Bilkent University

Department of Electrical & Electronics Engineering

EEE 443 – Neural Networks

Assignment #1

Hasan Emre Erdemoglu

21401462

15/10/2019

TABLE OF CONTENTS

Assignment # 1

Note: In this assignment the preferred coding medium is MATLAB. The entirety of the code can be found in the appendix. Please note that the details in the code implementation is written as comments in the code. Code segments can be also found within the answers to clarify the answers or to show which specific code segment is responsible for generating the output requested by the question.

Question 1:

Note that this problem does not have any MATLAB implementation due to question requirements. Calling this problem in MATLAB will result in a 'No output is available,' displayed in the command window.

The following figure represents the neuron described in the question. It receives input from m neurons with weights w_i where $i \in [1, m]$. After weighted inputs are summed, neuron has an activation function which predicts the output probability of a classes A & B, denoted by +1 and -1 respectively. The specifics of the activation function or the bias term is not present in the question. However due to classes being +1 and -1, the activation function for the neuron would be bi-polar.

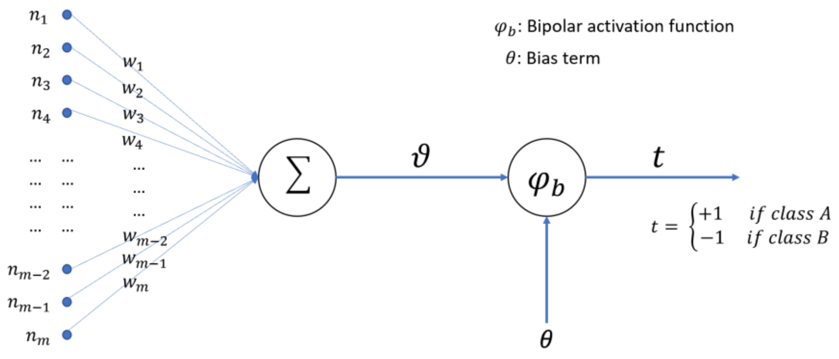


Figure 1. Neuron model for Question 1

Maximum a-posteriori (MAP) estimate for the network weights are given by the following optimization problem:

$$\hat{w}_{MAP} = argmin_w \sum_n \left(y^n - h(x^n, w)\right)^2 + \beta \sum_i w_i^2 \qquad (1)$$

In here, y^n is the ground truth labels of the dataset, for the nth sample; whereas $h(x^n, w)$ is the output of the neuron for the nth sample as shown in Figure 1, which is also denoted by 't'. In broad terms, MAP estimation of the network weights obtained by taking minimum weight parameters set which give rise to the least squared error between ground truth and neuron prediction with addition to a ridge (L2 regularization) term which penalizes high magnitude weights. An optimal solution would center weight coefficients around zero, while minimizing the squared error, namely, the loss, between ground truth and prediction.

In Bayesian statistics, random variables, for the context of this question, network weights, are treated as distributions [1]. These distributions are initially unknown, with possibly exception of a prior belief or probability distribution with very high entropy [2].

Using a set of observed data (x^n, y^n) where x^n is the input and y^n is the ground truth; one may utilize Bayes' rule to derive a posterior distribution. Posterior distribution has reduced entropy for the network weights, as it captures information from likelihood functions using observed data and prior insight of network weights. The Bayes' rule representation of the problem is formulated below:

$$P_{W|Y;X}(w|y;x) = \frac{P_{Y|W;X}(y|w;x) * P_W(w)}{P_Y(y)} \qquad (2)$$

The posterior distribution is given on the left-hand side of Equation (2). The network weights W are conditioned on Y and X, which are the observed output of the neuron and the input to the neuron data respectively.

Here, $P_{Y|W,X}(y|w,x)$ is the likelihood function which outputs probability of our neuron deciding on classes given a set of weights and the input data. $P_W(w)$ represents the prior belief/probability distribution on the network weights and $P_Y(y)$ represents the proportionality constant/evidence. Since $P_Y(y)$ is independent of network weights and as we are focused on network weights, we can simplify equation (2) to following equation:

$$P_{W|Y;X}(w|y;x)\ \alpha\ P_{Y|W;X}(y|w;x) * P_W(w) \quad (3)$$

Note that the Equation (3) can also be read as follows:

$$Posterior\ \alpha\ Likelihood * Prior$$

MAP estimate is a single point estimate on the set of network weights, which picks one optimal set of weights for the network using $argmin_w$ operation. The optimum can be a local optimum as well as a global optimum.

$$\widehat{w}_{MAP} = argmin_w(Posterior)\ \alpha\ argmin_w(Likelihood * Prior)$$

For the given optimization problem, logarithm operation can be applied without altering the result, due to logarithm operation being monotonically increasing. This allows posterior distribution to be written in terms of likelihood distribution with addition of prior distribution.

Continuing from Equation (3):

$$\widehat{w}_{MAP} = argmin_w\left(P_{W|Y;X}(w|y;x)\right) = argmin_w\left(P_{Y|W;X}(y|w;x) * P_W(w)\right)$$

$$\widehat{w}_{MAP} = argmin_w\left(\log\left(P_{W|Y;X}(w|y;x)\right)\right) =$$
$$argmin_w\left(\log\left(P_{Y|W;X}(y|w;x)\right) + \log\left(P_W(w)\right)\right) \quad (4)$$

Let's focus on the likelihood distribution $P_{Y|W;X}(y|w;x)$. Remember that x^n is the input vector from m neurons from nth sample, y^n is the observed ground truth for nth sample and $h(x^n,w)$ is the prediction of the neuron for the nth sample. For 'n' independent and identically distributed samples, the likelihood function is defined by:

$$P_{Y|W;X}(y|w;x) = \prod_{i=1}^{n} P_{Y|W;X}\left(y^i \middle| w;x^i\right)$$

Taking the logarithm of both sides would produce the log-likelihood function:

$$\log\left(P_{Y|W;X}(y|w;x)\right) = \sum\nolimits_{i=1}^{n} \log\left(P_{Y|W;X}\left(y^i \middle| w;x^i\right)\right)$$

Remember Equation (1), which was the original optimization problem for network weights:

$$\widehat{w}_{MAP} = argmin_w \left(\sum\nolimits_n \left(y^n - h(x^n,w)\right)^2 + \beta \sum\nolimits_i w_i^2\right) \qquad (1)$$

Using this definition and Equation (1) and (4) the following two relations can be observed, as both equations are encapsulated with $argmin_w$ operation:

- $\log\left(P_{Y|W;X}(y|w;x)\right) = \sum_{i=1}^{n} \log\left(P_{Y|W;X}\left(y^i \middle| w;x^i\right)\right) = \sum_n \left(y^n - h(x^n,w)\right)^2$ (5)
- $\log\left(P_W(w)\right) = \beta \sum_i w_i^2$ (6)

Note that Equation (5) was found similar due to summing of n samples in both equations. They were of the same form. This also implies that optimization of the log-likelihood function is the same as optimization of mean squared error between ground truth (y^n) and prediction ($h(x^n,w)$) using input data and network weights. This is reasonable, as the difference between ground truth and prediction is in line with what likelihood function is used for: estimating parameters of a model where observed data is most probable to the described model.

Also note that, the likelihood function has dependencies on Y, W and X. $\log\left(P_W(w)\right) \neq \sum_n \left(y^n - h(x^n,w)\right)^2$, as prior distribution cannot have dependencies to Y and X. Therefore the mapping at Equations (5) and (6) are correct.

Equation (6) dictates that log-prior distribution is the L2 normalization term given in the initial optimization problem. In order to find the prior probability distribution of the weights, some algebraic manipulations are needed. Getting rid of the logarithm in Equation (6) gives:

$$P_W(w) = 1 * e^{\beta \sum_i w_i^2}$$

This implies the prior probability distribution of the network weights is a Gaussian with $\mu = 0$, $\sigma = -\frac{1}{\sqrt{2\beta}}$.

Question 2:

The question is being asked for the logic circuit given as:

$$(X_1\ OR\ NOT\ X_2)\ XOR\ (NOT\ X_3\ OR\ NOT\ X_4) \quad \text{(C1)}$$

Additionally, specifications on the neural network that realizes this logic circuit is given as follows:

- Single hidden layer, 4 input neurons
- The single hidden layer has 4 hidden units; $n_{1,1}, n_{1,2}, n_{1,3}, n_{1,4}$
- Single output neuron; n_o
- Activation functions are unipolar, step functions
- Input is given by x_1, x_2, x_3, x_4

A visual representation of the neural network is given below. Note that the weights are drawn to be fully connected. This does not imply that all weights are non-zero, it is just drawn for the sake of completeness. Arrow operator implicitly multiplies input neuron or hidden layer output with the weight. In the figure, every neuron, sums up weighted inputs and passes the step activation to generate output automatically. θ_{hl} and θ_o are the bias terms for the hidden layer and output layer respectively. θ_{hl} applies for all neurons of the hidden layer as given in Figure 2.

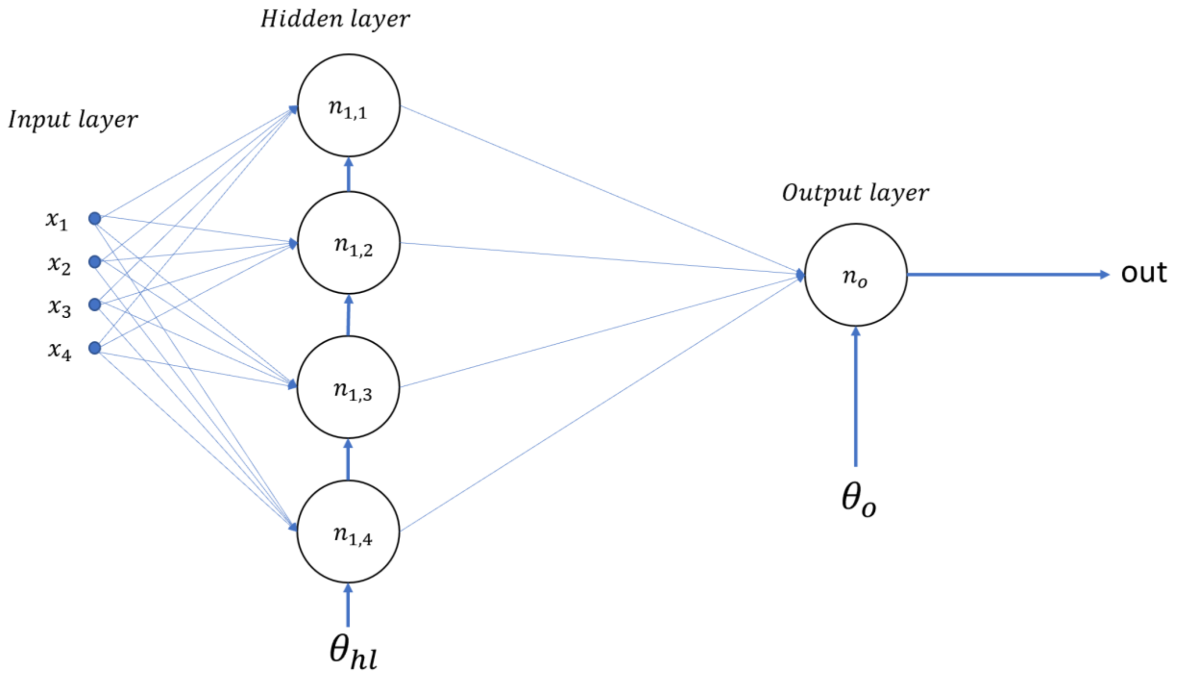


Figure 2. Neural Network in Problem Description

Part a:

