

Neural Networks

Assignment 1

Question 1

The optimization function that we want to find the prior for is:

$$W_{MAP} = \arg \min_W \sum_n (y^n - h(x^n, W))^2 + \beta \sum_i w_i^2$$

Which is same as

$$W_{MAP} = \arg \max_W -(\sum_n (y^n - h(x^n, W))^2 + \beta \sum_i w_i^2)$$

Now, the MAP is calculated as: (where $D = \{x_i, y_i\}_{i=1}^N$)

$$W_{MAP} = \arg \max_W P(W|D)$$

Using Bayes' rule

$$W_{MAP} = \arg \max_W \frac{P(D|W)P(W)}{P(D)}$$

$P(D)$ does not depend on W and does not affect W_{MAP} , therefore it can be ignored

$$W_{MAP} = \arg \max_W P(D|W)P(W)$$

Any scalar α not dependent on W can be multiplied and it does not change the results of the $\arg \max$ function.

$$W_{MAP} = \arg \max_W P(D|W) \cdot \alpha \cdot P(W)$$

Maximizing a function is same as maximizing the log of that function. Therefore, we take the log to convert the product into summation to match this with the optimization function.

$$W_{MAP} = \arg \max_W [\ln P(D|W) + \ln(\alpha \cdot P(W))]$$

Comparing this with the optimization function as

$$\arg \max_W [\ln P(D|W) + \ln(P(W))] = \arg \max_W -(\sum_n (y^n - h(x^n, W))^2 + \beta \sum_i w_i^2)$$

$$\ln(\alpha \cdot P(W)) = -\beta \sum_i w_i^2$$

$$\ln(\alpha \cdot P(W)) = -\beta W^T W$$

$$\alpha P(W) = e^{-\beta W^T W}$$

$$P(W) = \frac{e^{-\beta W^T W}}{\alpha}$$

Following would be the Gaussian Distribution of W with mean μ and covariance matrix C.

$$P(W) = \frac{1}{(2\pi)^{N/2} \sqrt{|C|}} e^{-\frac{1}{2}((W-\mu)^T C^{-1} (W-\mu))}$$

$P(W) = \frac{e^{-\beta W^T W}}{\alpha}$ can be written as:

$$P(W) = \frac{e^{-\frac{1}{2}(W-0)^T (2\beta I) (W-0)^T}}{\alpha}$$

Therefore, we can see that the prior distribution of W is Gaussian with:

$$\boxed{\mu = 0}$$

$$C^{-1} = 2\beta I$$

$$\boxed{C = \frac{I}{2\beta}}$$

$$\alpha = (2\pi)^{N/2} \sqrt{|C|}$$

$$\alpha = (2\pi)^{N/2} \sqrt{\frac{|I|}{4\beta^2}}$$

$$\boxed{\alpha = \frac{(2\pi)^{N/2}}{2\beta}}$$

Where α does not depend on W

So, W is distributed as $N \sim \left(0, \frac{I}{2\beta}\right)$

$$\boxed{P(W) = \frac{e^{-\beta W^T W}}{(\frac{\sqrt{2\pi}}{2\beta})^N}}$$

Question 2

a)

$O = (X_1 \text{ OR NOT } X_2) \text{ XOR } (\text{NOT } X_3 \text{ OR NOT } X_4)$ is the logical operation that we are trying to implement using neural network. However, we first need to simplify the expression and bring it into at least four terms in order to use 4 neuron units in the hidden layer which may then be combine using and OR gate in the output layer. For that we first get the truth table and then try to simplify the expression using k-maps simplification. Following is the truth table.

table - NumPy array

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	1

Format Resize ☒ Background color

Save and Close Close

Figure 1: Truth table of the expression above

After k-maps simplification, we obtain the following expression.

$$O = \bar{X}_1 X_2 \bar{X}_3 + \bar{X}_1 X_2 \bar{X}_4 + \bar{X}_2 X_3 X_4 + X_1 X_3 X_4$$

We have to implement 4 AND gates and then OR them. First and second term give the following truth table and the corresponding inequalities are mentioned below.

