modified equation is given by:

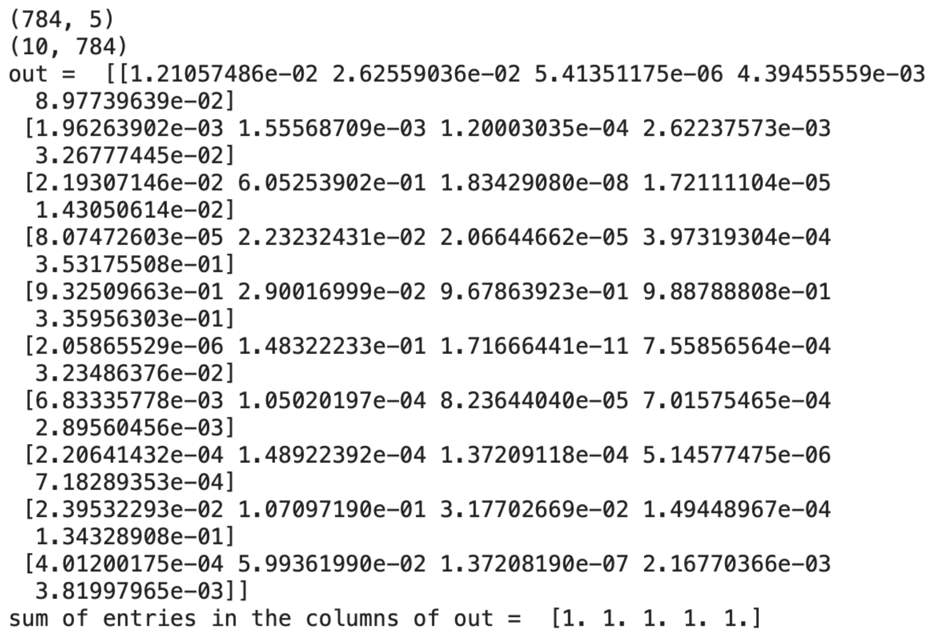
$$a(y) = \frac{e^{y_i + \log c}}{\sum\limits_{i=1}^{k} e^{y_i + \log c}} = \frac{e^{y_i - \max(y)}}{\sum\limits_{i=1}^{k} e^{y_i - \max(y)}}$$

In line 45 of the code, we examine the specified functions using generated data (`random.rand()`). In this testing, 5 examples are considered (`x = M["train4"][0:5]`). Figure 2 indicates the vectors of probabilities, and the sum of entries in the column is 1.

Part 2 code

```
   from pylab import *
   from numpy import *
   from numpy import random

 5 #import pickle

   def softmax(y):
       '''Return the output of the softmax function for the matrix of output y. y
       is an NxM matrix where N is the number of outputs for a single case, and M
10     is the number of cases'''
       y = y.T   # For better notation, I trasnpose the (10, batch_size) matrix
       e_y = zeros(y.shape)  # This will hold e^y
       out = zeros(y.shape)  # this will hold e^y/ e^y.sum
       for i in range(y.shape[0]): # For each element of the batch
15       e_y[i] = exp(y[i] - max(y[i]))  # Subtracting the max for each i
         out[i] = e_y[i]/e_y[i].sum()    # gets cancelled out in the division
       return out.T        # undo the transpose that was done above then return

   def id_layer(y, W, b):
20     '''Return the output of a layer for the input matrix y. y
       is an NxM matrix where N is the number of inputs for a single case, and M
       is the number of cases'''
       return dot(W.T, y)+b
```

```
25 def forward(x, W0, b0):
       ''' Go through the network once. x is the input data,
       Wb is the weight matrix for the first (and only) layer,
       b0 is the bias for the first (and only) layer.
       '''
30     L0 = id_layer(x, W0, b0)
       output = softmax(L0)
       return L0, output
   #%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   # Run part 1 first
35 random.seed(1)
   # Generate a random 784 by 10 matrix
   WRand = 2*random.rand(784, 10)-1

   # generate a 0 bias term
40 b = (2*random.rand(10)-1).reshape((10,1))
   x = M["train4"][0:5].T
   #x = M["train4"][0:5]
   print(x.shape)
   print(WRand.T.shape)
45 # Test the forward function using the generated data
   L1, out=forward(x,WRand,b)
   print("out = ", out)
   print("sum of entries in the columns of out = ",sum(out, axis=0))
```



```
(784, 5)
(10, 784)
out =  [[1.21057486e-02 2.62559036e-02 5.41351175e-06 4.39455559e-03
  8.97739639e-02]
 [1.96263902e-03 1.55568709e-03 1.20003035e-04 2.62237573e-03
  3.26777445e-02]
 [2.19307146e-02 6.05253902e-01 1.83429080e-08 1.72111104e-05
  1.43050614e-02]
 [8.07472603e-05 2.23232431e-02 2.06644662e-05 3.97319304e-04
  3.53175508e-01]
 [9.32509663e-01 2.90016999e-02 9.67863923e-01 9.88788808e-01
  3.35956303e-01]
 [2.05865529e-06 1.48322233e-01 1.71666441e-11 7.55856564e-04
  3.23486376e-02]
 [6.83335778e-03 1.05020197e-04 8.23644040e-05 7.01575465e-04
  2.89560456e-03]
 [2.20641432e-04 1.48922392e-04 1.37209118e-04 5.14577475e-06
  7.18289353e-04]
 [2.39532293e-02 1.07097190e-01 3.17702669e-02 1.49448967e-04
  1.34328908e-01]
 [4.01200175e-04 5.99361990e-02 1.37208190e-07 2.16770366e-03
  3.81997965e-03]]
sum of entries in the columns of out =  [1. 1. 1. 1. 1.]
```

Figure 2: The vectors of probabilities. Generated using line 45.