For visual representation, logic gate configuration for (C1) is given in the figure below:

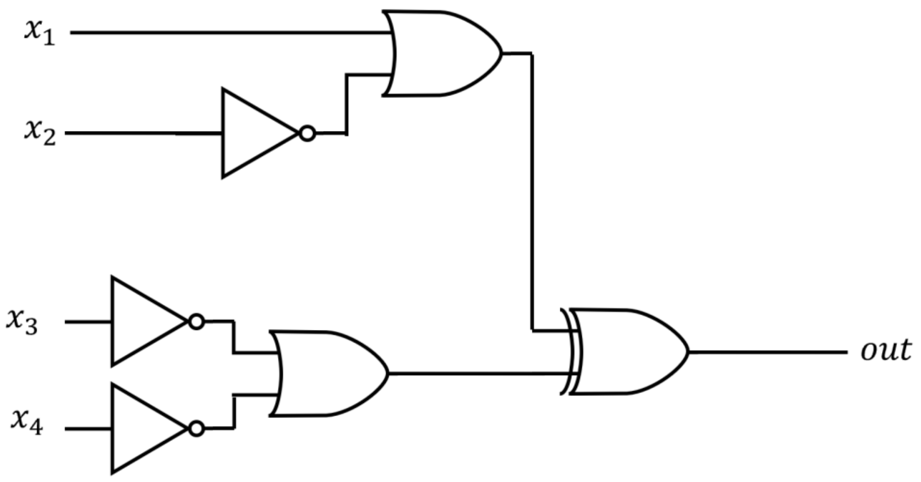
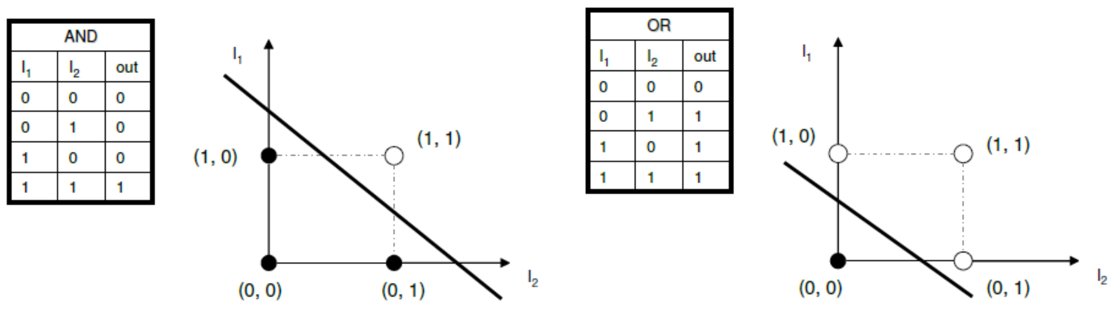


Figure 3. Logic Circuit Diagram for C1

Since XOR operation is not linearly separable using only one line; it cannot be described readily by a single neuron. It requires at least 2 neurons to realize XOR operation. Please refer to the figures below for one neuron and multi-neuron linear separability.



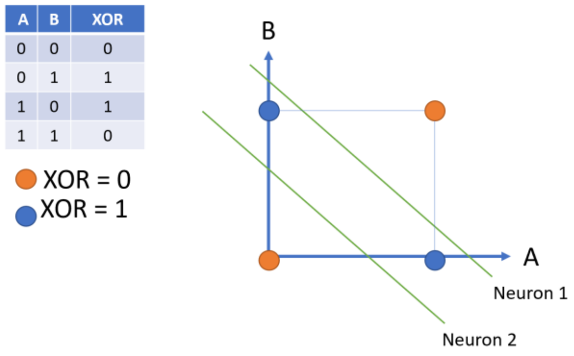
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1

OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4. Linear Decision Boundaries for AND & OR Logic Gates *[3]*

As seen in Figure 4, linear separability requires classes, in this case 0 and 1, to be grouped in such a way that it can be separated using a line. For 2D problems, a line; for multi-dimensional problems a hyperplane can separate the input space into two regions. AND, OR and NOT logic gates are linearly, as explained in the lecture notes and as proved in Figure 4.

XOR operation is not linearly separable. It requires combination of at least two neurons to realize such logic operation. Geometrical reason is described in the figure below:



A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5. Linear Decision Boundaries for XOR gate

As seen in Figure 5, when the input-output relations are mapped in 2D plane, there is no such line that separates orange and blue classes from each other. At least combination of 2 lines are required to divide the output classes from each other. Please note that Figure 5 is does not show the unique solution for the XOR problem. Infinitely many weights and biases can be assigned to inputs A & B, constrained with the geometrical shape above. Additionally, one may implement two neurons in y=x direction, like the shape above and may assign infinitely many weights and biases while conforming to the geometrical constraints.

Since the question constraints neural realization of logic circuit (C1) with a single 4-neuron hidden layer and an output layer, and XOR logic gate needs at least two layers, one should convert logic circuit C1 to Sums of Products (SOP) form using linearly separable AND, OR and NOT gates.

First, lets derive XOR in SOP using the sample circuit:

$$A\ XOR\ B \quad (C2)$$

The truth table and Karnaugh map for this operation is given in the figure below:

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

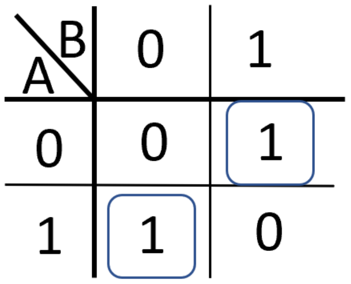


Figure 6. Truth Table & K-map for XOR operation *[4]*

Using the Karnaugh map, SOP representation of the XOR logic gate becomes:

$$A\ XOR\ B = (A\ AND\ NOT\ B)\ OR\ (NOT\ A\ AND\ B)$$

For ease of understanding, remember C1:

$$(X_1\ OR\ NOT\ X_2)\ XOR\ (NOT\ X_3\ OR\ NOT\ X_4) \quad \text{(C1)}$$

Now put $A = (X_1\ OR\ NOT\ X_2)$ and $B = (NOT\ X_3\ OR\ NOT\ X_4)$ to logic circuit (C3).

The equation becomes the following. Let's use (+) for OR, (*) sign for AND operations & bar notation for NOT operation for simplicity:

$$\left((X_1 + \bar{X}_2) * \overline{(\overline{X_3} + \overline{X_4})}\right) + \left(\overline{(X_1 + \overline{X_2})} * (\overline{X_3} + \overline{X_4})\right)$$

Using De Morgan's laws and Boolean logic, simplify the equation above:

$$\left((X_1 + \bar{X}_2) * (X_3 * X_4)\right) + \left((\overline{X_1} * X_2) * (\overline{X_3} + \overline{X_4})\right)$$

Distributing AND multiplication yields:

$$(X_1 * X_3 * X_4) + (\bar{X}_2 * X_3 * X_4) + (\overline{X_1} * X_2 * \overline{X_3}) + (\overline{X_1} * X_2 * \overline{X_4}) \quad \text{(C4)}$$

Logic circuit (C4) becomes SOP representation of the given logic circuit, (C1). Returning to Figure 2, each neuron in the hidden layer can represent one of the products in the logic circuit (C4) and the summation of these neurons can be represented by the output neuron. Therefore, hidden layer will use the AND operations to determine the regions, whereas output layer combine regions to produce output +1 or 0 given the inputs. Assume the following mapping from the logic circuit to the neural network from now on; as depending on how we assign product terms to the neurons, network weights can change:

- $n_{1,1}$ is responsible for $(X_1 * X_3 * X_4)$
- $n_{1,2}$ is responsible for $(\bar{X}_2 * X_3 * X_4)$
- $n_{1,3}$ is responsible for $(\overline{X_1} * X_2 * \overline{X_3})$
- $n_{1,4}$ is responsible for $(\overline{X_1} * X_2 * \overline{X_4})$
- n_o is responsible for summing up all weighted hidden layer neurons and applying the step function.

Let's derive the set of inequalities for each hidden unit, as asked in Part a. Remember Figure 1 from Question 1. Each neuron first multiplies its input by weights and then linearly combines them. After that, it processes this linear combination of inputs with an activation function after adding a bias.

Let w_{ij} be the weights for the inputs for all hidden units here "i" is for index of the input and "j" is the index of the neuron.

For $n_{1,1}$ output of the neuron is given below with the logic table. The set of inequalities are given in bold.

Table 1. Logic table for $x_1x_3x_4$

Ineq. #	$\boldsymbol{x_1}$	$\boldsymbol{x_3}$	$\boldsymbol{x_4}$	$\boldsymbol{out_{1,1}}$
1	0	0	0	**0**
2	0	0	1	**0**
3	0	1	0	**0**
4	0	1	1	**0**
5	1	0	0	**0**
6	1	0	1	**0**
7	1	1	0	**0**
8	1	1	1	**1**

$$out_{1,1} = step(w_{1,1}x_1 + w_{3,1}x_3 + w_{4,1}x_4 - \theta)$$

SET OF INEQUALITIES:

1. $0 = step(-\theta) \rightarrow -\theta < 0 \rightarrow \boldsymbol{\theta > 0}$
2. $0 = step(w_{4,1} - \theta) \rightarrow w_{4,1} - \theta < 0 \rightarrow \boldsymbol{w_{4,1} < \theta}$
3. $0 = step(w_{3,1} - \theta) \rightarrow w_{3,1} - \theta < 0 \rightarrow \boldsymbol{w_{3,1} < \theta}$
4. $0 = step(w_{3,1} + w_{4,1} - \theta) \rightarrow w_{3,1} + w_{4,1} - \theta < 0 \rightarrow \boldsymbol{w_{3,1} + w_{4,1} < \theta}$
5. $0 = step(w_{1,1} - \theta) \rightarrow w_{1,1} - \theta < 0 \rightarrow \boldsymbol{w_{1,1} < \theta}$
6. $0 = step(w_{1,1} + w_{4,1} - \theta) \rightarrow w_{1,1} + w_{4,1} - \theta < 0 \rightarrow \boldsymbol{w_{1,1} + w_{4,1} < \theta}$
7. $0 = step(w_{1,1} + w_{3,1} - \theta) \rightarrow w_{1,1} + w_{3,1} - \theta < 0 \rightarrow \boldsymbol{w_{1,1} + w_{3,1} < \theta}$
8. $0 = step(w_{1,1} + w_{3,1} + w_{4,1} - \theta) \rightarrow w_{1,1} + w_{3,1} + w_{4,1} - \theta < 0 \rightarrow \boldsymbol{w_{1,1} + w_{3,1} + w_{4,1} < \theta}$

One may find the summary of the set of inequalities below:

$\boldsymbol{\theta > 0}$	$\boldsymbol{w_{1,1} + w_{3,1} < \theta}$	$\boldsymbol{w_{1,1} + w_{3,1} + w_{4,1} < \theta}$
$\boldsymbol{w_{1,1} < \theta}$	$\boldsymbol{w_{1,1} + w_{4,1} < \theta}$	
$\boldsymbol{w_{3,1} < \theta}$	$\boldsymbol{w_{3,1} + w_{4,1} < \theta}$	
$\boldsymbol{w_{4,1} < \theta}$		

For $n_{1,2}$ output of the neuron is given below with the logic table. The set of inequalities are given in bold. Note that due to $\overline{x_2}$, whenever we encounter $\overline{x_2} = 1$, the weight is set to be negative, due to nature of NOT operation [3].

