Ayhan Okuyan
21601531

EEE443/543 Neural Networks - Assignment 1

Table of Contents

Ayhan Okuyan
21601531

Question 1

The question asks us to analytically find the prior distribution of the weights of a network with m neurons. The network weights are obtained by the minimization,

$$J(W) = argmin_W \sum_n (y^n - h(x^n, W))^2 + \beta \sum_i w_i^2$$

We will be using the rule that the posterior distribution of a random variable is the multiplication of its prior and the likelihood. Hence, we need to convert this argmin function to multiplication of two functions. In order to achieve that, we will be applying the function

$$f(x) = e^{-x}$$

This function is a monotonously decreasing function which will only convert the equation to an argmax function, and the critical point will not change.

$$f(J(W)) = argmax_W \, e^{-\sum_n (y^n - h(x^n,W))^2 - \beta \sum_i w_i^2}$$

$$= argmax_W \, e^{-\sum_n (y^n - h(x^n,W))^2} e^{-\beta \sum_i w_i^2}$$

If we are to assume that the data Is drawn from a Gaussian distribution, the expression given below holds and it becomes similar to the argmax function, which is the maximum a posteriori (MAP) estimation of the weights.

$$P(W|y^n) = \frac{P(W) * P(y^n|W)}{\int P(W) * P(y^n|W) * dW}$$

Here we can simply ignore the denominator since it is essentially a constant that comes out of the integral, after all what we are applying is an argument maximizing procedure. Hence, we can say that,

$$P(W|y^n) \propto P(W) * P(y^n|W)$$

Considering the argmax function, here we can accept the term with the squared error can be considered as the likelihood function and the sum of the squared weights multiplied with a constant can be separated as the prior distribution of the weights, with the scalar constant multiplied ignored. Finally, if we assume that the data follows a Gaussian distribution, we can separate the inside of argmax as follows and assume the likelihood and prior are in the form,

$$(Likelihood)\ L = \frac{1}{A} e^{-\frac{\varphi}{2}\sum_n (y^n - h(x^n,W))^2}$$

$$(Prior)\ P = Ae^{-\beta \sum_i w_i^2}$$

Ayhan Okuyan
21601531

In order to find the coefficient A, we will use the fact that the sum over all probability distribution equals to 1, which is

$$\int_{-\infty}^{\infty} f_X(x)dx = 1$$

Hence,

$$A \int_{-\infty}^{\infty} e^{-\beta \sum_i w_i^2} dw_i = 1$$

Furthermore, we know that we have m neurons, so we can write this integral as multiplications of m integrals.

$$A \int_{-\infty}^{\infty} e^{-\beta w_1^2} dw_1 \int_{-\infty}^{\infty} e^{-\beta w_2^2} dw_2 \dots \int_{-\infty}^{\infty} e^{-\beta w_m^2} dw_m = 1$$

Furthermore, we know that

$$\int_{-\infty}^{\infty} e^{-\beta w_i^2} dw_i = \frac{\sqrt{\pi}}{\sqrt{\beta}}$$

Then the equation becomes

$$A(\frac{\sqrt{\pi}}{\sqrt{\beta}})^m = 1$$

$$A = (\frac{\beta}{\pi})^{m/2}$$

Finally, the prior distribution becomes,

$$P = \left(\frac{\beta}{\pi}\right)^{m/2} e^{-\beta \sum_i w_i^2}$$

Question 2

Question 2 mentions an engineer that would like to design a neural network with a single hidden layer and four input neurons (with binary inputs), and a single output neuron to implement the following logic operation.

$$(x_1\ or\ \overline{x_2})\ xor\ (\overline{x_3}\ or\ \overline{x_4})$$

Furthermore, we were required to implement a hidden layer with four hidden units and a unipolar activation function, hence I have chosen the unit step activation as offered.

Part A

This part requires us to derive the set of inequalities that will be used to find the hyperplane decision boundaries. As we could not create an "xor" gate through one neuron since the "xor" itself cannot be separated through one decision boundary, we have written the xor in the combination of "and" and "or" gates as below. Given a and b as input terms "a xor b" can be written as

$$a\ xor\ b = (a\ and\ \overline{b})\ or\ (\overline{a}\ and\ b)$$

Hence, we can write the logic operation as,

$$(x_1 + \overline{x_2}) \oplus (\overline{x_3} + \overline{x_4}) = (x_1 + \overline{x_2})\,(x_3 x_4) + (\overline{x_1} x_2)\,(\overline{x_3} + \overline{x_4})$$

$$= (x_1 x_3 x_4) + (\overline{x_2} x_3 x_4) + (\overline{x_1} x_2 \overline{x_3}) + (\overline{x_1} x_2 \overline{x_4})$$

Hence, we have four three-input and gates and one four input or gate, which all of them can be implemented with a single neuron each. This approach that is recovered is also compatible with the criteria that is requested by the question. Hence, we move on to finding the inequalities for each neuron. Here, we see that in each neuron, one of the weights for that neuron doesn't have any impact on the overall outcome and I will be giving those weights zero for simplicity.

1st Neuron

Here, we implement the logic function $(x_1 x_3 x_4)$. Hence, we construct the truth table for this logic operation.

X1	X3	X4	O1
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**

Figure 1 – Truth Table for the 1st Neuron

Hence, we can define the output of this neuron as,

$$o_1 = u(w_{11}x_1 + w_{13}x_3 + w_{14}x_4 - \theta_1)$$

Where u is the described unit step function given as below.

$$u(x) = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases}$$

Then, we can further find the inequalities as,

$$w_{13} + w_{14} - \theta_1 < 0$$
$$w_{11} + w_{14} - \theta_1 < 0$$
$$w_{11} + w_{13} - \theta_1 < 0$$
$$w_{11} + w_{13} + w_{14} - \theta_1 \geq 0$$
$$w_{11} - \theta_1 < 0$$
$$w_{13} - \theta_1 < 0$$
$$w_{14} - \theta_1 < 0$$
$$-\theta_1 < 0$$

2nd Neuron

In this neuron, we implement the logic function $(\overline{x_2}x_3x_4)$. Hence, we construct the truth table for this logic operation.