X_1	X_2	X_3/X_4	$O_{1/2}$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

We use weights W_{ji} for input X_i and b_j for bias connected with neuron j in our derivations. The input not in the truth table expression will be assigned a weight of 0.

$$-b_1 < 0$$

$$W_{12} - b_1 > 0$$

$$W_{13} - b_1 < 0$$

$$W_{12} + W_{13} - b_1 < 0$$

$$W_{11} - b_1 < 0$$

$$W_{11} + W_{13} - b_1 < 0$$

$$W_{11} + W_{12} - b_1 < 0$$

$$W_{11} + W_{12} + W_{13} - b_1 < 0$$

One such set of weights are as follows which satisfy the inequalities:

$$W_{12} = 1$$

$$W_{13} = -1$$

$$W_{11} = -1$$

$$W_{14} = 0$$

$$b_1 = 1$$

The second AND gate, $\bar{X}_1 X_2 \bar{X}_4$, has the same truth table therefore we get:

$$W_{21} = -1$$

$$W_{22} = 1$$

$$W_{24} = -1$$

$$b_2 = 1$$

For $\bar{X}_2 X_3 X_4$, we get the following truth table.

X_2	X_3	X_4	O_3
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

$$-b_3 < 0$$

$$W_{34} - b_3 < 0$$

$$W_{33} - b_3 < 0$$

$$W_{33} + W_{34} - b_3 > 0$$

$$W_{32} - b_3 < 0$$

$$W_{32} + W_{34} - b_3 < 0$$

$$W_{32} + W_{33} - b_3 < 0$$

$$W_{32} + W_{33} + W_{34} - b_3 < 0$$

One such set of weights are as follows which satisfy the inequalities:

$$W_{32} = -1$$

$$W_{33} = 1$$

$$W_{34} = 1$$

$$b_3 = 2$$

For $X_1 X_3 X_4$, we get the following truth table.

X_1	X_3	X_4	O_4
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

$$-b_4 < 0$$

$$W_{44} - b_4 < 0$$

$$W_{43} - b_4 < 0$$

$$W_{43} + W_{44} - b_4 > 0$$

$$W_{41} - b_4 < 0$$

$$W_{41} + W_{44} - b_4 < 0$$

$$W_{41} + W_{43} - b_4 < 0$$

$$W_{41} + W_{43} + W_{44} - b_4 < 0$$

One such set of weights are as follows which satisfy the inequalities:

$$W_{42} = 1$$

$$W_{43} = 1$$

$$W_{44} = 1$$

$$b_4 = 3$$

b)

Using the selected weights and biases, the weight matrix for the hidden layer of non-robust decision boundary is as follows. The fifth column is the biases.

$$W_1 = \begin{bmatrix} -1 & 1 & -1 & 0 & 1 \\ -1 & 1 & 0 & -1 & 1 \\ 0 & -1 & 1 & 1 & 2 \\ 1 & 0 & 1 & 1 & 3 \end{bmatrix}$$

The outputs from the hidden layer are combined with new weights and biases and passed through step function such that an OR gate is implemented.

$$W_2 = [1 \quad 1 \quad 1 \quad 1 \quad 0.5]$$

The (400,4) input matrix is appended with a column of -1s to include the biases. The output is as follows where * represents matrix multiplication

$$O_1 = \text{Step}(X * W_1^T)$$

$$O_1' = O_1 \text{ appended with column of -1s}$$

$$O = \text{Step}(O_1' * W_2^T)$$

Using these equations or neural network implementation, 100% accuracy is achieved which is evident after running the code for question 2.

c)