Problem 3

Gradient computations.
In this part, we move to the backpropagation step, and this is the most important step in training because it gives the gradients for weights and bias, which are used in the gradient descent phase to update the learning weights.
We adopt the cross-entropy cost function for classification problems, which is simply the sum of the products of all the actual probabilities with the negative log of the predicted probabilities. For multi-class classification problems, the cross-entropy function is known to outperform the squared-loss. The function is defined as:

$$\text{cost} = -\sum_{i=1}^{k} y_{-i} \log(y_i) \ ,$$

where y_{-i} is the actual distribution and y_i is our predicted probability distribution, which is network's prediction. We keep tracking the cost to monitor the network performance and averaging the cost function over m examples in the gradient computation step.
For backpropagation, we need to know how the cost function changes based on W and b (The weights and biases). We use the `deriv_sin glelayer()` function to compute the gradient of the cross-entropy cost function based on parameters W and b of the network. The gradient acts as the optimal change that needs to be added to W to better estimate y.

Part 3 code

```
def cost(y, y_):
    return -sum(y_*log(y))

def deriv_singlelayer(W0, b0, x, L0, y, y_):
    ''' Function for computing the gradient of the cross-entropy
    cost function w.r.t the parameters of a neural network
    Input:
    W0 <- n_pixels x n_classes
    b0 <- n_classes x 1
    x <- n_pixels x n_examples
    L0 <- n_classes x n_examples
    y <-  n_classes x n_examples
    y_ <- n_classes x n_examples'''
    dCdx =  y - y_
    dCdx /= x.shape[1] # divide by n_examples as a first step to averaging
    dCdW0 = dot(x, dCdx.T)
    dCdb0 = sum(dCdx, axis=1, keepdims=True) # finalize averaging for db0
    return dCdb0, dCdW0
#******************************************************************
#******************************************************************

from keras.datasets import mnist
# Baseline MLP for MNIST dataset
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import np_utils
from keras.callbacks import LambdaCallback
import tensorflow as tf

# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# flatten 28*28 images to a 784 vector for each image
```

```
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape((X_train.shape[0], num_pixels)).astype('float32')
X_test = X_test.reshape((X_test.shape[0], num_pixels)).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255.0
X_test = X_test / 255.0
# one hot encode outputs (changes the list of numbers 0-9 to a list of 0s and a 1)
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]

WRand = 2*random.rand(784, 10)-1
b = (2*random.rand(10)-1).reshape((10,1))
x = array(X_train[50:100]).T
L1, y = forward(x, WRand, b)
y_ = array(y_train[50:100]).T
print("WRand = ", WRand.shape)
print("x = ", x.shape, type(x))
print("y_ = ", y_.shape)
print("y = ", y.shape)
dCdb, dCdWRand = deriv_singlelayer(WRand, b, x, L1, y, y_)
WRand -= dCdWRand
plot_W(WRand)
```

Problem 4

Verifying the gradients from part 3.
In part 4, we perform a gradient check using the finite-difference approximation to compare the analytic gradient to the numerical gradient. The formula is given by:

$$\frac{dC}{dw_{ij}} = \frac{C(w_{ij}+h) - C(w_{ij})}{h} ,$$

where C is the cost function and h is a very small number, we choose $h = 1e^{-}5$ in the implementation. Figure 3 demonstrates that the two gradients look the same (The first is generated using `f_a_gradient()`, the second with `deriv_singlelayer()`), and the gradient check is completed.

Part 4 code

```
def f_a_gradient(x, W, b, h, y_):
  L1, y = forward(x, W, b)
  dW = zeros(W.shape)
  for i in range(W.shape[0]):
    for j in range(W.shape[1]):
      W[i][j] += h      # add the small h to one entry
      Lh, yh = forward(x, W, b)
      dW[i][j] = (cost(yh, y_) - cost(y, y_))/h
      W[i][j] -= h       # undo the change for the next entry
  return dW
#**********************************************************
#**********************************************************
seed(9)
W0 = 2*rand(784, n_classes)-1
b0 = zeros(n_classes).reshape((n_classes,1))
h = 1e-5
x = array(X_train[0:50]).T
y_ = array(y_train[0:50]).T

dW = f_a_gradient(x, W0, b0, h, y_)
plot_W(dW)
L0, y = forward(x, W0, b0)
dCdb, dCdW = deriv_singlelayer(W0, b0, x, L0, y, y_)
plot_W(dCdW)
```