X2	**X3**	**X4**	**O2**
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**

Figure 2 – Truth Table for the 2nd Neuron

Hence, we can define the output of this neuron as,

$$o_2 = u(w_{22}x_2 + w_{23}x_3 + w_{24}x_4 - \theta_2)$$

Hence, the inequalities become as follows,

$$w_{23} + w_{24} - \theta_2 \geq 0$$
$$w_{22} + w_{24} - \theta_2 < 0$$
$$w_{22} + w_{23} - \theta_2 < 0$$
$$w_{22} + w_{23} + w_{24} - \theta_2 < 0$$
$$w_{22} - \theta_2 < 0$$
$$w_{23} - \theta_2 < 0$$
$$w_{24} - \theta_2 < 0$$
$$-\theta_2 < 0$$

3rd Neuron

In this neuron, we implement the logic function $(\overline{x_1}x_2\overline{x_3})$. Hence, we construct the truth table for this logic operation.

X1	X2	X3	O3
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**

Figure 3 – Truth Table for the 3rd Neuron

Hence, we can define the output of this neuron as,

$$o_3 = u(w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + w_{34}x_4 - \theta_3)$$

Hence, the inequalities become as follows,

$$w_{33} + w_{32} - \theta_3 < 0$$
$$w_{31} + w_{33} - \theta_3 < 0$$
$$w_{31} + w_{32} - \theta_3 < 0$$
$$w_{31} + w_{32} + w_{33} - \theta_3 < 0$$
$$w_{31} - \theta_3 < 0$$
$$w_{32} - \theta_3 \geq 0$$
$$w_{33} - \theta_3 < 0$$

$$-\theta_3 < 0$$

4th Neuron

In this neuron, we implement the logic function $(\overline{x_1}x_2\overline{x_4})$. Hence, we construct the truth table for this logic operation.

X1	X2	X4	O4
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**

Figure 4 – Truth Table for the 4th Neuron

Hence, we can define the output of this neuron as,

$$o_4 = u(w_{41}x_1 + w_{42}x_2 + w_{44}x_4 - \theta_4)$$

Hence, the inequalities become as follows,

$$w_{41} + w_{42} - \theta_4 < 0$$

$$w_{41} + w_{44} - \theta_4 < 0$$

$$w_{42} + w_{44} - \theta_4 < 0$$

$$w_{41} + w_{42} + w_{44} - \theta_4 < 0$$

$$w_{41} - \theta_4 < 0$$

$$w_{42} - \theta_4 \geq 0$$

$$w_{44} - \theta_4 < 0$$

$$-\theta_4 < 0$$

Output Neuron

After finishing the hidden layer outputs, we should implement the or gate that connects the hidden layer outputs, in the sections before, I have defined the outputs of the hidden layer neurons as $o_i, i\epsilon\{1,2,3,4\}$, hence I will be using those outputs as inputs in this section. Hence, I have constructed the truth table for the logic equation.

O1	**O2**	**O3**	**O4**	**Of**
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**1**
0	0	1	1	**1**
0	1	0	0	**1**
0	1	0	1	**1**
0	1	1	0	**1**
0	1	1	1	**1**
1	0	0	0	**1**
1	0	0	1	**1**
1	0	1	0	**1**
1	0	1	1	**1**
1	1	0	0	**1**
1	1	0	1	**1**
1	1	1	0	**1**
1	1	1	1	**1**

Figure 5 – The Truth Table for the Output Neuron

Furthermore, we can give the output expression as,

$$o_f = u(w_{51}o_1 + w_{52}o_2 + w_{53}o_3 + w_{54}o_4 - \theta_5)$$

Hence, we can define the inequalities as follows,

$$w_{51} - \theta_5 \geq 0$$

$$w_{52} - \theta_5 \geq 0$$

$$w_{53} - \theta_5 \geq 0$$

$$w_{54} - \theta_5 \geq 0$$

$$w_{51} + w_{52} - \theta_5 \geq 0$$

$$w_{51} + w_{53} - \theta_5 \geq 0$$

$$w_{51} + w_{54} - \theta_5 \geq 0$$

$$w_{52} + w_{53} - \theta_5 \geq 0$$

$$w_{52} + w_{54} - \theta_5 \geq 0$$

$$w_{53} + w_{54} - \theta_5 \geq 0$$

$$w_{51} + w_{52} + w_{53} - \theta_5 \geq 0$$

$$w_{51} + w_{52} + w_{54} - \theta_5 \geq 0$$

$$w_{51} + w_{53} + w_{54} - \theta_{55} \geq 0$$

$$w_{52} + w_{53} + w_{54} - \theta_5 \geq 0$$

$$w_{51} + w_{52} + w_{53} + w_{54} - \theta_5 \geq 0$$

$$-\theta_{55} < 0$$

Part B

In this part, we were asked to choose a weight and bias matrix that would cover for all the inequalities that are portrayed in Part A. Here, I the strategy that I followed was trial and error and picked random integers that would satisfy these inequalities. Hence, the weight matrix for the hidden layer that I have come up with are as follows,

$$W = \begin{bmatrix} 2 & 0 & 3 & 3 \\ 0 & -3 & 3 & 3 \\ -3 & 3 & -2 & 0 \\ -2 & 6 & 0 & -3 \end{bmatrix} \theta = \begin{bmatrix} 7 \\ 4 \\ 2 \\ 5 \end{bmatrix}$$

$$W = \left[\begin{array}{cccc:c} 2 & 0 & 3 & 3 & 7 \\ 0 & -3 & 3 & 3 & 4 \\ -3 & 3 & -2 & 0 & 2 \\ -2 & 6 & 0 & -3 & 5 \end{array}\right]$$