$$out_{1,2} = step(w_{2,2}\overline{x_2} + w_{3,2}x_3 + w_{4,2}x_4 - \theta)$$

Table 2. Logic table for $\overline{x_2}x_3x_4$

Ineq. #	$\boldsymbol{x_2}$	$\boldsymbol{\overline{x_2}}$	$\boldsymbol{x_3}$	$\boldsymbol{x_4}$	$\boldsymbol{out_{1,2}}$
1	0	1	0	0	**0**
2	0	1	0	1	**0**
3	0	1	1	0	**0**
4	0	1	1	1	**1**
5	1	0	0	0	**0**
6	1	0	0	1	**0**
7	1	0	1	0	**0**
8	1	0	1	1	**0**

<u>SET OF INEQUALITIES:</u>

1. $0 = step(-w_{2,2} - \theta) \rightarrow -w_{2,2} - \theta < 0 \rightarrow$ $\boldsymbol{w_{2,2} > -\theta}$
2. $0 = step(-w_{2,2} + w_{4,2} - \theta) \rightarrow -w_{2,2} + w_{4,2} - \theta < 0 \rightarrow$ $\boldsymbol{-w_{2,2} + w_{4,2} < \theta}$
3. $0 = step(-w_{2,2} + w_{3,2} - \theta) \rightarrow -w_{2,2} + w_{3,2} - \theta < 0 \rightarrow \boldsymbol{-w_{2,2} + w_{3,2} < \theta}$
4. $1 = step(-w_{2,2} + w_{3,2} + w_{4,2} - \theta) \rightarrow -w_{2,2} + w_{3,2} + w_{4,2} - \theta > 0 \rightarrow$ $\boldsymbol{-w_{2,2} + w_{3,2} + w_{4,2} > \theta}$
5. $0 = step(-\theta) \rightarrow -\theta < 0 \rightarrow \boldsymbol{\theta > 0}$
6. $0 = step(w_{4,2} - \theta) \rightarrow w_{4,2} - \theta < 0 \rightarrow \boldsymbol{w_{4,2} < \theta}$
7. $0 = step(w_{3,2} - \theta) \rightarrow w_{3,2} - \theta < 0 \rightarrow \boldsymbol{w_{3,2} < \theta}$
8. $0 = step(w_{3,2} + w_{4,2} - \theta) \rightarrow w_{3,2} + w_{4,2} - \theta < 0 \rightarrow$ $\boldsymbol{w_{3,2} + w_{4,2} < \theta}$

One may find the summary of the set of inequalities below:

$\boldsymbol{\theta > 0}$	$\boldsymbol{-w_{2,2} + w_{3,2} < \theta}$	$\boldsymbol{-w_{2,2} + w_{3,2} + w_{4,2} > \theta}$
$\boldsymbol{w_{2,2} > -\theta}$	$\boldsymbol{-w_{2,2} + w_{4,2} < \theta}$	
$\boldsymbol{w_{3,2} < \theta}$ $\boldsymbol{w_{4,2} < \theta}$	$\boldsymbol{w_{3,2} + w_{4,2} < \theta}$	

For $n_{1,3}$ output of the neuron is given below with the logic table. The set of inequalities are given in bold. Note that due to $\overline{x_1}$ and $\overline{x_3}$, whenever we encounter $\overline{x_1} = 1$ or $\overline{x_3} = 1$ the particular weight is set to be negative, due to nature of NOT operation.

Table 3. Logic table for $\overline{x_1}x_2\overline{x_3}$

Ineq. #	$\boldsymbol{x_1}$	$\boldsymbol{x_3}$	$\boldsymbol{\overline{x_1}}$	$\boldsymbol{x_2}$	$\boldsymbol{\overline{x_3}}$	$\boldsymbol{out_{1,3}}$
1	0	0	1	0	1	**0**
2	0	0	1	1	1	**1**
3	0	1	1	0	0	**0**
4	0	1	1	1	0	**0**
5	1	0	0	0	1	**0**
6	1	0	0	1	1	**0**
7	1	1	0	0	0	**0**
8	1	1	0	1	0	**0**

$$out_{1,3} = step(w_{1,3}\overline{x_1} + w_{2,3}x_2 + w_{3,3}\overline{x_3} - \theta)$$

<u>SET OF INEQUALITIES:</u>

1. $0 = step(-w_{1,3} - w_{3,3} - \theta) \rightarrow -w_{1,3} - w_{3,3} - \theta < 0 \rightarrow \boldsymbol{-w_{1,3} - w_{3,3} < \theta}$
2. $1 = step(-w_{1,3} + w_{2,3} - w_{3,3} - \theta) \rightarrow -w_{1,3} + w_{2,3} - w_{3,3} - \theta > 0 \rightarrow \boldsymbol{-w_{1,3} + w_{2,3} - w_{3,3} > \theta}$
3. $0 = step(-w_{1,3} - \theta) \rightarrow -w_{1,3} - \theta < 0 \rightarrow \boldsymbol{w_{1,3} > -\theta}$
4. $0 = step(-w_{1,3} + w_{2,3} - \theta) \rightarrow -w_{1,3} + w_{2,3} - \theta < 0 \rightarrow \boldsymbol{-w_{1,3} + w_{2,3} < \theta}$
5. $0 = step(-w_{3,3} - \theta) \rightarrow -w_{3,3} - \theta < 0 \rightarrow \boldsymbol{w_{3,3} > -\theta}$
6. $0 = step(w_{2,3} - w_{3,3} - \theta) \rightarrow w_{2,3} - w_{3,3} - \theta < 0 \rightarrow \boldsymbol{w_{2,3} - w_{3,3} < \theta}$
7. $0 = step(-\theta) \rightarrow -\theta < 0 \rightarrow \boldsymbol{\theta > 0}$
8. $0 = step(w_{2,3} - \theta) \rightarrow w_{2,3} - \theta < 0 \rightarrow \boldsymbol{w_{2,3} < \theta}$

One may find the summary of the set of inequalities below:

$$\boldsymbol{\theta > 0} \qquad \boldsymbol{-w_{1,3} + w_{2,3} < \theta} \qquad \boldsymbol{-w_{1,3} + w_{2,3} - w_{3,3} > \theta}$$

$$\boldsymbol{w_{1,3} > -\theta} \qquad \boldsymbol{-w_{1,3} - w_{3,3} < \theta}$$

$$\boldsymbol{w_{2,3} < \theta} \qquad \boldsymbol{w_{2,3} - w_{3,3} < \theta}$$

$$\boldsymbol{w_{3,3} > -\theta}$$

For $n_{1,4}$ output of the neuron is given below with the logic table. The set of inequalities are given in bold. Note that due to $\overline{x_1}$ and $\overline{x_4}$, whenever we encounter $\overline{x_1} = 1$ or $\overline{x_4} = 1$ the particular weight is set to be negative, due to nature of NOT operation.

$$out_{1,4} = step(w_{1,4}\overline{x_1} + w_{2,4}x_2 + w_{4,4}\overline{x_4} - \theta)$$

Table 4. Logic table for $\overline{x_1}x_2\overline{x_4}$

Ineq. #	$\boldsymbol{x_1}$	$\boldsymbol{x_4}$	$\boldsymbol{\overline{x_1}}$	$\boldsymbol{x_2}$	$\boldsymbol{\overline{x_4}}$	$\boldsymbol{out_{1,4}}$
1	0	0	1	0	1	**0**
2	0	0	1	1	1	**1**
3	0	1	1	0	0	**0**
4	0	1	1	1	0	**0**
5	1	0	0	0	1	**0**
6	1	0	0	1	1	**0**
7	1	1	0	0	0	**0**
8	1	1	0	1	0	**0**

<u>SET OF INEQUALITIES:</u>

1. $0 = step(-w_{1,4} - w_{4,4} - \theta) \rightarrow -w_{1,4} - w_{4,4} - \theta < 0 \rightarrow \boldsymbol{-w_{1,4} - w_{4,4} < \theta}$
2. $1 = step(-w_{1,4} + w_{2,4} - w_{4,4} - \theta) \rightarrow -w_{1,4} + w_{2,4} - w_{4,4} - \theta > 0 \rightarrow \boldsymbol{-w_{1,4} + w_{2,4} - w_{4,4} > \theta}$
3. $0 = step(-w_{1,4} - \theta) \rightarrow -w_{1,4} - \theta < 0 \rightarrow \boldsymbol{w_{1,4} > -\theta}$
4. $0 = step(-w_{1,4} + w_{2,4} - \theta) \rightarrow -w_{1,4} + w_{2,4} - \theta < 0 \rightarrow \boldsymbol{-w_{1,4} + w_{2,4} < \theta}$
5. $0 = step(-w_{4,4} - \theta) \rightarrow -w_{4,4} - \theta < 0 \rightarrow \boldsymbol{w_{4,4} > -\theta}$
6. $0 = step(w_{2,4} - w_{4,4} - \theta) \rightarrow w_{2,4} - w_{4,4} - \theta < 0 \rightarrow \boldsymbol{w_{2,4} - w_{4,4} < \theta}$
7. $0 = step(-\theta) \rightarrow -\theta < 0 \rightarrow \boldsymbol{\theta > 0}$
8. $0 = step(w_{2,4} - \theta) \rightarrow w_{2,4} - \theta < 0 \rightarrow \boldsymbol{w_{2,4} < \theta}$

One may find the summary of the set of inequalities below:

$$\boldsymbol{\theta > 0} \qquad \boldsymbol{-w_{1,4} + w_{2,4} < \theta} \qquad \boldsymbol{-w_{1,4} + w_{2,4} - w_{4,4} > \theta}$$

$$\boldsymbol{w_{1,4} > -\theta} \qquad \boldsymbol{-w_{1,4} - w_{4,4} < \theta}$$

$$\boldsymbol{w_{2,4} < \theta} \qquad \boldsymbol{w_{2,4} - w_{4,4} < \theta}$$

$$\boldsymbol{w_{4,4} > -\theta}$$

Part b:

In Figure 2, just for a basic representation inputs were fully connected to the hidden layer. Given the logic circuit (C1) is realized with neural network presented in Figure 2, and the set of inequalities for the weights and the biases are realized; Figure 2 can be specialized to the following:

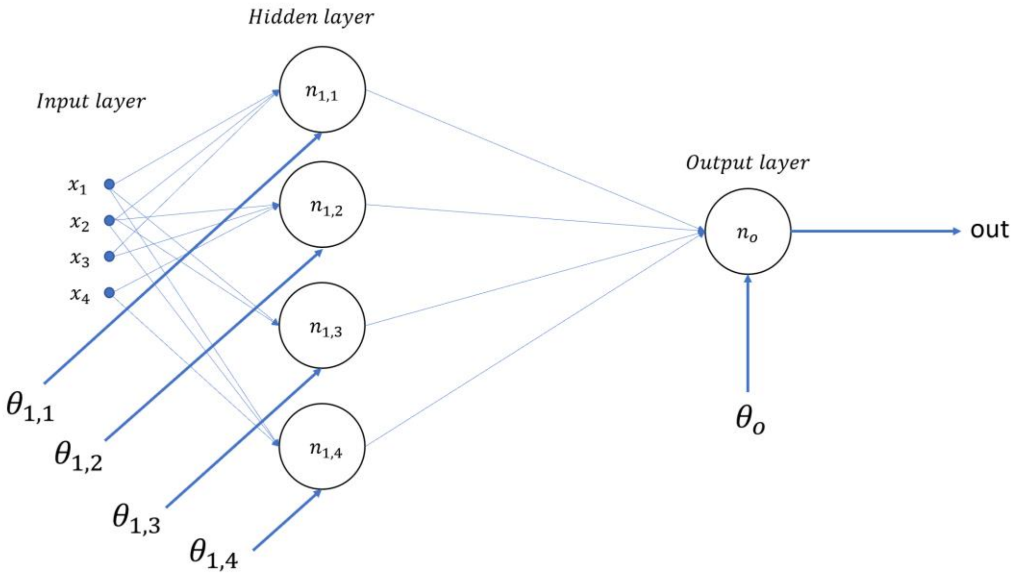


Figure 7. Specialized Neural Network Model for Question 2

Note that I have omitted the arrows that corresponding to the weights which are zero given set of inequalities for the hidden units found in part a. However, later in the report, I will assume input layer is fully connected to the hidden layer to create matrix notation for the weights and biases. The representation above is given for clarification purposes only.