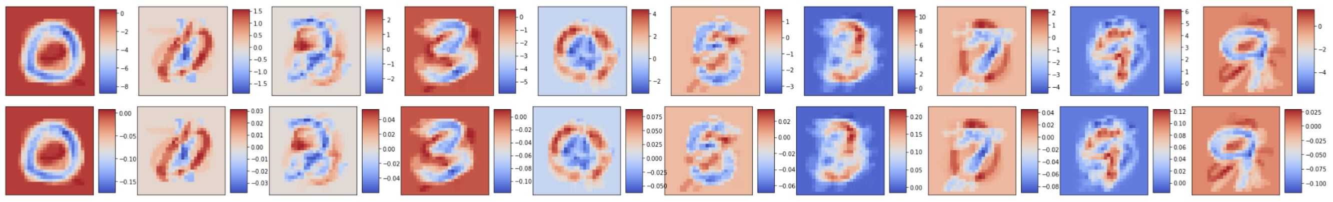


Figure 3: The matching graphs prove the validation of the gradients in part 3.

Problem 5

Performance testing.
The final step of the training process is to update weights that we get during backward propagation (part 3). Gradient descent is an iterative algorithm for finding the minimum, and we use mini-batch gradient descent to update weights, with learning rate = 0.1 and batch size of 50. In this way, we can achieve updates much faster by looking at only a small portion of the training examples at a time.
The correct and incorrect digit recognition determined by the testing model is seen in Figures 4 and 5.

The following 20 digits were classified **correctly**.

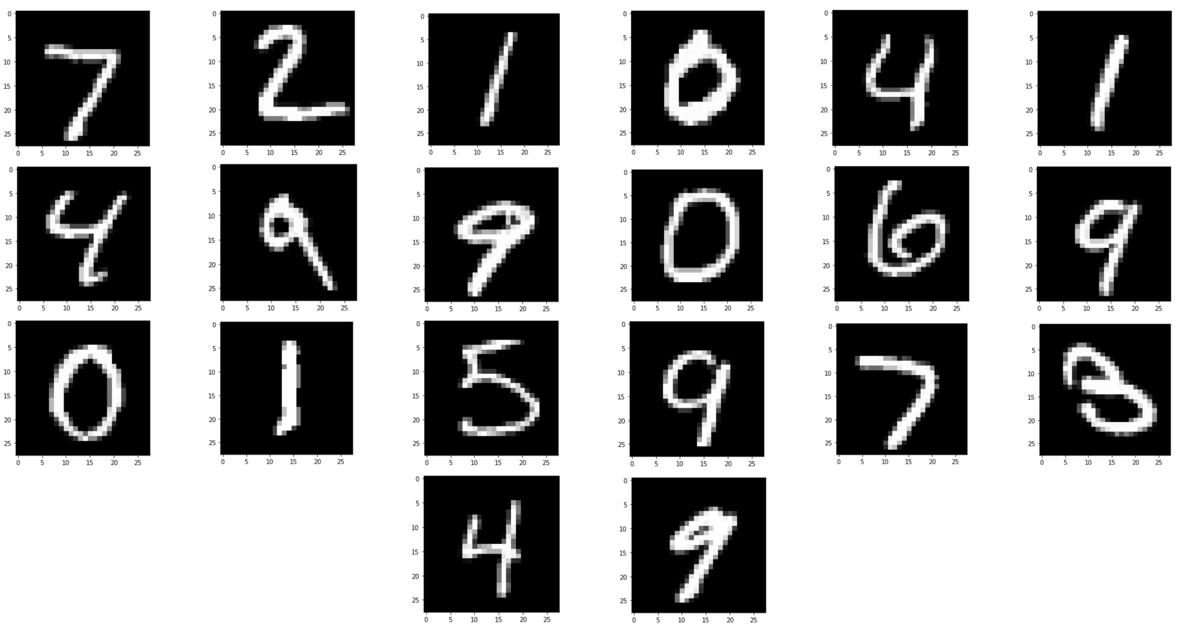


Figure 4: The digits are correctly classified.

The following 10 digits from the test set were classified **incorrectly**.

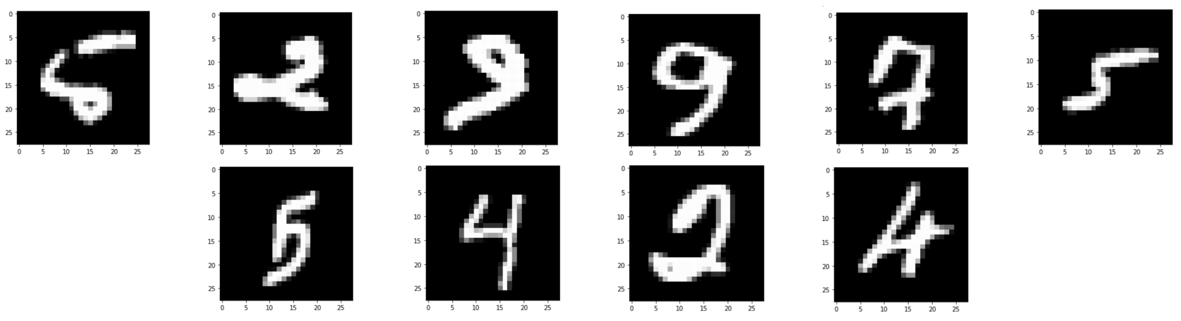


Figure 5: The digits are incorrectly classified.