The Python script to create this matrix is given below.

```
hiddenWeights = np.matrix('2 0 3 3 7; 0 -3 3 3 4; -3 3 -2 0 2; -2 6 0 -3 5')
```

Then in order to test if the neural network operates with an accuracy of 100%, I have created the input matrix that contains one of each of the possible input combinations and I have concatenated the weight matrix and the bias vector to handle the forwarding of the input with only one matrix multiplication. The constructed input matrix is given below.

$$X = \begin{bmatrix} 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 1 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 1 & -1 \\ 0 & 1 & 1 & 0 & -1 \\ 0 & 1 & 1 & 1 & -1 \\ 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 1 & -1 \\ 1 & 0 & 1 & 0 & -1 \\ 1 & 0 & 1 & 1 & -1 \\ 1 & 1 & 0 & 0 & -1 \\ 1 & 1 & 0 & 1 & -1 \\ 1 & 1 & 1 & 0 & -1 \\ 1 & 1 & 1 & 1 & -1 \end{bmatrix}$$

Here is the code used to construct this matrix.

```
inputs = np.zeros([16,5])
i3 = -1
i2 = -1
i1 = -1
i0 = -1

for i in range(16):
    inputs[i,0] = i0
    inputs[i,1] = i1
    inputs[i,2] = i2
    inputs[i,3] = i3

    i3 = -i3
    if(i % 2):
        i2 = -i2
    if(i % 4 == 3):
        i1 = -i1
    if(i % 8 == 7):
        i0 = -i0
inputs[inputs < 0] = 0
inputs[:,4] = -1
```

Hence, we can find the output of this layer by the equation

$$O_{hidden} = u(W.X^T)$$

In order to be able to run this operation, we have defined a unit step function as follows,

```
 def unitStep(x):
    x[x >= 0] = 1
    x[x < 0] = 0
    y = x
    return y
```

Then, we were able to recover the hidden layer outputs as follows.

```
hiddenOuts = unitStep(np.matmul(hiddenWeights,inputs.T))
```

We have found the weights and the bias term for the output neuron as below.

$$W_o = \begin{bmatrix} 2 & 3 & 4 & 5 \end{bmatrix} \ \theta = 1$$

$$W_o = \left[\begin{array}{cccc:c} 2 & 3 & 4 & 5 & 1 \end{array}\right]$$

After that we have transposed the O_{hidden} matrix and added a column for the bias term and then calculated,

$$O_f = u(W_o^T.O_{hidden}')$$

The code segment is below.

```
hiddenOuts = hiddenOuts.T
outBiases = np.ones([16,1])
outBiases[outBiases > 0] = -1
hiddenOuts = np.concatenate((hiddenOuts, outBiases), axis=1)
outWeights = np.matrix('2 3 4 5 1')
```

```
outs = unitStep(np.matmul(outWeights, hiddenOuts.T))
```

After finding the network outputs, I have created the logic function that we were supposed to implement as a function and passed the input matrix through this function in order to match the outputs that we have recovered in order to observe if they are a perfect match. However, in order to do that, we have first defined and xor function to be able to use that with the requested logic function. The code segments for this part are below.

```
def xor(x1, x2):
    return (x1 and (not x2)) or ((not x1) and x2)

def logicFnc(x):
    return xor(x[0] or (not x[1]), (not x[2]) or (not x[3]))

inputsCheck = inputs[:,0:4]
checkOuts = list()

for i in range(16):
    checkOuts.append(logicFnc(inputsCheck[i,:]))
```

Then finally, we match the two output vectors to see if they match by the following code

```
print((outs == checkOuts).all())
True
```

Here, we observe that the implemented network works 100% as requested.

Part C

In the previous section, we have observed that the network works perfectly under stable conditions, where the input didn't contain any noise. However, if there is a noise factor involved, the system wouldn't as good as desired, since the noise affects the input values and may change them between two binary values depending on the value of the noise. Hence, I wouldn't consider the created network as robust to noise.

In order to find the most robust decision boundary, I have selected the weights either 1 or -1 for orthogonality and symmetry and then decided on the bias values that are in the middle of the inequalities. We have used 1 for positive inputs and -1 for inverted inputs and zero for the non-existing inputs as explained in Part B. This, way the noise would affect the network less and the system would become more robust. Hence, we have used the inequalities that we have found in the previous question as follows.

1st Neuron

Here, we were implementing the logic equation $(x_1 x_3 x_4)$ and the weights become as,

$$w_{11} = 1$$

$$w_{13} = 1$$

$$w_{14} = 1$$

In order to find bias, we use the following inequalities,

$$w_{11} + w_{13} - \theta_1 < 0$$

$$w_{11} + w_{13} + w_{14} - \theta_1 \geq 0$$

Hence for θ_1, we can observe the following inequality.

$$w_{11} + w_{13} < \theta_1 \leq \ w_{11} + w_{13} + w_{14}$$

$$2 < \theta_1 \leq \ 3$$

Hence, we choose the bias in the middle of this inequality as 2.5.

2nd Neuron

Here, we were implementing the logic equation $(\overline{x_2}x_3x_4)$ and the weights become as,

$$w_{22} = -1$$

$$w_{23} = 1$$

$$w_{24} = 1$$

In order to find bias, we use the following inequalities,

$$w_{22} + w_{23} + w_{24} - \theta_2 < 0$$

$$w_{23} + w_{24} - \theta_2 \geq 0$$

Hence for θ_2, we can observe the following inequality.

$$w_{22} + w_{23} + w_{24} < \theta_2 \leq \ w_{23} + w_{24}$$

$$1 < \theta_2 \leq \ 2$$

Hence, we choose the bias in the middle of this inequality as 1.5.