The realized logic circuit was defined by (C4) and it was equivalent to initial logic circuit (C1):

$$(X_1 * X_3 * X_4) + (\bar{X}_2 * X_3 * X_4) + (\overline{X_1} * X_2 * \overline{X_3}) + (\overline{X_1} * X_2 * \overline{X_4}) \quad \text{(C4)}$$

Note that this the output of the logic circuit. Output layer must output 1, if any of the products happen to return as 1; output is 0 otherwise; as suggested by the property of OR operation. For the output neuron we may have the following relation:

$$out = step\left(w_{n1,1}n_{1,1} + w_{n1,2}n_{1,2} + w_{n1,3}n_{1,3} + w_{n1,4}n_{1,4} - \theta_o \right)$$

As we use step as our activation function; the neuron outputs can only become 0 or 1. I assume that there is no noise present at the outputs of any neurons. This ensures that the corresponding weight either becomes zero, implying hidden unit did not fire, or equal to the weight, implying that the hidden unit has fired. Given that the function above is 4 input OR operation, if $w_{n1,1}n_{1,1} + w_{n1,2}n_{1,2} + w_{n1,3}n_{1,3} + w_{n1,4}n_{1,4} - \theta_o \geq 0$ this means the output of the entire logic circuit becomes 1. In that case, Therefore, $w_{n1,1}$, $w_{n1,2}$, $w_{n1,3}$, $w_{n1,4}$ can be chosen as 1 and θ_o can also be selected as 1. If none of the hidden units fire then, -1 bias will force output neuron to not fire, outputting a zero. Therefore, given that hidden units feeding noise-free and correct inputs to the output layer, output layer will achieve 100 % performance, nonetheless.

The extended weight vector and extended input for the output layer respectively would be:

$$[W_o \mid \theta_o] = [1\ 1\ 1\ 1\ |1]$$

$$[n_1|-1] = \left[n_{1,1}\ n_{1,2}\ n_{1,3}\ n_{1,4}\ \middle| -1\right]$$

As for the weight vector for the input layer, the set of inequalities for the hidden units have infinitely many solutions given the constraints are satisfied. I have selected following arbitrary weights as one solution for this question. As activation function is step function, given the weighted sum with the bias becomes positive within the activation, the neuron outputs 1, otherwise 0. For matrix representation, I will follow convention of Figure 2, where input layer is fully connected to the hidden layer 1. I will also use extended notation where bias is also written along the weight terms [3]. Note that weight matrix is transposed, due to previous notation.

$$[W^T|\theta] = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & \theta_{1,1} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & \theta_{1,2} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & \theta_{1,3} \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & \theta_{1,4} \end{bmatrix} = \begin{bmatrix} 2 & 0 & 1.5 & 2 & 5 \\ 0 & -3 & 3 & 4 & 9 \\ -2 & 8 & -3 & 0 & 12 \\ -2 & 8 & 0 & -3 & 12 \end{bmatrix}$$

$$[X|\theta] = [x_1\ x_2\ x_3\ x_4|-1]^T$$

One may put the weights to their corresponding set of inequalities and will see that, indeed, all 8 inequalities are satisfied. Since all inequalities are found with assumption of inputs of the hidden layer (input layer neurons) being binary (either 0 or 1) on part a, satisfying all 32 inequalities for the hidden units prove that the entire hidden layer achieves 100 % classification performance. One may put given weights to their respective places in the inequalities to verify it indeed works.

Shortly, since the hidden units realize AND gates, sum of all weights should be less than the bias for all hidden units would be a necessary and enough condition to apply in order to get 100 % performance on the neural network. This occurs because for the AND gate to fire, all input neurons must output 1. In this case weights are directly translated inside the non-linear activation. Since step function outputs +1 if the sum of weights + bias is greater than equal to zero; such condition is enough to derive a solution.

As hidden layer perfectly realizes the product terms and output layer perfectly realizes sums of products the designed network will achieve 100 % performance. Of course, this is achieved assuming that the input is perfectly binary and there is no noise in input data, the hidden layer or the output layer outputs. Any noise in the inputs will be amplified in the activation signal and may perturb the output of the neuron if the weights are large enough. Of course, bias term can be adjusted to make up for the large weights; yet combination of these large weights may make the output of the activation signal more varying so that the bias term is not strong enough to separate classes. This will be investigated in part c.

Another brute force proof of performance includes creating all 16 input combinations and using hidden layer and output layer weights and biases realizing the logic circuit described in (C1). If the neural network provides the same logic table with (C1), then it is proved that neural network achieves 100 % performance. Since this will be done in part d, using MATLAB, the derivation will be skipped here.

Part c:

The hidden unit inequalities and the weights which were selected in part b does not accommodate for input noise, it was assumed that the input can be either 1 or 0. Therefore, it will not work robustly under noisy conditions. Any input noise present in the inputs gets amplified by the neuron weights. Bias must be adjusted carefully to accommodate the variability of the weighted error that is presented by the noise which is added on top of the inputs. Large weights can also increase the variability of the activation signal due to the noise.

Without noise, if an input exists, the weighted sum would have the weight with input coefficient 1, imposing the weight without a scaling factor. If an input does not exist, the corresponding weight would not be seen on the weighted sum of the neuron.

However, if the input values are not equal to 0 or 1, the input may scale the weight, change its sign, if the error is extreme. Consider following arbitrary examples:

- $w = 10, x = 1 - 0.2 = 0.8$. The weighted term inside activation becomes: $wx = 8$. This causes activation to be less active. If other weighted terms output < -8, the neuron does not fire, where it should fire.
- $w = 10, x = 0 - 0.1 = -0.1$. he weighted term inside activation becomes: $wx = -1$. This reduces the activation potential of the neuron by 1, where such term should not exist after all. Given other weighted sums in the activation, it may cause neuron to misfire or not fire at all.

These were two error cases: many different error cases can be identified as well. If hidden layer does not work in 100 % performance, output layer carries on these mistakes, even though it is assumed no noise is present in outputs of any of the neurons. Therefore, by using a counterexample it is proven that small random fluctuations may hurt realizability of the logic circuit (C1), reducing the performance of the neural network.

One way to create a robust decision boundary is to implement support vectors to the decision boundaries. Since entire exhaustive state space outputs are available to us and since it is assumed that we have small fluctuations in the data, rather than noises that overlap two classes together, this implementation could be useful.

I will be working on neuron 1 to generate support vector for the decision boundaries for the sake of giving an example. Keep in mind that since all four hidden units does the same AND operation, using different inputs, the derivation for the decision boundary is the same. I will be using 2D AND operation decision boundaries and then expand upon 3D case, since all hidden layers are connected to 3 inputs given logic circuit constraints. Note that 2D case will merely be a projection of 3D case where I'll be working on 1st quadrant; rather than 1st octant.

Remember neuron 1’s activation signal:

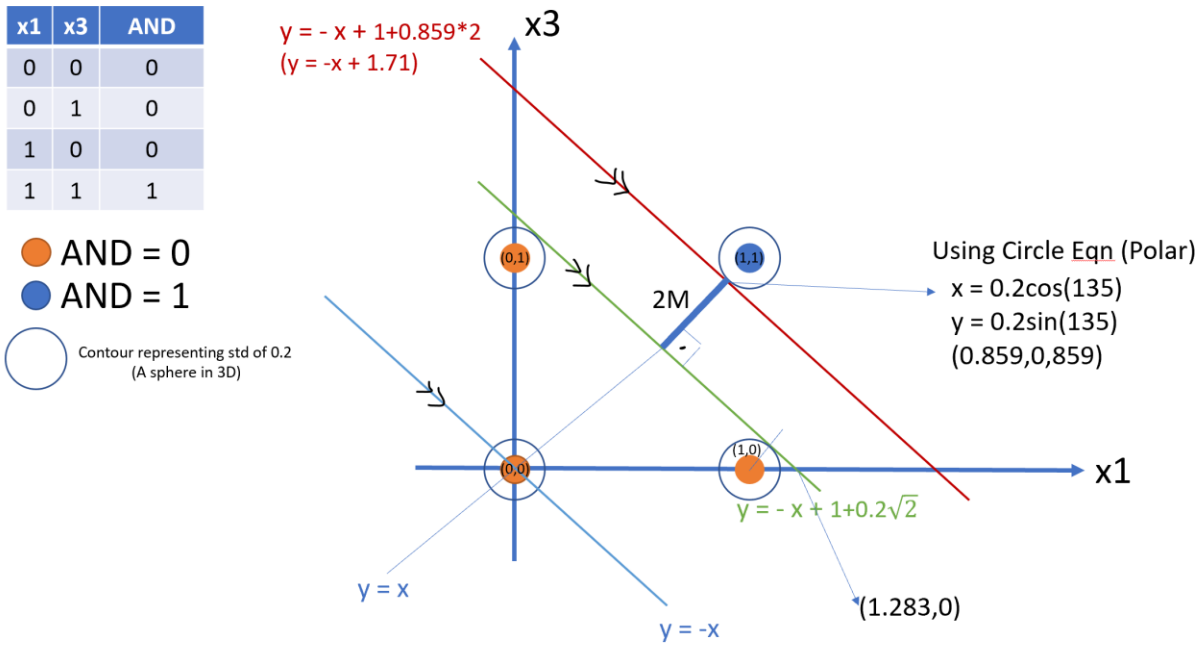
$$v = w_1 x_1 + w_3 x_3 + w_4 x_4 - \theta$$

Remember that the non-linear activation function is step function, so given activation signal is greater than 0, the neuron will output 1, otherwise zero.

Now, since we are assuming an additive Gaussian noise for all inputs with mean 0 and standard deviation 0.2, we can write the activation signal as follows:

$$v = w_1 x_1 + w_3 x_3 + w_4 x_4 + w_1 \varepsilon_1 + w_3 \varepsilon_3 + w_4 \varepsilon_4 - \theta$$

Now let’s look at the decision boundary for AND operation and try to come up with an analytic solution. A sketch in 2D is given below for x_1 and x_3; the same will also apply for x_1-x_4 and x_3-x_4. At the end, 3D will be the projections on these three planes, as the data points sit in corner of 1st Quadrant/1st Octant.



x1	x3	AND
0	0	0
0	1	0
1	0	0
1	1	1

Figure 8. Sketch of Decision Boundaries for Neuron 1 in 2D

First by observation it is decided that the decision boundary will pass from x_3=-x_1 as noise is the same for all inputs shown in the Figure and as the outputs sit at the corners of the Boolean input space.

Then, using the standard deviation contour, with radius 0.2, offset for the green support vector is found. Here we create a right isosceles triangle shown in the bottom right data point. Using a point found on the line and the slope of the decision line as -1 we can set up the lower bound boundary.

For the red (upper support vector), I have use polar equation to extract the coordinate for the upper support vector. Since I know that (0,0) and (1,1) data points sit on the graph it was easy to find.

Ideally, we want the decision boundary to divide the classes with widest margins possible at both sides. So, the line segment drawn in bold blue will be divided into two separate segment and the midpoint will adjust the offset of x_3=-x_1 line.