The network that has been designed in part a can work robustly under noisy conditions if we choose a good set of biases and then corresponding weights. The idea is similar to support vector machines. The biases should form a decision surface such that it lies in the middle of the space separating two classes (0 and 1). In that case any input lying near the decision surface will have no biasness towards one class or the other and the network will perform robustly. The weights/biases for hidden layer obtained for robust surface is as follows. They are chosen by taking the weights as 1s and -1s respectively for inputs with and without NOT gate. Then the bias is chosen to be in the middle of allowed range. Bias plays the key role in maximizing the margin between classes. The weights then adjust the slope/ gradient of the separating surface. The output weights/ bias does not affect the robustness.

$$W_1 = \begin{bmatrix} -1 & 1 & -1 & 0 & 0.5 \\ -1 & 1 & 0 & -1 & 0.5 \\ 0 & -1 & 1 & 1 & 1.5 \\ 1 & 0 & 1 & 1 & 2.5 \end{bmatrix}$$

$$W_2 = [1 \quad 1 \quad 1 \quad 1 \quad 0.5]$$

d)

After using both set of weights, the performance is evaluated and is found as follows. The accuracies of the input with noise may change every time due to randomness of the noise but it can be seen that the robust boundary performs better than the non-robust one every time.

output - Dictionary (3 elements)

Key	Type	Size	Value
Accuracy Non Robust	float64	1	82.75
Accuracy Robust	float64	1	91.25
Accuracy part b	float64	1	100.0

Save and Close Close

Figure 2: Accuracies with different boundaries

Question 3

a) Following are the samples displayed form each class.

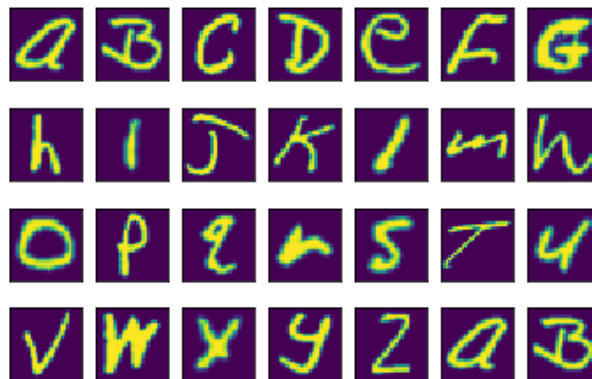


Figure 3: Visualized classes A-Z

Following is the visualized correlation matrix between different classes.

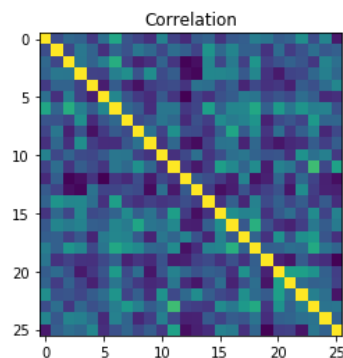


Figure 4: Correlation matrix

It can be seen that the within correlation between different classes is low whereas correlation between same class is high (across the diagonal). As an example, we can see that there is high similarity between the letter C and G and there corresponding correlation is also high as can be seen in the figure. Whereas, similarity between A and Z is very low and hence their corresponding correlation is very low.

b)

Different learning rates are tried in the range (0, 1) and 0.05 gives the best performance. Therefore, it is used.

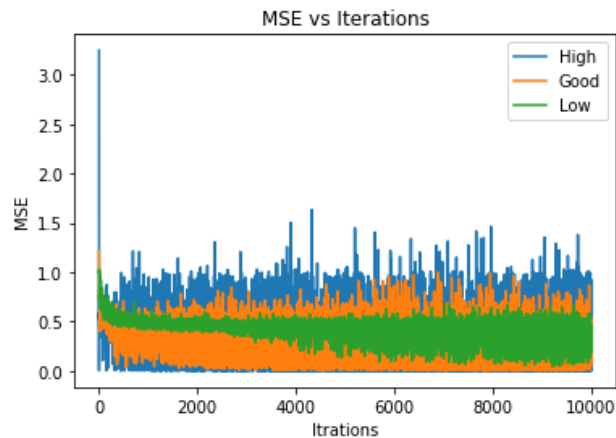


Figure 5: Training MSE over 10000 iterations

Final network weights with learning rate 0.05 are displayed below.