3rd Neuron

Here, we were implementing the logic equation $(\overline{x_1}x_2\overline{x_3})$ and the weights become as,

$$w_{31} = -1$$

$$w_{32} = 1$$

$$w_{33} = -1$$

In order to find bias, we use the following inequalities,

$$w_{32} - \theta_3 \geq 0$$

$$-\theta_3 < 0$$

Hence, we can observe the bias term as

$$0 < \theta_3 \leq w_{32}$$

$$0 < \theta_3 \leq 1$$

Hence, we choose θ_3 as 0.5.

4th Neuron

Here, we were implementing the logic equation $(\overline{x_1}x_2\overline{x_4})$ and the weights become as,

$$w_{41} = -1$$

$$w_{42} = 1$$

$$w_{43} = -1$$

In order to find bias, we use the following inequalities,

$$w_{42} - \theta_4 \geq 0$$

$$-\theta_4 < 0$$

Hence, we can observe the bias term as,

$$0 < \theta_4 \leq w_{42}$$

$$0 < \theta_4 \leq 1$$

Hence, we choose θ_4 as 0.5, as we have in the 3rd neuron. After finding the hidden layer weights, we have found the output layer weights.

Output Neuron

Here we have implemented the logic equation $(o_1 + o_2 + o_3 + o_4)$. Here, as we have done with the previous neurons, for each input that is positive, meaning all of them, we insert 1 as their weights. Hence, the weights become,

$$w_{51} = w_{52} = w_{53} = w_{54} = 1$$

Hence, we use the following inequalities in order to find the appropriate bias term.

$$w_{51} - \theta_5 \geq 0$$

$$-\theta_5 < 0$$

Hence, we find the fifth and the final bias term as,

$$0 < \theta_5 \leq w_{51}$$

$$0 < \theta_5 \leq 1$$

Hence, we select θ_5 as 0.5. As the process of finding the weights and biases are done, we can observe the final coefficients in the matrix form. The first set of weights and biases are given for the hidden layer while the second is for the output layer.

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

$$W_o = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \; \theta_o = 0.5$$

Part D

In this part of the question, we are asked to test both of the networks that we have created in order to find out if the network that we have constructed in Part C functions more robustly than the one we have created in Part B in the presence of a noise factor.

In order to create the input matrix, the question requires us to create an input matrix, and while columns represent the inputs, there would be 25 of each possible input combination concatenated and since there are 16 possible configurations, the final matrix turns out to be a 400x4 matrix. After that, we have created a noise matrix that has the same dimensions and drawn from a Gaussian distribution with zero mean and 0.2 standard deviation.

$$\boldsymbol{N}_{ixj}, \qquad N_{i,j} \sim Gaussian(0,0.2)$$

Then, we would reach the final input matrix by linearly combining the two matrices. As follows.

$$\boldsymbol{X} = X + \boldsymbol{N}$$

The Python code to construct this matrix can be found below. In the code, we use the input matrix which was created in Part B since we have one row of each combination of inputs already present in that matrix.

```
inputsRobust = np.zeros([400,5])
for i in range(16):
    for j in range(25):
        inputsRobust[i*25+j,:] = inputs[i]

meanPrm = 0
stdPrm = 0.2
noiseMatrix = np.random.normal(meanPrm, stdPrm, 2000).reshape(400,5)
noiseMatrix[:,4] = 0

inputsNoise = inputsRobust + noiseMatrix
```

After the completion of this step, we have constructed the weights and biases that we have calculated in Part B in the following form with the code below.

$$W = \left[\begin{array}{cccc:c} 1 & 0 & 1 & 1 & 2.5 \\ 0 & -1 & 1 & 1 & 1.5 \\ -1 & 1 & -1 & 0 & 0.5 \\ -1 & 1 & 0 & -1 & 0.5 \end{array}\right]$$

```
hiddenWeightsRobust = np.matrix('1 0 1 1 2.5; 0 -1 1 1 1.5; -1 1 -1 0 0
.5; -1 1 0 -1 0.5')
```

After that, we have used the following code the we have used in Part B to find the outputs of the network with the robust biases that we have found, here we have also defined the output layer weights and biases as given below.

$$W_o = \left[\begin{array}{cccc:c} 1 & 1 & 1 & 1 & 0.5 \end{array}\right]$$

```
hiddenOutsRobust = unitStep(np.matmul(hiddenWeightsRobust,inputsNoise.T))
hiddenOutsRobust = hiddenOutsRobust.T
outBiases = np.ones([400,1])
outBiases[outBiases > 0] = -1
hiddenOutsRobust = np.concatenate((hiddenOutsRobust, outBiases), axis=1)
outWeightsRobust = np.matrix('1 1 1 1 0.5')
outsRobust = unitStep(np.matmul(outWeightsRobust, hiddenOutsRobust.T))
```

Then, we have applied the same procedure to the network that is constructed in Part B.

```
hiddenOuts = unitStep(np.matmul(hiddenWeights,inputsNoise.T))
hiddenOuts = hiddenOuts.T
outBiases = np.ones([400,1])
outBiases[outBiases > 0] = -1
```

```
hiddenOuts = np.concatenate((hiddenOuts, outBiases), axis=1)
outWeights = np.matrix('2 3 4 5 1')
outs = unitStep(np.matmul(outWeights, hiddenOuts.T))
```

After that, we have used the logicFnc function that we have constructed beforehand in order to create the output matrix for the 400x4 input matrix in order to be able to check the correlation between the theoretical outputs and the neural networks' outputs.

```
inputsCheckRobust = inputsRobust[:,0:4]
checkOuts = list()

for i in range(400):
    checkOuts.append(logicFnc(inputsCheckRobust[i,:]))
```