The following are learning curves for training and test sets. We observe that the more epochs, the lower the error rate; particularly, the algorithm's accuracy improves as the algorithm trains more samples.

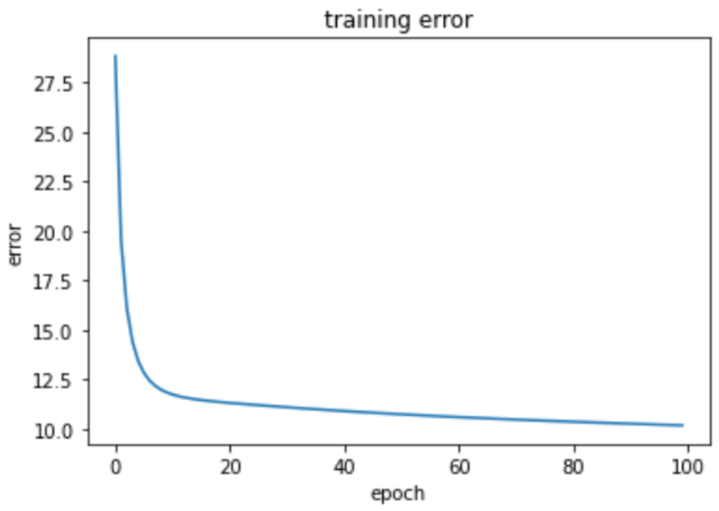


Figure 6: The learning curve of the training set.

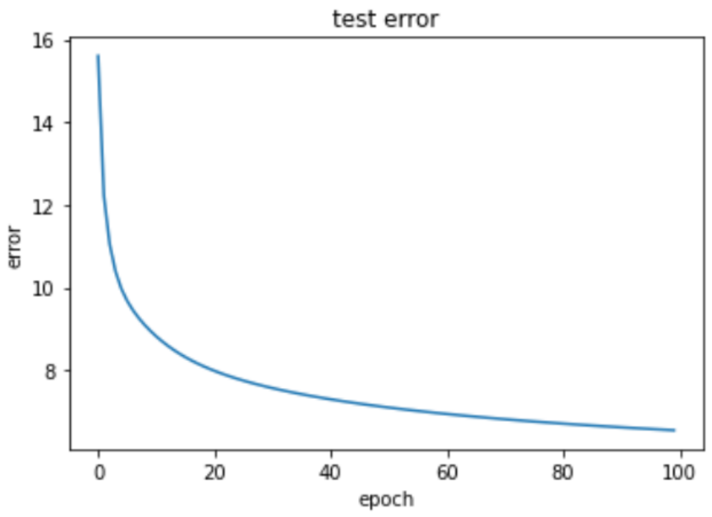


Figure 7: The learning curve of the test set.

Problem 6

Visualizing the weights.
In this part, we plot the weights as heat maps to get an idea of what the network is using to predict each digit.
From Figure 8, we observe the hole in the center of the image for digit 0, the center line for digit 1 and the familiar shape of the digit 3. The other digits in Figure 9 are not as clear but do have their characteristic features pointed out through red pixels.

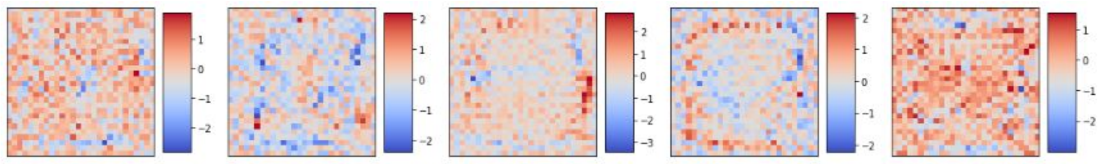


Figure 8: The heat maps show digits from 0 to 4.

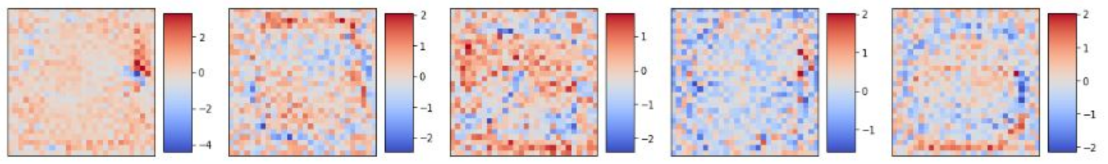


Figure 9: The heat maps show digits from 5 to 9.

Problem 7

Using a hidden layer for digit classification.
In this part, we implement a neural network with a hidden layer for digit classification. Our network uses tanh as a hidden layer activation function and 300 hidden units.