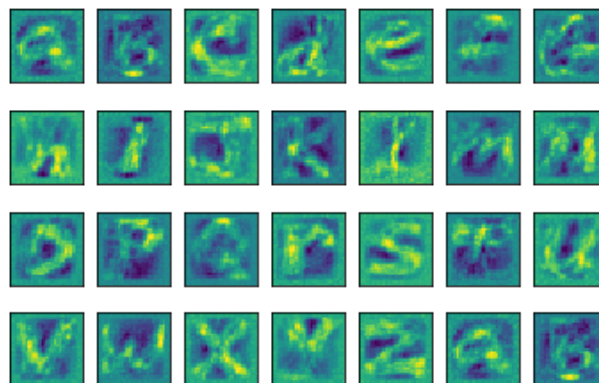


Figure 6: Trained weights

Analyzing the weights, we can see that the trained weights are similar to the actual letters. At the central part of the images, the values of the weights are high where they overlap with the actual letters, whereas the values of the weights are low where they do not overlap. The values of the weights at the edges are moderate since they do not contribute to making the decision as the edges are generally empty for all training images.

The gradient descent update rule in vectorized form is as follows:

$$\begin{aligned} w &= w - \eta \frac{dJ}{dw} \\ b &= b - \eta \frac{dJ}{db} \end{aligned}$$

$$\text{Where } J = \frac{1}{2}(y - \hat{y})^T(y - \hat{y})$$

$$\text{And } \hat{y} = \sigma(w^T x + b)$$

$$\frac{dJ}{dw} = \frac{dJ}{d\hat{y}} \cdot \frac{d\hat{y}}{dw}$$

$$\frac{dJ}{db} = \frac{dJ}{d\hat{y}} \cdot \frac{d\hat{y}}{db}$$

$$\frac{dJ}{d\hat{y}} = -(y - \hat{y})$$

The matrix multiplication is represented by * and element wise by dot (.).

$$\frac{d\hat{y}}{dw} = x^T * \left(\sigma(w^T x + b) \cdot (1 - \sigma(w^T x + b)) \right)^T = x^T * (\hat{y} \cdot (1 - \hat{y}))^T$$

$$\frac{dJ}{dw} = -x^T * ((y - \hat{y}) \cdot \hat{y} \cdot (1 - \hat{y}))^T$$

$$\frac{d\hat{y}}{db} = \sigma(w^T x + b) \cdot (1 - \sigma(w^T x + b)) = \hat{y} \cdot (1 - \hat{y})$$

$$\frac{dJ}{db} = -(y - \hat{y}) \cdot \hat{y} \cdot (1 - \hat{y})$$

c) The training is done using same initial weights and the learning rates are chosen as:

$$\eta_{\text{High}} = 0.5$$

$$\eta_{\text{Good}} = 0.05$$

$$\eta_{\text{Low}} = 0.005$$

The MSE curves are shown below for each learning rate.