After that we can finally compare the outputs and see how network has performed, with the help of the following code.

```
true_cnt_normal = 0
for i in range(400):
    if(checkOuts[i] == outs[0,i]):
        true_cnt_normal += 1
print('Accuracy = ' + str(true_cnt_normal/400*100) + "%")
```

The output for one random set of noises is as follows.

```
Accuracy = 85.75%
```

Then, we use the same code to find the accuracy for the robust network.

```
true_cnt_robust = 0
for i in range(400):
    if(checkOuts[i] == outsRobust[0,i]):
        true_cnt_robust += 1
print('Accuracy = ' + str(true_cnt_robust/400*100) + "%")
```

The output for the same set of inputs is as below

```
Accuracy = 91.0%
```

These findings set the completion of this part and the question. We can see that the robust network performed with an accuracy of 91% while the non-robust network for which we have created by assigning random values (all the inequalities were satisfied) have performed with 85.75%. The most important observation was that we were able to create a significantly better network by selecting weights and biases more carefully. While the non-robust network could apply the intended logic function to 343 out of 400 samples, the robust network could correctly give the output of 364 samples out of 400. However, we should note, that the percentage is not 100 since the gaussian noise with 0.2 standard deviation reversed the input values between one and zero. If the standard deviation could be smaller, we would expect the accuracy to go even higher, up to 100% after some value.

Question 3

Part A

For this part, we have first acquired the training images and their labels, then used the first sample of each class for visualization.

```
train_ims = np.load('train_images.npy')
train_labels = np.load('train_labels.npy')
train_size = train_labels.shape[0]


cur_letter = 1
sample_ind = list()

row = 4
col = 7
fig = plt.figure()

for i in range(train_size):
    if(cur_letter == train_labels[i]):
        ax = plt.subplot(row, col, cur_letter)
        plt.imshow(train_ims[:,:,i], cmap='gray')
        ax.axis('off')
        ax.autoscale(False)
        sample_ind.append(i)
        cur_letter += 1
```

Hence the visualizations are given below.

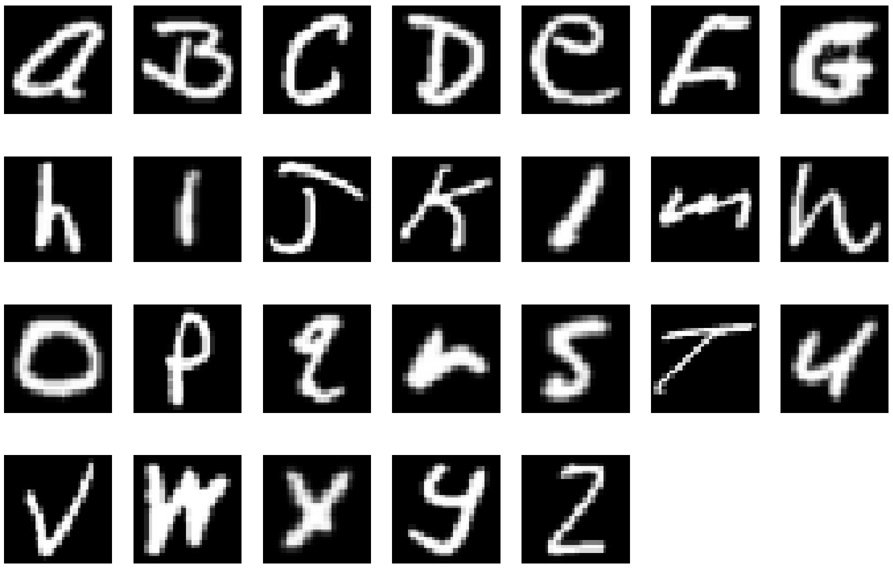

Figure 6 – The Visualizations for Each Class

Hence, we have used the correlation method provided by NumPy which gives the Pearson correlation coefficient given as,

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

This explains the linear correlation between two random variables. The matrix of the correlations between samples of each class are given below with the code used to construct it.

```
num_class = 26
corr_matrix = np.zeros(num_class**2).reshape(num_class,num_class)

for i in range(num_class):
    for j in range(num_class):
        corr_matrix[i,j] = np.corrcoef(train_ims[:,:,sample_ind[i]].fla
t, train_ims[:,:,sample_ind[j]].flat)[0,1]

cormat = pd.DataFrame(corr_matrix)
display(cormat)
```

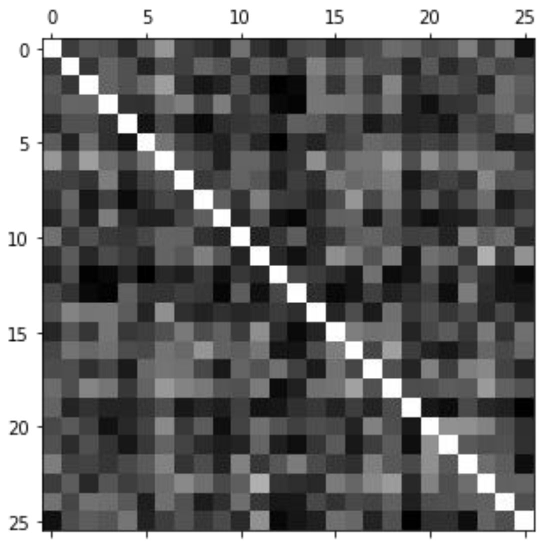

Figure 7 – The Correlation Matrix of the Selected Samples

Here, we see observe a diagonal which is white, as these are all correlation coefficients which vary in between 0 and 1, corresponds to 1. This output is highly natural since the diagonal corresponds to the same images hence,

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \xrightarrow{X=Y} \rho_{X,X} = \frac{cov(X,X)}{\sigma_X \sigma_X} = \frac{var(X)}{\sigma_X \sigma_X} = \frac{\sigma_X^2}{\sigma_X^2} = 1$$

Here, we can presumably assume that the within-class variability is lower than the across class variability. Furthermore, we see that the across-class variability is quite high, darker the color, the lower correlation between the images. However, we can see there are plenty of grey points which correspond to medium correlation which shows that there are some images across-class that are alike, which would eventually make the classification task harder to achieve. Furthermore, in order to understand the within-class variability even better, we could look the correlations in between two set of sample images that are different from each other, only by slightly altering the code in the upper section. Here is the correlation matrix.