With some algebra the line segment has two points as (0.64,0.64) and (0.859,0.859). Since we are on x_3=x_1 line, we have the midpoint as: (0.7495,0.7495) which gives rise to the decision boundary as: x_3 = -x_1 + 1.49.

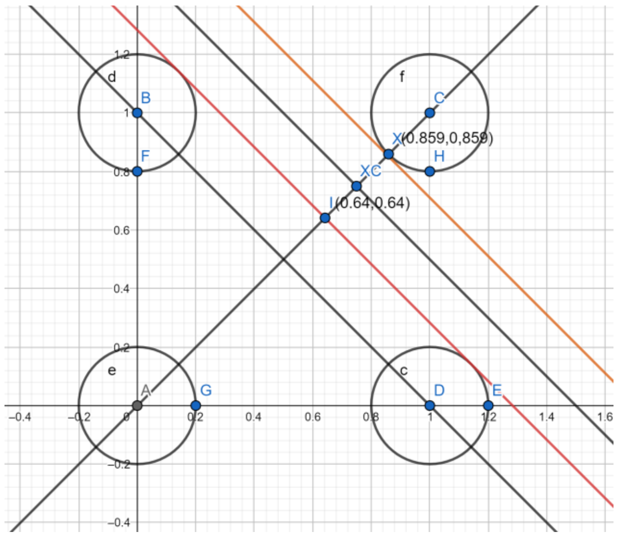


Figure 9. Computer Sketch of the Boundary Selection

Since the importance of the inputs are equal, there is no scaling happening on any of the 2D planes, therefore this boundary will also be retained in 3D. Red equation $x_3 = -x_1 + 0.2\sqrt{2}$ is particularly important. It is the decision boundary with bias $0.2\sqrt{2} = 0.283$, which for 0 outputs, it may tolerate noises up to 1 standard deviation up. This also means there will be no error in classifying class 1 as it the decision boundary is more than 2 standard deviations away.

Therefore, for input biases choosing a of 0.283 should reduce the effects of the noise and increase the overall performance. For robust weights I propose the following matrix, with input matrix and output layer weights being the same. Note that 1 is chosen to be the scaling factor for 2D and 3D cases, therefore the weight factors for the inputs are equalized:

$$[W^T|\theta] = \begin{bmatrix} 1 & 0 & 1 & 1 & 2.717 \\ 0 & -1 & 1 & 1 & 2.717 \\ -1 & 1 & -1 & 0 & 2.717 \\ -1 & 1 & 0 & -1 & 2.717 \end{bmatrix}$$

Please note that all neurons get 3 inputs and they all do AND operations, so the equations are the same. For inputs that are inverted, we have negative weights as usual. Also keep in mind that we have to set the activation signal to 0 if we want to have an output 1 from the hidden unit. Since AND operation requires all neurons to be active, the noise free bias should be set to 3 and this noise bias should be removed from the equation.

Part d:

This question implements the weights found in parts b and c to show the performance of the weights that are found. The code has explanations within, one may check the comments on the code to further understand how the network was implemented.

The code first builds the logic table for all quadruple inputs x, then realizes the logic table for the individual neurons. For the logic table developed, logic circuit (C1) was solved by hand exhaustively and the outputs for the logic circuit was developed. The logic table for the entire system is shown below:

Table 5. Extended Logic Table for XOR operation

These are OR'ed					
(x_1,x_2,x_3,x_4)	$(x_1*x_3*x_4)$	$(nx_2*x_3*nx_4)$	$(nx_1*x_2*nx_3)$	$(nx_1*x_2*nx_4)$	**out**
0000	**000**	**100**	**101**	**101**	**0**
0001	**001**	**101**	**101**	**100**	**0**
0010	**010**	**110**	**100**	**101**	**0**
0011	**011**	**111**	**100**	**101**	**1**
0100	**000**	**000**	**111**	**111**	**1**
0101	**001**	**011**	**111**	**100**	**1**
0110	**010**	**010**	**110**	**111**	**1**
0111	**011**	**011**	**110**	**110**	**0**
1000	**100**	**100**	**001**	**001**	**0**
1001	**101**	**101**	**001**	**000**	**0**
1010	**110**	**110**	**000**	**001**	**0**
1011	**111**	**111**	**000**	**000**	**1**
1100	**100**	**000**	**011**	**011**	**0**
1101	**101**	**001**	**011**	**010**	**0**
1110	**110**	**010**	**010**	**011**	**0**
1111	**111**	**011**	**010**	**010**	**1**

After the following table was hard coded in MATLAB, Gaussian noise was generated, it was added on top of the samples. A non-noise version of the 400 samples are kept in order to benchmark the robustness of the weights on parts b and c. Output layers for both networks are kept the same to isolate the behavior of the hidden layer.

This function realizes the networks described with hidden layer and output layer weights:

```
function [actv_sig_out_store] = realizeMyNetwork(x, w_hidden, w_out)
sampSize = size(x,3);
for i = 1:sampSize
    % Calculate activations for hidden layer:
    actv_sig_hl = diag(w_hidden*x(:,:,i)'); % diagonal entries for weighted sums
    % Step function applied to each neuron.
    actv_sig_hl(actv_sig_hl >= 0) = 1;
    actv_sig_hl(actv_sig_hl < 0) = 0;
    actv_sig_hl = [actv_sig_hl; 1]; % Inp to Out layer, bias ext'd.
    % Calculate activations for the output layer:
    actv_sig_out = w_out*actv_sig_hl;
    % Step function applied to output neuron:
    actv_sig_out(actv_sig_out >= 0) = 1;
    actv_sig_out(actv_sig_out < 0) = 0;
    % Store elements:
    actv_sig_out_store(i,1) = actv_sig_out;
end
end
```

This code just multiplies the hidden layer weights with the inputs (which are the same, one with the noise, other without the noise). It decides on the activation using Heaviside function. The output is fed to the next layer, which is the output layer and the same operation is repeated there. This operation runs in a loop, as long as all samples in the validation set is completed.

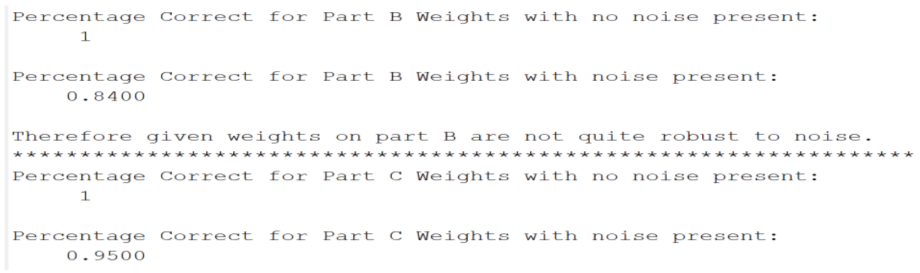
This function calculates the performance of the network weights by calculating how many samples the network managed to classify correctly:

```
function correct = perc_correct(y_act, y_pred, sample_size)
% Calculate percentage correct answers.
error = abs(y_act - y_pred); % make -1 and +1 errors the same
no_errors = sum(error);
correct = (sample_size - no_errors)/ sample_size;
end
```

This is the code to find and display the results, the same operation is also done for part c.

```
% Part b: - My weights:
part_b_output = realizeMyNetwork(x_in,wh_b,w_o);
part_b_noise_output = realizeMyNetwork(x_in_noise,wh_b,w_o);
perct_corr_b = perc_correct(y,part_b_output, size(y,1));
perct_corr_b_noise = perc_correct(y,part_b_noise_output, size(y,1));
% Display results:
disp('Percentage Correct for Part B Weights with no noise present:')
disp(perct_corr_b);
disp('Percentage Correct for Part B Weights with noise present:')
disp(perct_corr_b_noise);
disp('Therefore given weights on part B are not quite robust to noise.')
```

At the end MATLAB displays the following results. Running the code will prompt the screen which is below, with percentage correct data probably being changed due to random noise introduced to the samples:



```
Percentage Correct for Part B Weights with no noise present:
     1

Percentage Correct for Part B Weights with noise present:
    0.8400

Therefore given weights on part B are not quite robust to noise.
*******************************************************************
Percentage Correct for Part C Weights with no noise present:
     1

Percentage Correct for Part C Weights with noise present:
    0.9500
```

It looks like the weights on Part B is theoretically and practically correct, i.e., it satisfies all inequalities and has 100 percent performance. It is a neural representation of logic circuit (C1) that the question requires me to answer.

It is certain that the weights at part b realize given logic circuit as it was able to classify all data correctly without any noise. However, the weights were large, and the weighing factors were quite unbalanced. It is seen that when noise is introduced in the system, even though it did fine; it failed at 16 percent of the validation samples that we have given to the network. This is due to uneven and large weighing between inputs given they have the same standard deviation on the noise.

In part C, I have exploited the 3D input space representation of the inputs to the neurons. All neurons were at the 1st Octant and all inputs had the same standard deviation and mean values. Since all neuron had their input on the first octant (maybe except one which stands at the origin), it was easy to project the problem down to 2D and solve the problem on 2D and expand back to 3D and then to all 4 neurons simultaneously (Increasing dimensionality to 4D, where all class-1 members sit at corners of the strict positive octants).

Since the classes for the AND operation was linearly separable and class 0 had more members on their membership sets, I decided to use the lower bound of the support vector as my decision boundary. It allowed class-1 members to separate out from the class-0 members more than 2 standard deviations, which improved the classification given noise. It also made sure that at least 1 standard deviation error is correctly separated from class-1.

This choice on the decision boundary has increased the performance of the system with the noise 11 percent which is a significant increase (same inputs are used on hidden layers of part b and c). The noise free performance of hidden layer weights at part c is 100 %, implying that this neural network also fully realizes the logic circuit given in the question.

Finally, 5 % error on the network at part c can be caused by the misclassification of class 0, as decision boundary does not cover up more than 1 standard deviation. These samples may have been falsely classified as 1. Since the noise is fed to the inputs by random, one may not create a perfect classifier for this question but the proposed weights at part c is a very robust set of weights for this question. As the standard deviation of the noises increase, classes tend to intermix more, therefore the performance of the neural network decreases. In these cases, further changes on the network is required.

Question 3:

This question deals with a dataset of handwritten English alphabet letters. In this question I will implement a single layer neural network to recognize handwritten English letters. For the dataset the following specifications are given:

- trainims: 5200 images of single handwritten letters which are black and white. The image size is 28x28. There are 26 classes from A to Z and each class has 200 samples each. These samples are kept in alphabetical order. The size of the array is 28x28x5200.
- trainlbls: 5200 labels for the trainims. The label indices are in line with the trainims indices. The labels are from 1-26 which translate to A to Z respectively in alphabetical order. The size of the array is 1x5200.
- testims: It has the same structure with trainims with the exception that now we have 1300 samples available where each class have 50 samples available. The size of the array is 28x28x1300.
- Testlbls: It has the same structure with testims with the exception that now we have 1300 samples where each class have 50 samples labeled. The size of the array is 1x1300.

The indices of the arrays and vectors are significant as many operations on the neurons will be done in batches, so matrix operations will be utilized.

Part a:

After loading the dataset, I have done normalization on the data by dividing every 28x28 train and test images to 255. I also converted the images to double type from uint8 type.

```
% Load the dataset
load assign1_data1.mat
% Normalize images:
trainims = double(trainims) / 255;
testims = double(testims) / 255;
```

I have kept the dataset properties as hard coded variables. Using size method on dataset matrices to extract this information automatically is also viable, but I decided to go this way.