- **Load Data**

  The code starts by loading data using the Keras library. The Keras library already contains some datasets such as MNIST. Keras helps us to build and train models efficiently by saving time in code implementation. The inputs are images with 28 by 28-pixel squares. The code flattens 28×28 images to a 784 vector for each image and normalizes inputs. Normalizing the input data helps to speed up the training.

- **Create Model**

  We create a model using `Sequential()` that allows us to add any layer to the model. Furthermore, using `Dense` allows us to connect each node of every layer to all the nodes of the next layer. As required in the problem, we consider `tanh` as the activation function. The weights in the layers are organized as arrays of dictionaries and get updated in each epoch and after each mini-batch of size 50. The output layer has one neuron for each class of digits, from 0 to 9. As mentioned in problem 2, a Softmax transfer function is used on the output layer to turn the outputs into probability-like values.

- **Compile Model**

  In this step, we compile the model by taking three parameters: optimizer, loss and metrics. For optimizer, we defined `opt`; it helps us to choose our desired learning rate. Learning rate controls the weights' change, it smooths the weights' progress. We use `Sparse-Categorical-Cross-entropy` for loss, and for metrics, we use `accuracy` to perceive the accuracy score on the validation set while training the model.

- **Train model**

  We fit the model on the training data set. Here, validation data is the test set. We consider $epochs = 50$, which means the algorithm learns all data 50 times from the beginning to the end. In order to see the accuracy and loss of each epoch, we used `verbose`.

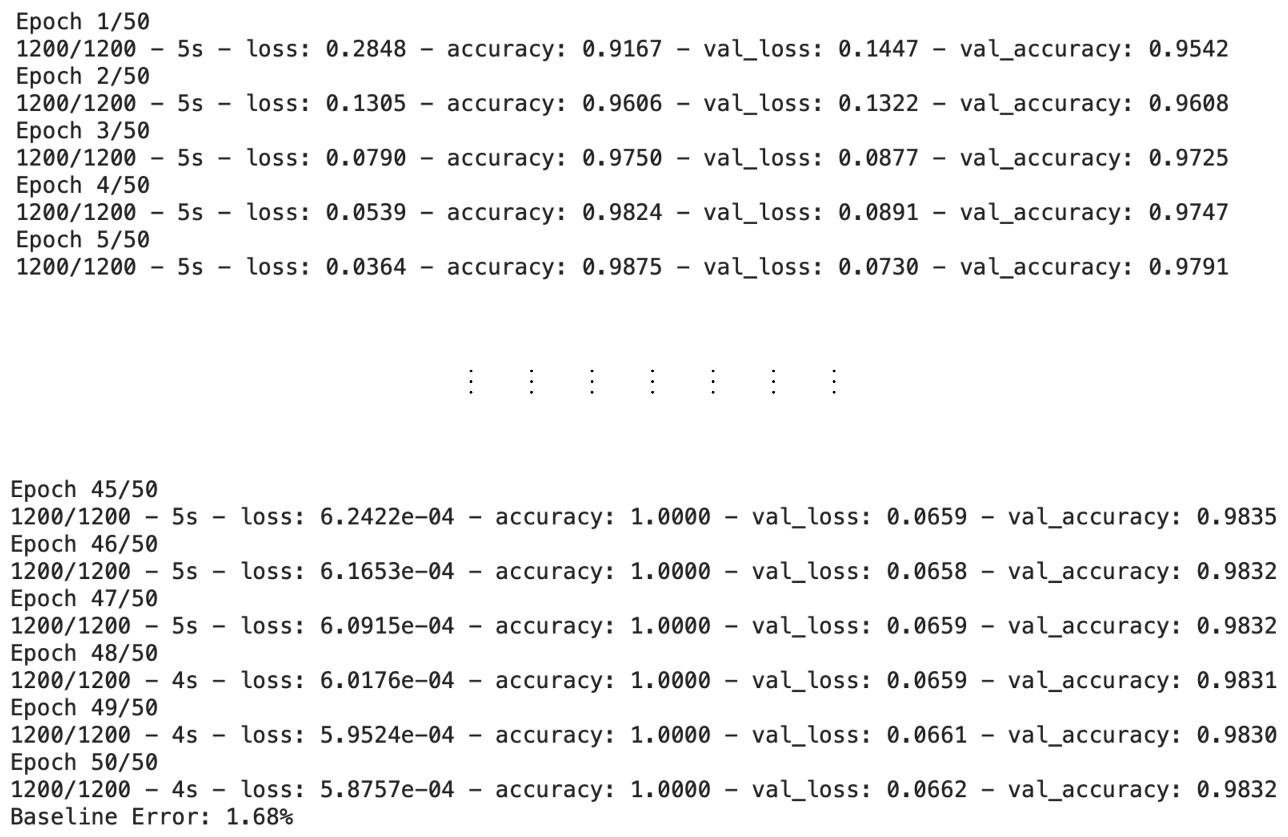
- **Evaluate Model**

  Finally, we evaluate the model using " `model.evaluate()`" to estimate the Baseline Error.

Problem 8

Evaluation of model.