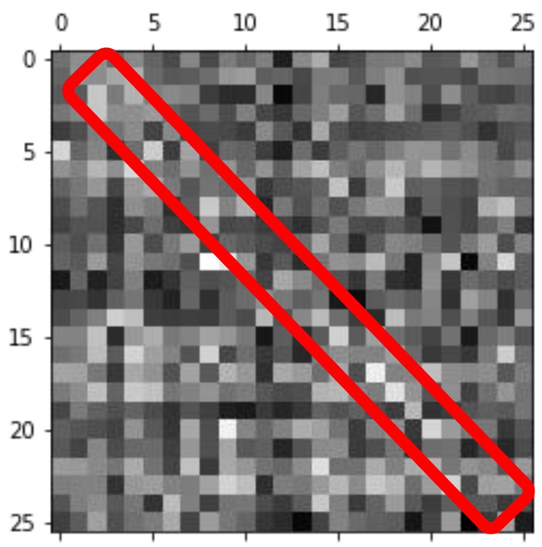

Figure 8 – The Correlation Matrix for two Different Set of Inputs

Here, we see that the diagonal has significantly gone darker, which means the within-class variability is not as low as we have initially considered. Although we can still see the diagonal trend, as highlighted with red , we cannot apply a classification using the correlations only.

Part B

The proposed structure is a perceptron that contains number of neurons that are equal with the number of classes each assigned for a different letter in the alphabet. Hence, in order to use the labels as desired, I have converted the integer labeled outputs to one-hot encoding type of outputs. Since there are 5200 training samples, at the end, I have a label matrix, 26x5200, each column representing the output of a sample.

```
#convert training outputs to one-hot encodings
train_one_hot = np.zeros(num_class*train_size).reshape(num_class,train_size)

for i in range(train_size):
    train_one_hot[train_labels[i]-1,i] = 1
```

In order to create a perceptron, first I have defined some functions that are needed to be defined, being the sigmoid function and a forward function to pass the data forward.

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

def forward(W, x, b):
    return sigmoid(np.matmul(W,x) - b)
```

Then, I have constructed the weight and bias matrix separately. Each image consists of 28x28 pixels; hence I have 784-pixel values for each input value. Furthermore, with the 26 classes, in total we have a 26x784 weight matrix and a 26x1 bias vector. All values are initialized from a Gaussian distribution with mean 0 and standard deviation 0.01.

```
bias = np.random.normal(meanPrm, stdPrm, num_class).reshape(num_class,1)

weights = np.random.normal(meanPrm,stdPrm, num_class*pixelWH**2).reshape(num_class,
pixelWH**2)
```

Then, as requested, we have used a stochastic gradient descent approach to optimize the weights and biases. The measure of error is Mean Squared Error, which is given by,

$$MSE = \frac{1}{N}\sum_{i=0}^{N}||y_i - \hat{y}_i||^2 \; := \frac{1}{N}\sum_{i=0}^{N} \mathrm{L}$$

Hence, we can also give the L as a matrix multiplication as

$$L = \big(Y - \sigma(Wx - B)\big).\,(Y - \sigma(Wx - B))^T$$

Hence, we are using the squared loss in order to update the parameters. The gradient descent update rule is as given which uses a learning rate to calibrate the learning speed, which we will be modifying.

$$W' := W - \mu\frac{\partial \mathrm{L}}{\partial \mathrm{W}}$$

$$B' := B - \mu\frac{\partial \mathrm{L}}{\partial \mathrm{B}}$$

Hence, if we take the derivative of the loss function we get,

$$\frac{\partial \mathrm{L}}{\partial \mathrm{W}} = -2\big(Y - \sigma(WX - B)\big)(\sigma(WX - B))(1 - \sigma(WX - B))X^T$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{B}} = 2(Y - \sigma(WX - B))$$

Then, we have constructed the code snippet below to apply the online learning algorithm for 10000 iterations and train the model according to the update rule shown. We have also collected the mean squared errors as requested by the question.

```
mseList = list()
num_iter = 10000
lrn_rate = 0.04

for i in range(num_iter):
    random_index = random.randint(0,train_size-1)

    random_img = train_ims[:,:,random_index].reshape(pixelWH**2,1)
    random_img = random_img/np.amax(random_img)
    random_y = train_one_hot[:,random_index].reshape(num_class,1)

    y_hat = forward(weights, random_img, bias)

    y_diff = random_y - y_hat

    w_grd = -2*np.matmul(y_diff*(y_hat)*(1-y_hat),np.transpose(random_img))
    b_grd = 2*y_diff*(y_hat)*(1-y_hat)

    weights -= lrn_rate*w_grd
    bias -= lrn_rate*b_grd

    mseList.append(np.sum(y_diff**2)/(y_diff.shape[0]))
```

Here, it is seen that the learning rate has been selected as 0.04, which was found through trial and error in order to observe the best outcome, I have decided the best learning rate by observing the final MSE value. The next part of the question requires us to show the trained network weights as images and visualize them. Here is the Python code and the image grid.