Since there are 26 classes in total, I have decided on a 5x6 subplot to show sample images. This layout will be preserved throughout this question. Since no specifications were given, I took the first entry of all classes as samples. The code below automatically searches for the classes, picks and subplots an example of each class.

Please note that method "sgtitle" is a built-in MATLAB function which is available after 2018b. If your version is lower than this, the code will fail here. Comment the line containing "sgtitle" to remove the title from plot. After doing this, the code will no longer fail (except for part b where final network weights are shown with another subplot).

The code and the output are given below. Correlation between sample classes are found using 'corr2' of MATLAB.

```
%% Part A:
% We have 26 classes, 28x28 pixels shown on a 6x5 matrix.
classes = 26; pxls = 28;  x_ax = 5;  y_ax = 6;
% Visualize sample image from each class:
% Since no specification given, I will pick first index of the given
% class, from training images.
% Sample image pre-allocation:
smpl_ims = zeros(pxls, pxls, classes);
for i = 1:classes
    find_im_idx = find(trainlbls==i,1);
    smpl_ims(:,:,i) = trainims(:,:,find_im_idx);
    smpl_im = trainims(:,:,find_im_idx);
    subplot(x_ax,y_ax,i); imshow(smpl_im);
    colormap(gray);
end
% Beautify plot by adding title: -- Works after 2018b, comment if fails.
sgtitle('Alphabet Letters Labeled From 1-26 (Left to Right, Up to Down)');
% Correlation Coefficients Calculation:
corr_vals = zeros(classes,classes);
for i = 1:classes
    for j=1:classes
        corr_vals(i,j) = corr2(smpl_ims(:,:,i),smpl_ims(:,:,j));
    end
end
% Print Correlation values in matrix format:sgtitle('Alphabet Letters Labeled From 1-
26 (Left to Right, Up to Down)');
% disp('Correlation Values in Matrix Form');
% disp(corr_vals); % display correlation in matrix form
figure; imagesc(corr_vals); colormap(gray); colorbar; %display image form
title('Correlation Matrix of Sample Letters')
```

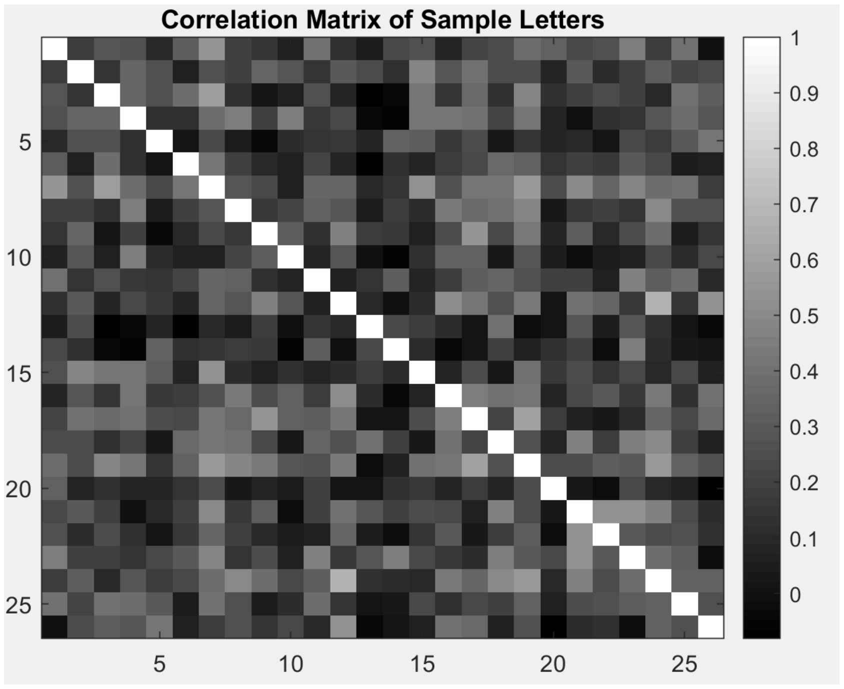


Figure 10. Correlation Matrix of Sample Letters

Figure 10 is formed from the 2D correlations between sample classes that I have selected. These classes can be seen below. Figure 10 suggests, within class variability is very low. Given we compare the class with itself, its only natural to see that within class variability is very low. From upper and lower triangles, we can see that across class variability is fair to large. For example, 4-14 are significantly different than each other. See the figure below to see 4 is ‘D’ and 14 is ‘N’. Again looking at 8-‘H’ and 14-‘N’ we see medium variability. Finally, 12-‘I’ and 24-‘X’ look somewhat similar. Using Figure 11 below, one may confirm the results described above.

Note that running Question 3, using the given code will also output these. I believe one should also look at within-class variability by comparing the correlations between different samples of each classes. Because further in the assignment we will encounter cases where some letters were drawn in many different ways, where some are drawn persistently in the same way, which will reduce performance the learned weights, compared to other classes.

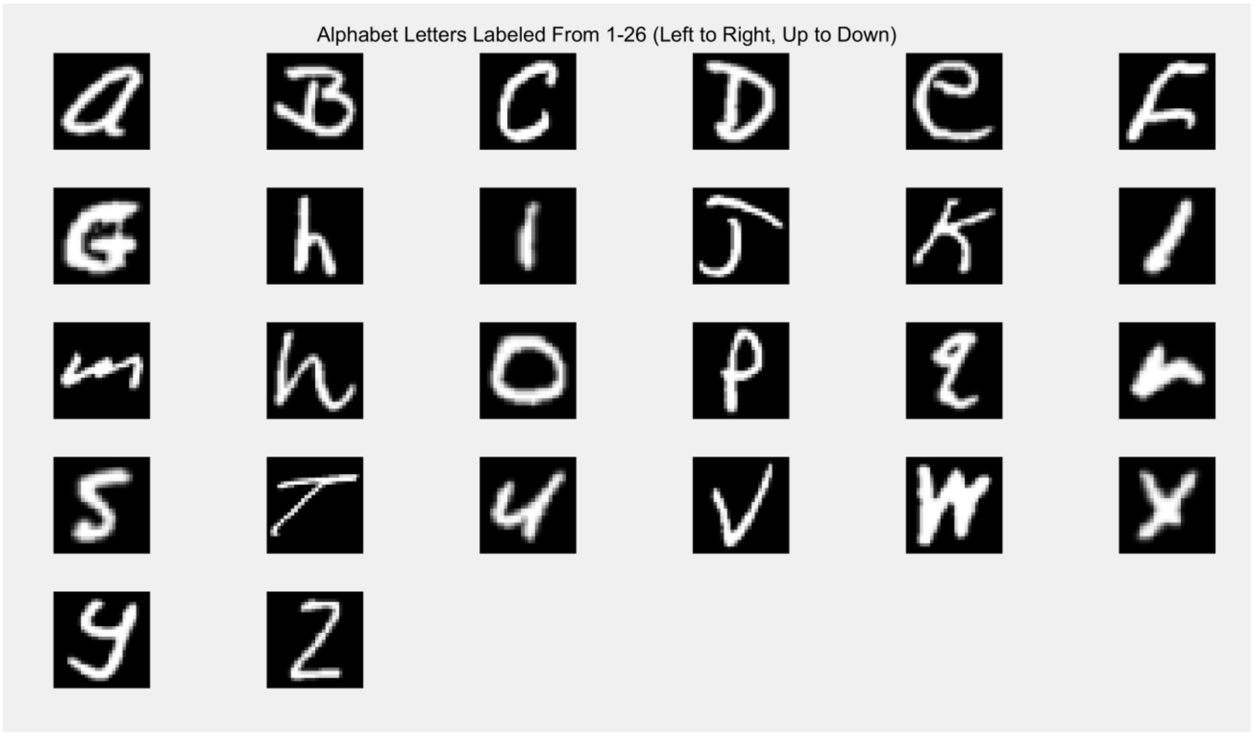

Alphabet Letters Labeled From 1-26 (Left to Right, Up to Down)

Figure 11. Handwritten English Alphabet Letters (Labeled in Alphabetical Order from 1 to 26)

Part b:

The following conditions and requirements are given for this question:

- Network will be Single Layer Perceptron (SLP): Since we have 26 classes, we will have 26 neurons which will output a response for their own respective class.
- Initial weights and biases should be kept in separate matrices/vectors and should be drawn from a Gaussian distribution with mean 0 and standard deviation 0.01.
- The activation function for the single layer is sigmoid, which is a unipolar activation which outputs a continuous value from 0 to 1. Point of inflection is at 0.5.
- Online training algorithm picks image from training set at random and does this for 10000 iterations.
- Network weights are to be updated by gradient descent rule. Final bias and weights values must be kept, the algorithm must be run for different learning rates and a best learning rate should be selected using these experiments.
- Loss function is mean squared error function and learning rate must be tuned according to the loss function.
- Finalized weights should be displayed on the screen just like I did in part a. Finalized weights must be found in accordance to the best learning rate.

Using these conditions and requirements; I will propose the following network:

- The SLP will have 26 units available, one for each character class which are fully connected to the pixels of the training and testing images with their respective weights.
- The images will be flattened for ease matrix multiplications. Each image was 28x28, therefore the images will now be flattened to a vector with size, 784x1. Weights matrix will have size 26x784, as we have 784 weights per each neuron in the single layer. Since there are 26 neurons, 26x1 bias vector will be described.
- All weights and biases will be drawn from N(0,0.01).
- Labels for the training and test sets will be mapped from 1-26 range to a one hot encoded list. Every label will be represented by a 1x26 vector where j'th element will be 1 and the rest is zero. 'j' stands for the initial mapping from 1 to 26. This is done in order to vectorize gradient descent algorithm. It allows error to be calculated for every neuron in each iteration to simultaneously update the weights. For wrong classifications of the neuron it decreases the weights and increases the weights if the classification is correct simultaneously (Hebbian learning).

The following code is written for part b. I will go along the lines and then explain the outputs.

```
%% Part B:
mu = 0; sigma = 0.01; % Given Gaussian variables.
its = 10000; % Given iteration rate
rng default; % for reproducability in MATLAB, adviced by documentation.

% Initialize w & b: NN total number of parameters 28x28
% - flattened for my ease, I will work with vectors rather than matrices
% Rows: Connecting neuron index, Columns: Weights to next neuron
[w,b] = init_with_gauss(mu, sigma, classes, ...
    pxls, pxls);

% One hot-encode training and test data labels: To be used later:
% Rows: Labels, Columns: Neuron index for the label
trainlbls_onehot = encode_onehot(trainlbls);
testlbls_onehot = encode_onehot(testlbls);

% Flatten the input images:
% Columns: Image samples, Rows: Data for given column (image sample)
trainims = double(reshape(trainims, [size(trainims,1)*size(trainims,2) ...
    , size(trainims,3)]));
testims = double(reshape(testims, [size(testims,1)*size(testims,2) ...
    , size(testims,3)]));

% Learning & Tuning for Learning Rate:
% Online learning will be used. (update every sample)
% Keep the record in each iteration in memory, specifically, w, b and MSE
% values.
lr = 0:0.05:2;
for l = 1:size(lr,2)
    [w_lr(:,:,l),b_lr(:,l),mse_lr(:,l)] = ...
        learn_weights(trainims,trainlbls_onehot,w, b, lr(l), its);
end

% Pick best learning rate: -- take minimizinng MSE over iterations
[minMSE, best_lr_idx] = min(sum(mse_lr));

% Display results:
disp('Best Learning Rate Found:'); lr_star = lr(best_lr_idx);
disp(lr_star);
%%
figure();
plot(lr,sum(mse_lr)/10000);
title('MSE vs Learning Rate'); xlabel('Learning Rate'); ylabel('MSE');
hold on; plot(lr(best_lr_idx),(mse_lr(best_lr_idx))/10,'*r')
legend('lr vs MSE','Best point')

%% Show network weights for the best learning rate available:
figure;
for i = 1:26
    best_w(i,:,:) = reshape(w_lr(i,:,best_lr_idx), [28, 28]);
    subplot(x_ax,y_ax,i);
        imagesc(squeeze(best_w(i,:,:))); colormap(gray);
%   imshow(squeeze(best_w(i,:,:))*10); %scaled for better visual
end
sgtitle('Finalized Weights of Neuron from 1 to 26');
```

First I set up the mu, sigma parameters for the randomized weight and bias initialization and also set the number of iterations fixed with the value 10000 requested in the question. Then the following custom code is executed for initialization of weights and biases, as explained above:

```
function [w,b] = init_with_gauss(mu, sigma, layerSize, imSize_x, imSize_y)
% Initialize weights and bias by gaussian distribution, given layer size
% Customized to convert 2D image connections, flattening them to vectors
w = normrnd(mu, sigma, [layerSize,imSize_x*imSize_y]); % includes bias term
b = normrnd(mu, sigma, [layerSize, 1]);
end
```

The function is straight forward. It draws a 26x784 matrix for the weights for all 26 neurons and a vector of 26x1 for the biases. These will be later used for training.