For this part, we ran the code provided in part 7. We applied the learning rate of 0.1 and batch sizes of 50. The following is the result obtained from running the code with epoch=50. In Figure 10, accuracy is being improved by training the network with more epochs. For example, in $epochs = 1$, the loss is 0.2848 and accuracy is 0.9167 while in $epochs = 50$, loss turns to $5.8757e^{-4}$ and accuracy is 1.0000. After epoch=2, we can obtain test classification performance of over 95 percents correct classification.



```
Epoch 1/50
1200/1200 - 5s - loss: 0.2848 - accuracy: 0.9167 - val_loss: 0.1447 - val_accuracy: 0.9542
Epoch 2/50
1200/1200 - 5s - loss: 0.1305 - accuracy: 0.9606 - val_loss: 0.1322 - val_accuracy: 0.9608
Epoch 3/50
1200/1200 - 5s - loss: 0.0790 - accuracy: 0.9750 - val_loss: 0.0877 - val_accuracy: 0.9725
Epoch 4/50
1200/1200 - 5s - loss: 0.0539 - accuracy: 0.9824 - val_loss: 0.0891 - val_accuracy: 0.9747
Epoch 5/50
1200/1200 - 5s - loss: 0.0364 - accuracy: 0.9875 - val_loss: 0.0730 - val_accuracy: 0.9791
```

$$\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots$$

```
Epoch 45/50
1200/1200 - 5s - loss: 6.2422e-04 - accuracy: 1.0000 - val_loss: 0.0659 - val_accuracy: 0.9835
Epoch 46/50
1200/1200 - 5s - loss: 6.1653e-04 - accuracy: 1.0000 - val_loss: 0.0658 - val_accuracy: 0.9832
Epoch 47/50
1200/1200 - 5s - loss: 6.0915e-04 - accuracy: 1.0000 - val_loss: 0.0659 - val_accuracy: 0.9832
Epoch 48/50
1200/1200 - 4s - loss: 6.0176e-04 - accuracy: 1.0000 - val_loss: 0.0659 - val_accuracy: 0.9831
Epoch 49/50
1200/1200 - 4s - loss: 5.9524e-04 - accuracy: 1.0000 - val_loss: 0.0661 - val_accuracy: 0.9830
Epoch 50/50
1200/1200 - 4s - loss: 5.8757e-04 - accuracy: 1.0000 - val_loss: 0.0662 - val_accuracy: 0.9832
Baseline Error: 1.68%
```

Figure 10: Evaluate the model after each epoch.

As the learning rate sets to 0.1, batch sizes of 50, and epoch=50, the corresponding learning curves are displayed in Figure 11 and 12.

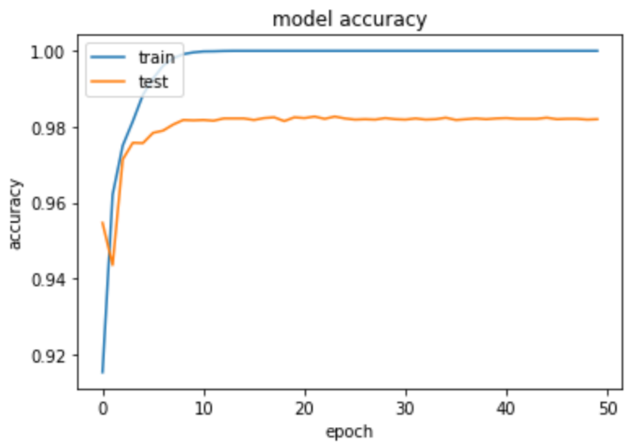


Figure 11: Accuracy Curve

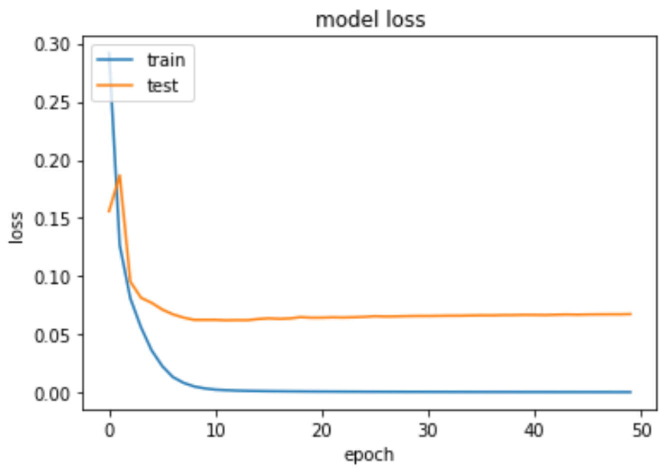


Figure 12: Loss Curve

The following 20 digits were classified **correctly**.