```
row = 4
col = 7


for i in range(num_class):
    ax2 = plt.subplot(row, col, i+1)
    weight_dig = weights[i,:].reshape(pixelWH,pixelWH)
    plt.imshow(weight_dig, cmap='gray')
    plt.rcParams['figure.figsize'] = [8,8]
    ax2.axis('off')
    ax2.autoscale(False)
```

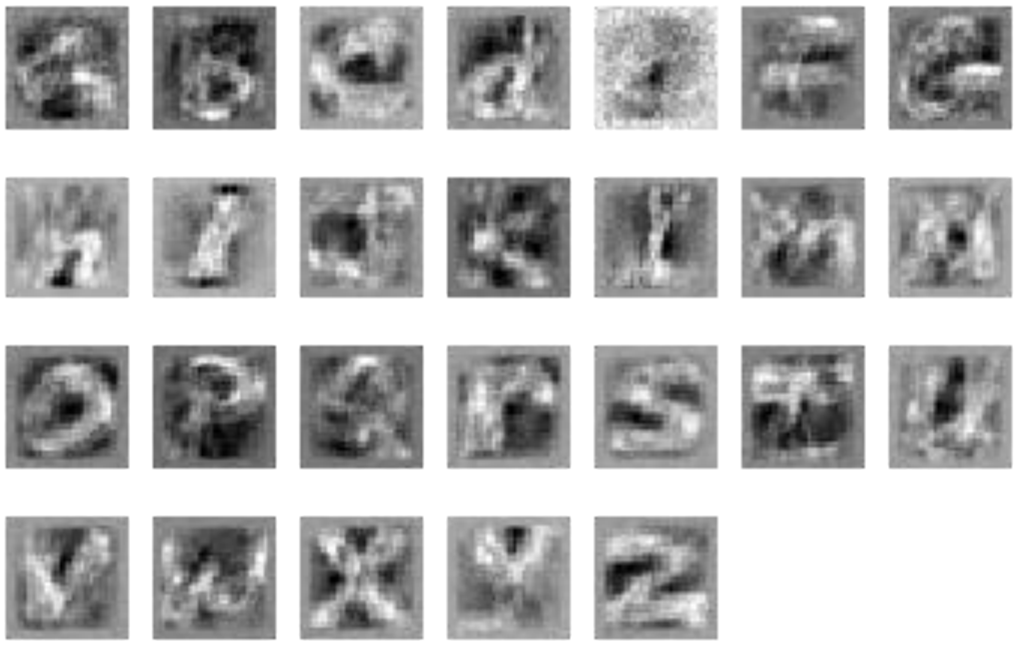

Figure 9 – Visualizations of Neuron Weights for Each Neuron

By looking at the figure, we can easily see that the network is learning since the visualized weights represent the output images that they are mean to represent. However, there are some letters which could not be learned due to the various representations of the letter and note that the dataset contains both upper case and lower-case letters under the same label which makes the learning task harder. We can make an interpretation that the letters whose upper case and lower-case letters look alike are the ones that were learned more successfully like the letter “X” and “x” or “Z” and “z”.

Part C

This part requires us to repeat the online training process that we have done in the previous part with two different learning rates than what we have used, which one is substantially higher then than the optimal learning rate and one is substantially lower. The reason behind this task is to be able to see if these abnormal learning rates makes the gradient descent algorithm stuck in a local minimum. Usually, the learning rate is supposed to be in the interval (0, 1), hence, I have chosen the learning rates as below.

$$\mu_{High} = 0.9$$

$$\mu_{Low} = 0.0001$$

The code that is used for this part is given below.

```
biasHi = np.random.normal(meanPrm, stdPrm, num_class).reshape(num_class,1)
weightsHi = np.random.normal(meanPrm,stdPrm, num_class*pixelWH**2).reshape(num_clas
s,pixelWH**2)

biasLow = np.random.normal(meanPrm, stdPrm, num_class).reshape(num_class,1)
weightsLow = np.random.normal(meanPrm,stdPrm, num_class*pixelWH**2).reshape(num_cla
ss,pixelWH**2)

lrn_rate_Low = 0.0001
lrn_rate_Hi = 0.9

mseListHi = list()
mseListLow = list()

for i in range(num_iter):
    random_index = random.randint(0,train_size-1)

    random_img = train_ims[:,:,random_index].reshape(pixelWH**2,1)
    random_img = random_img/np.amax(random_img)
    random_y = train_one_hot[:,random_index].reshape(num_class,1)

    y_hat = forward(weightsHi, random_img, biasHi)

    y_diff = random_y - y_hat

    w_grd = -2*np.matmul(y_diff*(y_hat)*(1-y_hat),np.transpose(random_img))
    b_grd = 2*y_diff*(y_hat)*(1-y_hat)

    weightsHi -= lrn_rate_Hi*w_grd
    biasHi -= lrn_rate_Hi*b_grd

    mseListHi.append(np.sum(y_diff**2)/(y_diff.shape[0]))

for i in range(num_iter):
    random_index = random.randint(0,train_size-1)

    random_img = train_ims[:,:,random_index].reshape(pixelWH**2,1)
    random_img = random_img/np.amax(random_img)
    random_y = train_one_hot[:,random_index].reshape(num_class,1)

    y_hat = forward(weightsLow, random_img, biasLow)

    y_diff = random_y - y_hat

    w_grd = -2*np.matmul(y_diff*(y_hat)*(1-y_hat),np.transpose(random_img))
    b_grd = 2*y_diff*(y_hat)*(1-y_hat)

    weightsLow -= lrn_rate_Low*w_grd
    biasLow -= lrn_rate_Low*b_grd

    mseListLow.append(np.sum(y_diff**2)/(y_diff.shape[0]))
```

After the training is done, we have plotted the MSE curves using the following code. The plot is also given below.