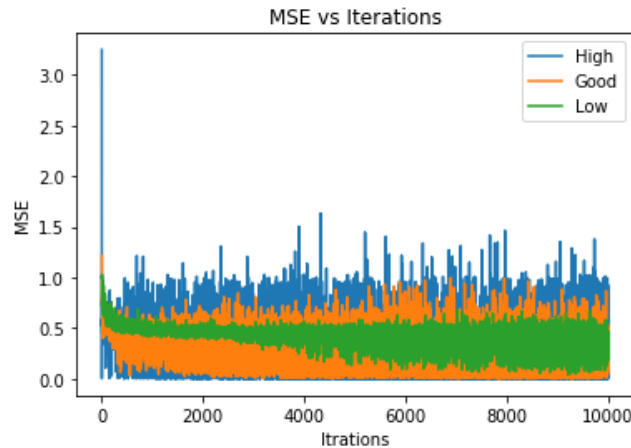
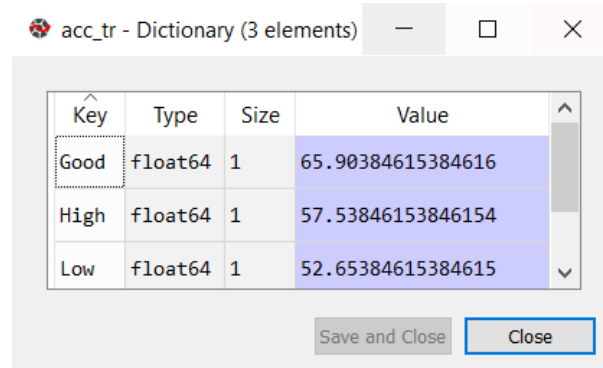


Figure 7: Training MSE over 10000 iterations

The MSE curves show that the high learning rate causes the error to lower quickly but every iteration introduces a huge distortion and divergent behavior in the learning. Whereas the low learning rate causes the error to decrease very slowly such that it will

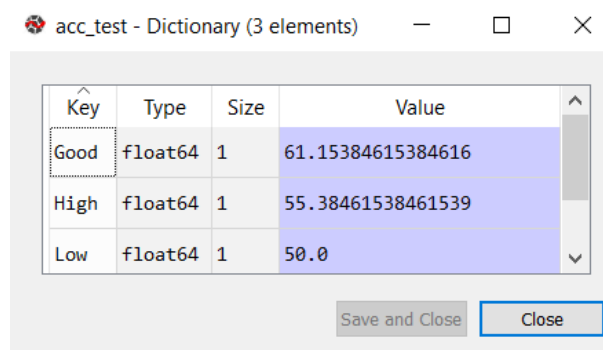
take a lot more iterations to converge. However, the tuned learning rate actually gives a balanced learning. Accuracy on the training data was calculated to be as follows where it can be seen that the tuned learning rate gives the best performance in terms of accuracy.



Key	Type	Size	Value
Good	float64	1	65.90384615384616
High	float64	1	57.53846153846154
Low	float64	1	52.65384615384615

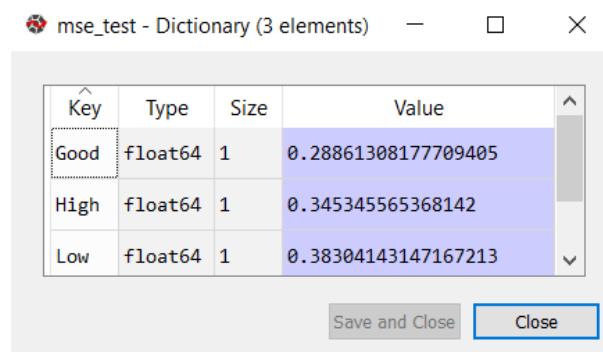
Figure 8: Train Accuracy

- d) After using the trained weights from all three learning weights, MSE and accuracy on the test data has been obtained which can be seen as follows. The results are consistent with our previous conclusions about the learning rate where the tuned rate gives the best performance and least MSE.