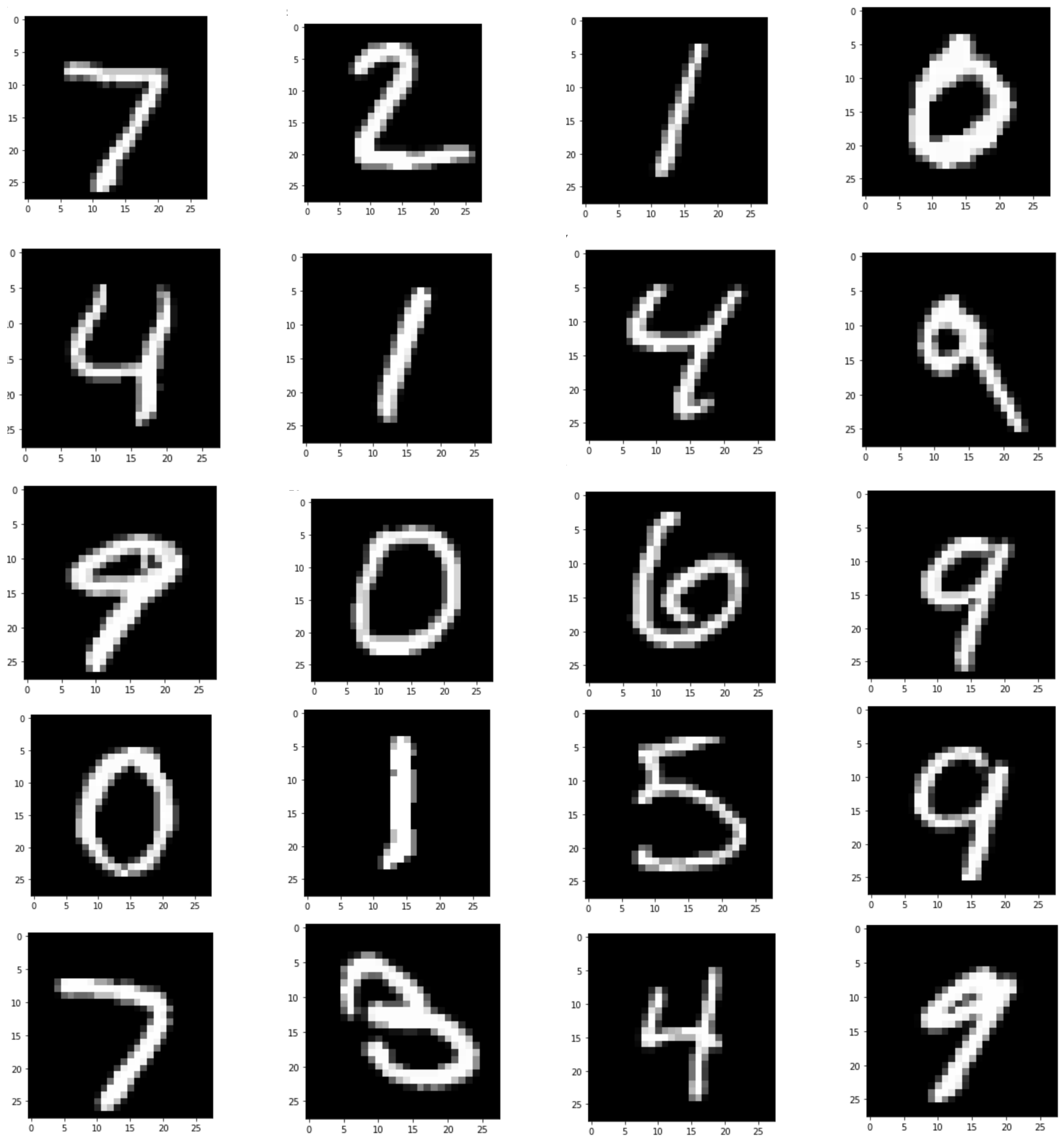


Figure 13: The digits are correctly classified.

The following 10 digits from the test set were classified **incorrectly**.