Train and test labels were represented as one hot encoded labels using the following custom function after setting initial weights:

```
function [encoded] = encode_onehot(labels)
% Encode 1-26 labels to one hot version.
encoded = zeros(size(labels,1),26);
for i=1:size(labels,1)
    encoded(i,labels(i)) = 1;
end
end
```

This code generates a vector of zeros with size 1x26 and places 1 at the position where initial label was seen for example for label 5-'E' we have:

$$5 = [0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$$

All labels were transformed to one-hot encoded labels just like the example shown above. This is done for all training and testing labels. Note that the indices of the samples were not changed in this operation therefore, training and testing images and their respective labels are still in the same order.

```
% Flatten the input images:
% Columns: Image samples, Rows: Data for given column (image sample)
trainims = double(reshape(trainims, [size(trainims,1)*size(trainims,2) ...
    , size(trainims,3)]));
testims = double(reshape(testims, [size(testims,1)*size(testims,2) ...
    , size(testims,3)]));
```

For the matrix operations the training and test images are also flattened and converted to double from uint8 with the code given above. The images become 1x784, training dataset becomes 784x5200 and test dataset becomes 784x1300.

```
% Learning & Tuning for Learning Rate:
% Online learning will be used. (update every sample)
% Keep the record in each iteration in memory, specifically, w, b and MSE
% values.
lr = 0:0.05:1;
for l = 1:size(lr,2)
    [w_lr(:,:,l),b_lr(:,l),mse_lr(:,l)] = ...
        learn_weights(trainims,trainlbls_onehot,w, b, lr(l), its);
end
```

The code above is then used for training networks using training images and with random image selection. It is repeated for many different learning rates to see which learning rate minimized MSE loss to furthest extent. First I will explain the implementation of custom function "learn_weights", then I will be talking about the best learning rate and the visualization of the weights.

The following is the "learn_weights' function:

```
function [w, b, mse] = ...
    learn_weights(images_flat,labels_onehot,wb_init,b_init,lr, iterations)
% This function does the learning of the weights, bias and MSE and also
% updates weights. Specifically written to ease read in Parts B,C,D.
w = wb_init; b = b_init; mse = zeros(1,iterations);
for it = 1:iterations
    % Pick a random sample: - Already normalized. Flattened to ease my code
    % Image converted to double to allow matrix calculations with w.
    idx = ceil(size(images_flat,2)*rand);
    x = images_flat(:,idx); % flattened img selected, with bias added
    % Do feedforward operation with sigmoid:
    output = sigmoid(w*x-b)';
    error = (labels_onehot(idx,:)-output);
    % MSE loss at this stage
    mse(it) = 1/26 .* error * error';
    % Do weight update: - Gradient Descent w/ sigmoid:
    dw = 2*(error .* ((output) .* (1-(output)))) .* x;
    db = 2*(error .* ((output) .* (1-(output))));
    % Optimizes the parameters, -- Namely, update w and b
    w = w + lr*dw';
    b = b + lr*db';
    % Returns MSE for 10000 iterations and final w and b
end
end
```

This is the sigmoid implementation in MATLAB. It is written in this way to read the implementation of the "learn_weights" easily. This sigmoid function takes the activation signal and applies unipolar sigmoid activation.

```
function out = sigmoid(z)
% Calculates sigmoid function given a matrix value set.
% Used to simplify code in Question 3 - Part B.
out = 1./(1+exp(-z));
end
```

The function "learn_weights" takes 6 parameters and output final weights, final bias and MSE records for all iterations that the loop has. It takes trainims in flattened form, trainlbls in one-hot encoded form, initial weight and bias terms, the learning rate and the number of iterations to train from respectively.

From the dataset it picks a random index and retrieves the flattened image at that index. Since this we are working with a single layer network, this image is being fed to all neurons at the same time. The activation function becomes:

$$v = w * x - b$$

Since weights have 26x784 size and the selected image has 784x1 size, the multiplication of these matrices gives a 26x1 matrix. The bias is subtracted from this easily as it also is 26x1. Bias was added without using extended form. This is fed to the sigmoid function and the activation signal is mapped from (-∞, ∞) to [0,1] range. Since ‘v’ is 26x1 we know that sigmoid function gives the output of all 26 neurons. This means the forward pass is done simultaneously for all the neurons.

The output of the sigmoid function ‘output’ gives values from 0 to 1. This can be used as a bounded error metric. The output gives confidence about how likely that neuron is to be activated. Since every sample image is a member of a class, one may use winner takes all approach. Here, I will be using the one-hot encoded signal as a binary ground truth signal. This means if sigmoid output is close to 0 for a neuron and the input is not in the class of neuron the error between sigmoid output and one-hot encoded label will be zero. If the neuron fires close to 1 and the signal states that that the input is in the neuron’s class, the error will still be one. Any output of the sigmoid in between will be considered as errors, or weak correspondence and gradient descent algorithm will be used to punish errors by changing associated weights of the neuron while strengthening correct associations just like we have seen as Hebbian learning in the course.

```
error = (labels_onehot(idx,:)-output);
```

In order to update the weights, we need to implement gradient descent. Until now we have used sigmoid function and let the cost function be MSE function. MSE calculation is given by:

```
% MSE loss at this stage
mse(it) = 1/26 .* error * error';
```

The one-hot encoded model does not only vectorize the error calculations, it also makes MSE fast to calculate. Error was a 1x26 vector and for each iteration the code above will give a single MSE output, for which we will use to optimize our learning rate.

To update the weights correctly using the error metric, we need to use gradient descent. Basically, what we try to do at each iteration of the algorithm is given below:

$$w = w - lr * \Delta w$$

$$b = b - lr * \Delta b$$

The gradient descent over the MSE can be found by using the chain rule. Lets write the MSE function (left index, right vectoral form):

$$MSE = \frac{1}{26}\sum_{i=1}^{26}\left(y_i - \sigma(x_i, w)\right)^2 = \frac{1}{26}\left(y - \sigma(x, w)\right)^2$$

Let’s ignore 1/26 term as it does not have any effect on the differentiation. Using the chain rule, we get:

$$\frac{\partial MSE}{\partial w} = 2 * \left(y - \sigma(x, w)\right) * \frac{\partial \sigma}{\delta w} * x$$

$$\frac{\partial MSE}{\partial b} = 2 * \left(y - \sigma(x, w)\right) * \frac{\partial \sigma}{\delta b}$$

We know that $\sigma' = \sigma * (1-\sigma)$:

$$\frac{\partial MSE}{\partial w} = 2 * \big(y - \sigma(x,w)\big) * \sigma(x,w) * (1 - \sigma(x,w)) * x$$

$$\frac{\partial MSE}{\partial b} = 2 * \big(y - \sigma(x,w)\big) * \sigma(x,w) * (1 - \sigma(x,w))$$

Note that $\big(y - \sigma(x,w)\big)$ is the one-hot encoded error, $\sigma(x,w)$ is the output of the sigmoid function and x is the input that we feed to the network at given iteration. We can code the gradients as follows:

```
dw = 2*(error .* ((output) .* (1-(output)))) .* x;
db = 2*(error .* ((output) .* (1-(output))));
```

These gradients can be scaled with 1/26 and will update the weights simultaneously for every neuron. If the error is very small, the update will be very small, or if the error is very big, depending on the sign, the gradients will be significantly changed in favor of the winning neuron's weights and against the favor of the losing neuron's weights. At the end of all iterations, it is expected that the weights associated with letter's pixel positions will be amplified where other weights will be silenced.

```
% Optimizes the parameters, -- Namely, update w and b
    w = w + lr*dw';
    b = b + lr*db';
```

The code above gives the update rules. Since this algorithm was run for various learning rates, the code below, tries many different learning rates to find the learning rate that minimizes overall MSE, it plots overall MSE for different learning rates for visualization. Then for the best learning rate available, it shows the weights in image format. Here are the outputs:

```
% Pick best learning rate: -- take minimizinng MSE over iterations
[minMSE, best_lr_idx] = min(sum(mse_lr));

% Display results:
disp('Best Learning Rate Found:'); lr_star = lr(best_lr_idx);
disp(lr_star);
%%
figure();
plot(lr,sum(mse_lr)/10000);
title('MSE vs Learning Rate'); xlabel('Learning Rate'); ylabel('MSE');
hold on; plot(lr(best_lr_idx),(mse_lr(best_lr_idx))/10,'*r')
legend('lr vs MSE','Best point')

%% Show network weights for the best learning rate available:
figure;
for i = 1:26
    best_w(i,:,:) = reshape(w_lr(i,:,best_lr_idx), [28, 28]);
    subplot(x_ax,y_ax,i);
        imagesc(squeeze(best_w(i,:,:))); colormap(gray);
%   imshow(squeeze(best_w(i,:,:))*10); %scaled for better visual
end
sgtitle('Finalized Weights of Neuron from 1 to 26');
```

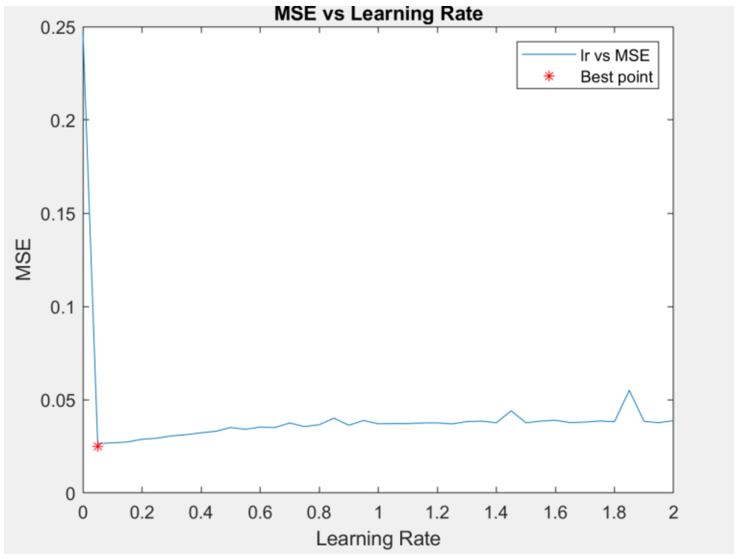


Figure 12. MSE vs Learning Rate Graph

The best learning rate in Figure 12 was displayed on command window too. It is found to be 0.05, with tests that start from 0 to 2 with 0.05 increments. The MSE starts to slightly increase after this point.

```
Best Learning Rate Found:
    0.0500
```