Key	Type	Size	Value
Good	float64	1	61.15384615384616
High	float64	1	55.38461538461539
Low	float64	1	50.0

Figure 9: Test Accuracies



Key	Type	Size	Value
Good	float64	1	0.28861308177709405
High	float64	1	0.345345565368142
Low	float64	1	0.38304143147167213

Figure 10: Test MSE

Appendix

```
# -*- coding: utf-8 -*-
```

```
"""
```

```
Created on Sun Oct 13 20:41:41 2019
```

```
@author: basit
```

```
"""
```

```
import sys
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
import scipy.io
```

```
import pylab as py
```

```
question = '2'
```

```
def AbdulBasit_Anees_21600659_hw1(question):
```

```
    if question == '2':
```

```
        print(question)
```

```
        acc, acc_b, acc_c = q2d()
```

```
        return {'Accuracy part b': acc, 'Accuracy Non Robust': acc_b, 'Accuracy Robust': acc_c}
```

```
    elif question == '3':
```

```
        print(question)
```

```
        corr, w, b, acc_tr, acc_test, mse_test = q3()
```

```
        return {'Correlation': corr, 'w': w, 'b': b, 'Train Accuracy': acc_tr, 'Test Accuracy': acc_test, 'Test MSE':  
mse_test}
```

```
def q2d():
```

```
    def truth_table(N):
```

```
        X = np.zeros((2 ** N, N))
```

```
        for j in range(N):
```

```
            nIterate = 2 ** (N - (j + 1))
```

```
            step = 2 ** j
```

```
    prev = step
    for i in range(nIterate):
        X[prev:prev + step, N - (j + 1)] = np.ones((step))
        prev = prev + 2 * step
    return X

def NNb(w1, w2):
    X = truth_table(4)
    Y = np.logical_xor(X[:,0] + (X[:,1] == 0), ((X[:,2] == 0) + (X[:,3] == 0)))
    X = np.hstack((X, -1*np.ones((16,1))))
    o = (np.hstack((X @ w1.T >= 0), -1*np.ones((16,1)))) @ w2 >= 0
    trues = np.sum(o == Y)
    acc = 100 * trues / 16
    return acc

def NNd(w1, w2, sd):
    X = truth_table(4)
    Y = np.logical_xor(X[:,0] + (X[:,1] == 0), ((X[:,2] == 0) + (X[:,3] == 0)))
    y = np.tile(Y, 25)
    noise = np.random.normal(0, sd, (25, 16, 4))
    x = (X + noise).reshape((400, 4))
    x = np.hstack((x, -1*np.ones((400, 1))))
    o = (np.hstack((x @ w1.T >= 0), -1*np.ones((400, 1)))) @ w2 >= 0
    trues = np.sum(o == y)
    acc = trues / 4
    return acc

#Non robust boundary weights
w1 = np.array((( -1, 1, -1, 0, 1), (-1, 1, 0, -1, 1), (0, -1, 1, 1, 2), (1, 0, 1, 1, 3)))
w2 = np.array((1, 1, 1, 1, 0.5))
acc = NNb(w1, w2)
sd = 0.2
acc_b = NNd(w1, w2, sd)
#Robust boundary weights
```

```
w1 = np.array(((1,1,-1,0,0.5),(-1,1,0,-1,0.5),(0,-1,1,1,1.5),(1,0,1,1,2.5)))
w2 = np.array((1,1,1,1,0.5))
acc_c = NNd(w1, w2, sd)
return acc, acc_b, acc_c
```

```
def q3():
```

```
    #Load data
    data = scipy.io.loadmat("assign1_data1.mat")
    x_tr = data["trainims"].transpose((2,0,1))
    x_test = data["testims"].transpose((2,0,1))
    y_tr = data["trainlbls"]
    y_test = data["testlbls"]
```

```
def make_grid(images):
```

```
    _, grid = plt.subplots(nrows=4,ncols=7)
    for p in range(28):
        if p < 26:
            grid[int(p/7),p%7].imshow(images[p%26])
            grid[int(p/7),p%7].axes.get_xaxis().set_visible(False)
            grid[int(p/7),p%7].axes.get_yaxis().set_visible(False)
    #Show letters from each class