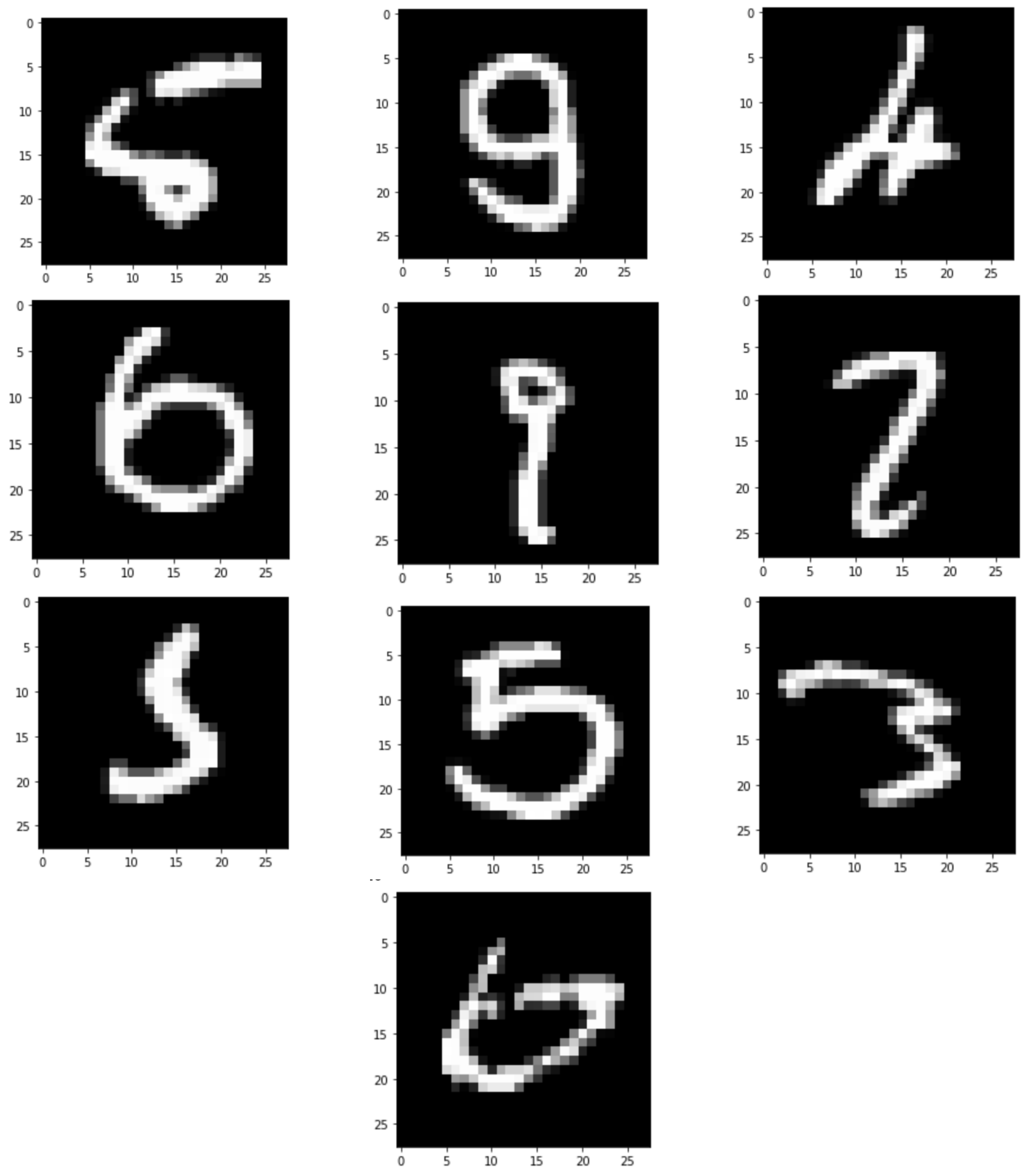


Figure 14: The digits are incorrectly classified.

Problem 9

Visualizing the weights.
In this section, we visualize W's connected to the hidden layer as if they were digits.
We know in the input layer of our model, the model has 784 weights corresponding to the 28×28 pixel input images. In the (first) hidden layer we have $784\times$ (number of hidden units) weights, in our case we have 784×300 weights.
We can reshape each of the columns of the (784×300) matrix to 28×28 images. Since these weights are in the hidden layer of the neural network, the images have low level features. Hence, visualizing the weights will tend to show incomplete or features of digits instead of complete ones.
We selected W(2) and W(299) out of the total of 300 Ws from the last epoch. In Figure 15, we can see a question mark. In reality, it is suspected that this is a mix of digit 2 and digit 9. In Figure 16, there is a distinct blue diagonal line. This diagonal can be attributed to the numbers 2, 7, 8 and 9, indicating that if this neuron produces a high value after the activation function (i.e. fires), we would increase the output value of these numbers.

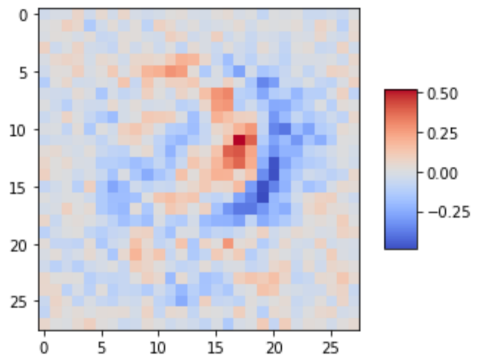


Figure 15: The weights connecting the hidden unit that corresponds to W(299) to the output units (before the application of tanh) are:
$[0.1888029, 0.4102521, 0.2384080, -0.5223419, 0.3521273, -1.241525, -0.2427901, 0.1020877, -0.5807386, 1.206554]$

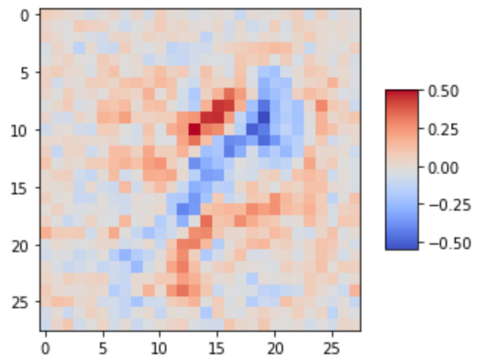


Figure 16: The weights connecting the hidden unit that corresponds to W(2) to the output units (before the application of tanh) are :
$[0.240613, 0.4430935, -0.521063, -0.9100362, 0.4861983, 1.354505, -0.1662664, 0.2744546, -0.82103, -0.289754]$