```
plt.rcParams['figure.figsize'] = [5, 5]
plt.plot(mseListHi)
plt.plot(mseListLow)
plt.plot(mseList)
plt.legend(["MSE for u="+str(lrn_rate_Hi), "MSE for u="+str(lrn_rate_Lo
w), "MSE for u="+str(lrn_rate)])
plt.title("Mean Squared Errors for Different Learning Rates")
plt.xlabel("Iteration Number")
plt.ylabel("MSE")
plt.show()
```

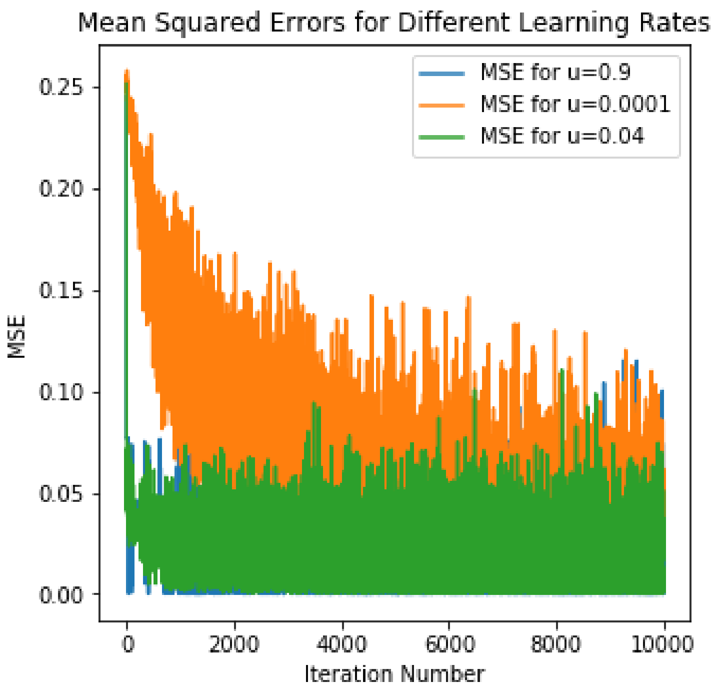

Figure 10 – The MSE Curves for Three Different Learning Rates

Here, we have the MSE curves for three different learning rates. We can observe the curve for 0.0001 learning rate was too slow, but a steady learning curve, however the curve with 0.9 learning rate was staying steady throughout since the learning rate was too high, it had stuck in a local minimum and no matter how many iterations, the MSE wouldn't decrease.

Part D

This final part required us to test the built network on the test data that the algorithm has never seen. In order to put the data through the network, I have used the forward and sigmoid functions that I have defined beforehand. The output is given as,

$$O = \sigma(WX' - B)$$

Where $X' = \frac{X}{\max(X)}$

In order to use the

I have also converted the labels to one hot encoding to be able to compare them with the outputs of the network. The code is given below.

```
test_img = np.load('test_images.npy')
test_labels = np.load('test_labels.npy')

test_img = test_img.reshape(pixelWH**2,test_labels.shape[0])

test_size = test_labels.shape[0]

bias_matrix = np.zeros(num_class*test_size).reshape(num_class, test_size)
for i in range(test_size):
    bias_matrix[:,i] = bias.flatten()

test_img_nor = test_img/np.amax(test_img)
pred = sigmoid(np.matmul(weights,test_img_nor)-bias_matrix)
```

After that, I have written the code to find each network's accuracy. The code below, shows the accuracy of the network trained with a learning rate of 0.04.

```
pred_indices = np.zeros(pred.shape[1])

for i in range(pred.shape[1]):
    pred_indices[i] = np.argmax(pred[:,i])+1

true_cnt = 0
for i in range(pred_indices.shape[0]):
    if(pred_indices[i] == test_labels[i]):
        true_cnt += 1;

print('Accuracy: ', true_cnt/test_labels.shape[0]*100)
```

The output is as follows,

```
Accuracy:  59.30769230769231
```

Below are the code segments that that are used to print the accuracy of the networks trained with learning rates 0.0001 and 0.9 respectively.

```
bias_matrix_low = np.zeros(num_class*test_size).reshape(num_class, test_siz
e)
for i in range(test_size):
    bias_matrix_low[:,i] = biasLow.flatten()

predLow = sigmoid(np.matmul(weightsLow,test_img_nor)-bias_matrix_low)

pred_indices_Low = np.zeros(predLow.shape[1])

for i in range(predLow.shape[1]):
    pred_indices_Low[i] = np.argmax(predLow[:,i])+1

true_cnt_low = 0
for i in range(pred_indices_Low.shape[0]):
    if(pred_indices_Low[i] == test_labels[i]):
        true_cnt_low += 1;

print('Accuracy: ', true_cnt_low/test_labels.shape[0]*100)

Accuracy:  28.000000000000004
```

```
bias_matrix_hi = np.zeros(num_class*test_size).reshape(num_class, test_size
)
for i in range(test_size):
    bias_matrix_hi[:,i] = biasHi.flatten()

predHi = sigmoid(np.matmul(weightsHi,test_img_nor)-bias_matrix_hi)

pred_indices_Hi= np.zeros(predHi.shape[1])

for i in range(predHi.shape[1]):
    pred_indices_Hi[i] = np.argmax(predHi[:,i])+1

true_cnt_hi = 0
for i in range(pred_indices_Hi.shape[0]):
    if(pred_indices_Hi[i] == test_labels[i]):
        true_cnt_hi += 1;

print('Accuracy: ', true_cnt_hi/test_labels.shape[0]*100)
```

```
Accuracy:  25.769230769230766
```

As it can be observed from the outputs, the optimal learning rate reaches for about 59-60% accuracy, the non-optimal learning rates could only reach about 28% and 25% accuracy respectively.