    images = []
    for i in range(1,27):
        index = np.argwhere(y_tr==i)[0,0]
        plt.figure()
        images.append(x_tr[index,:,:])
        plt.imshow(x_tr[index,:,:])
    make_grid(images)
```

```
def corr(a, b):
```

```
    product = np.mean((a - a.mean()) * (b - b.mean()))
    stds = a.std() * b.std()
    product = product / stds
```

```
    return product

#Plot correlation
correlation = np.zeros((26,26))
for i in range(26):
    for j in range(26):
        correlation[i,j] = corr(images[i], images[j])

plt.figure()
plt.title('Correlation')
plt.imshow(correlation)


def to_categorical(y):
    out = np.zeros((len(y),26))
    for i in range(26):
        out[:,i] = ((y-1) == i).reshape(len(y))
    return out


def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def train(w, b, x_tr, y_trn, iterations, rate):
    W = np.zeros((iterations, 28 * 28, 26))
    n = 0
    mse = np.zeros((iterations, 1))
    while n < iterations:
        ind = np.random.randint(0, x_tr.shape[0])
        x = x_tr[ind,:].reshape((28*28,1))
        y = y_trn[ind,:].reshape(26,1)
        y_p = sigmoid((w.T @ x) + b)
        grad_w = x @ (- (y - y_p) * (y_p * (1 - y_p))).T
        grad_b = - (y - y_p) * (y_p * (1 - y_p))
        w = w - rate * grad_w
        b = b - rate * grad_b
        W[n, :, :] = w
```

```
mse[n, :] = (y - y_p).T@(y - y_p)/2
n = n + 1
return w, b, mse

def accuracy(w, b, x_tr, y_tr, samples):
    y_pr = sigmoid((x_tr.reshape((samples,28*28)) @ w)+b.T)
    y_pr = (np.argmax(y_pr, 1).reshape((samples,1))) + 1
    trues = np.sum(y_pr == y_tr)
    acc = 100 * (trues / samples)
    return acc

#Initialize variables
x_tr = x_tr.reshape((5200, 28*28))
x_tr = (x_tr/255)
y_train = to_categorical(y_tr)
iterations = 10000
L = [0.5, 0.05, 0.005]
labels = ["High","Good","Low"]
acc_tr = { }
errors = []
weights = { }
biases = { }
w_0 = np.random.normal(0, 0.01, (28 * 28, 26))
b = np.random.normal(0, 0.01, (26, 1))
#Train
for i in range(len(L)):
    w, b, mse = train(w_0, b, x_tr, y_train, iterations, L[i])
    acc = accuracy(w, b, x_tr, y_tr, 5200)
    acc_tr[labels[i]] = (acc)
    weights[labels[i]] = (np.copy(w))
    biases[labels[i]] = (np.copy(b))
    errors.append(mse)

#Display weights
w_disp = []
```

```
for i in range(26):
    w_disp.append(weights['Good'][:, i].reshape((28,28)))
    plt.figure()
    plt.imshow(w_disp[i])
make_grid(w_disp)
#Plot errors
plt.figure()
for i in range(len(L)):
    plt.plot(np.arange(iterations),errors[i], label = labels[i] )
py.legend(loc = 'upper right')
plt.xlabel('Iterations')
plt.ylabel('MSE')
plt.title('MSE vs Iterations')
plt.show()
# Find test accuracy and loss
x_test = x_test.reshape((1300, 28*28))
x_test = (x_test/255)
y_testt = to_categorical(y_test)
w = weights
b = biases
acc_test = { }
mse_test = { }
for i in range(len(w)):
    acc_test[labels[i]] = (accuracy(w[labels[i]], b[labels[i]], x_test, y_test, 1300))
    y_pr = sigmoid((x_test.reshape((1300,28*28)) @ w[labels[i]]+ b[labels[i]].T)
    mse_test[labels[i]] = (np.sum(((y_testt - y_pr)**2))/(2*1300))
plt.close('all')
return correlation, w, b, acc_tr, acc_test, mse_test
```

output = AbdulBasit_Anees_21600659_hw1(question)