Here is the network weight plot for the best learning rate after 10000 iterations. Having more iterations can teach the network more:

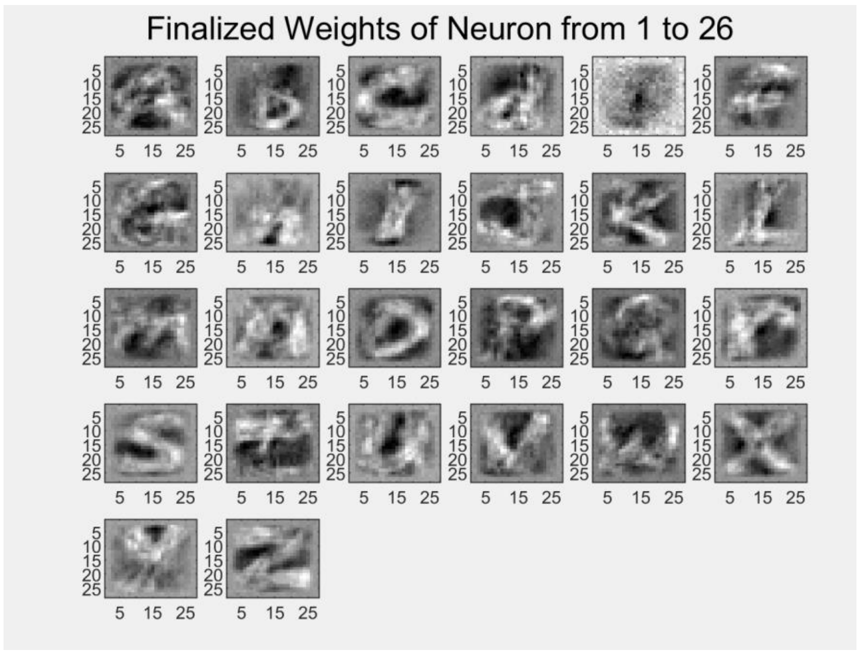


Figure 13. Finalized Weights, LR of 0.05

Note that the images were picked at random. We see that the class weights try to extract the overall shape of the letter. The shapes are similar to the letters; therefore, it can be said that the network is successful. However, class ‘E’ was not detected clearly. Using more iterations may solve this issue. However, if we look at the intra-class variability of ‘E’ versus other letters, ‘E’ can be mixed with other letters too, as the shape is somewhat like other classes partially in many different locations. Due to this, as we train ‘E’ may have been misclassified as other letters and examples of ‘E’ was not detected enough to reinforce training of the weights for that class.

A good way is to increase number of layers to detect more non-linearities. Another improvement can be use of convolutional layer rather than fully connected layers.

Part c:

This part was straightforward. As I have found the best learning rate available, I have used low (*0.01) and high (*1000) learning rates. The code is given below:

```
%% Part C:
lr_low = 1/100 * lr_star;
lr_high = 100 * lr_star;
lr_c = [lr_low, lr_star, lr_high];
% Do the training for given set of learning rates
for k = 1:3
    [w_lrc(:,:,k),b_lrc(:,k),mse_lrc(:,k)] = ...
        learn_weights(trainims,trainlbls_onehot,w, b, lr_c(k), its);
end
%% Plot MSE curves:
figure(); xlabel('Iterations'); ylabel('MSE values'); hold on;
plot(1:its,mse_lrc(:,1)); plot(1:its,mse_lrc(:,2));
plot(1:its,mse_lrc(:,3));
legend('lr-low', 'lr-best', 'lr-high'); grid on;
%%
% Display total average mean squared error:
averages = 1/10000 * sum(mse_lrc);
disp('Average MSE of lr_low:');
disp(averages(1));
disp('Average MSE of lr_best:');
disp(averages(2));
disp('Average MSE of lr_high:');
disp(averages(3));
```

The averages over 10000 iterations was displayed on the command window, where the MSE over iteration was also plotted by this code. Since the training process was explain in part d, I am directly heading over the results.

These are the mean MSE’s of low (0.0005), best (0.05) and high (5) learning rates:

```
Average MSE of lr_low:
    0.0467

Average MSE of lr_best:
    0.0267

Average MSE of lr_high:
    0.0391
```

The mean MSE is in line with common sense, as the best learning rate gives the least mean squared values. The graphs for MSE evolution with respect to iterations is given below:

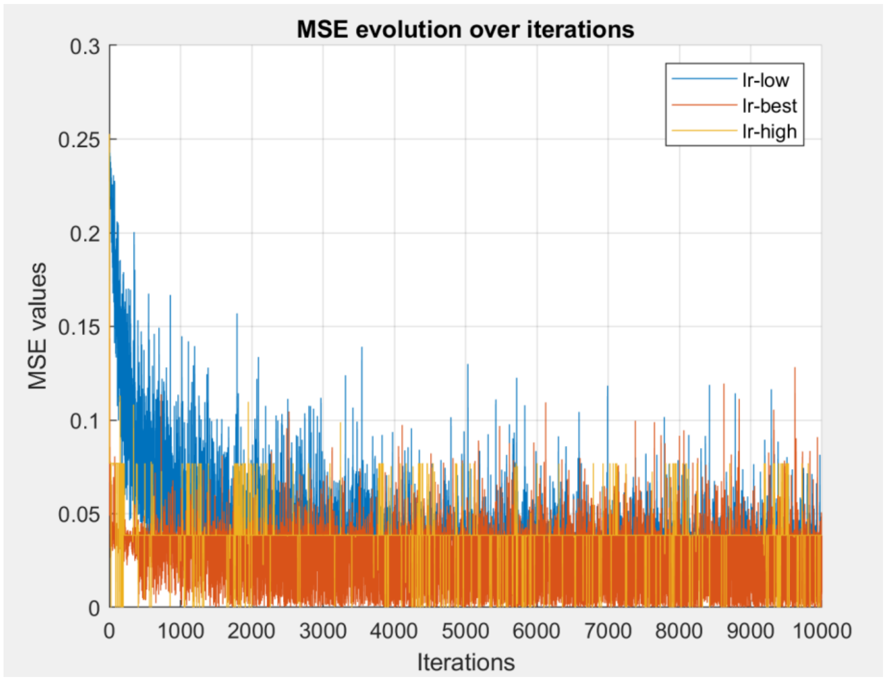


Figure 14. MSE evolution over iteration for low, best and high learning rates.

It is seen with the low learning rate that the MSE converges very slowly compared to high and best learning rates. On average, it has the largest MSE value, probably due to slow convergence. High learning rate directly converges to low values of MSE however, it constantly spikes to large MSE values due to large update on the weights. Best learning rate's MSE converges almost as fast as MSE of high learning rate, however it does not overshoot frequently when compared with high learning rate. Therefore, it is clearly visible that using a very small or a very large learning rate is not beneficial for a neural network, if not it does not affect performance of the network badly.

Part d:

The following code is written for part d.

```
%% Part D:
% Do the training for given set of learning rates.
% Using the weights from part C: 'w_lrc'
w_dlow = w_lrc(:,:,1); w_dstar = w_lrc(:,:,2); w_dhigh = w_lrc(:,:,3);
b_dlow = b_lrc(:,1); b_dstar = b_lrc(:,2); b_dhigh = b_lrc(:,3);
% Calculate the output given the input
out_dlow = sigmoid((w_dlow*testims)-repmat(b_dlow, [1,1300]));
out_dstar = sigmoid((w_dstar*testims)-repmat(b_dstar, [1,1300]));
out_dhigh = sigmoid((w_dhigh*testims)-repmat(b_dhigh, [1,1300]));
% Winner takes all: May the best neuron win: Get winning idx per each
% sample.
[vals_dlow, indices_dlow] = max(out_dlow);
[vals_dstar, indices_dstar] = max(out_dstar);
[vals_dhigh, indices_dhigh] = max(out_dhigh);
% Now do check classification error using original labels, one hot enode
% not needed as comparison from indices are easy:
perc_corr_dlow = (1 - sum(abs(indices_dlow ~= testlbls')) / 1300);
perc_corr_dstar = (1 - sum(abs(indices_dstar ~= testlbls')) / 1300);
perc_corr_dhigh = (1 - sum(abs(indices_dhigh ~= testlbls')) / 1300);
% Display the results:
disp('Percentage of output correct for lr: 0.0005 - Low Learning Rate');
disp(perc_corr_dlow)
disp('*************')
disp('Percentage of output correct for lr: 0.05 - Best Learning Rate');
disp(perc_corr_dstar)
disp('*************')
disp('Percentage of output correct for lr: 5 - High Learning Rate');
disp(perc_corr_dhigh)
```

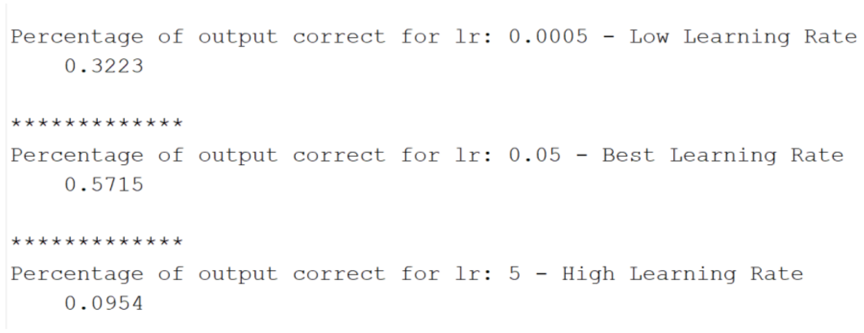
The weight and bias matrices were calculated in part c, so those results were migrated here. For the test set, I have used trained weights multiplied them with the input and added learned bias. This forced neurons to weight the input in the way, which it previously has seen. For example, if the input character is ‘A’ letter, the neurons which learned the image of the general shape ‘A’ (in many of its forms), resulted in a much higher activation signal as the general shape of the input and the weights have matched. All other neurons had reduced activation signals due to the mismatch of the input shape and their learned letter shape.

After sigmoid operation, activation signals are mapped from negative infinity-positive infinity range to [0,1] range. Here, we take the maximum value of sigmoid output vector to decide which class the input is in, using a winner-take-all approach. This process is done as a vectoral operation therefore all test images were processed simultaneously.

As neuron indices implicitly contain class label information, the maximum neuron’s index translate to the class membership of the input.

```
perc_corr_dstar = (1 - sum(abs(indices_dstar ~= testlbls')) / 1300);
```

This operation above compares test labels (which were not one-hot encoded) and the maximum output indices. If they are not equal, it returns 1. Since the dataset is analyzed in a single line, all errors are summed, subtracted from 1 and divided to 1300 (test sample size) to calculate percentage correct performance metric. This metric is displayed on the command prompt of MATLAB for low, best and high learning rates. The results are listed below:



```
Percentage of output correct for lr: 0.0005 - Low Learning Rate
    0.3223

*************
Percentage of output correct for lr: 0.05 - Best Learning Rate
    0.5715

*************
Percentage of output correct for lr: 5 - High Learning Rate
    0.0954
```

As expected, the best learning rate gave the best performance in overall. The results are in line with MSE minimization which are calculated, compared and discussed in part c. High learning rate had fast convergence but overshot very much, where low learning rate had lower rate of convergence and the mean MSE was large compared to MSE of best learning rate.

The performance of the neural network topped at 57% approximately. This is not very good. However, given this is a single layer perceptron, the network was not able to extract many more detailed non-linear relations in the images. Increasing the layer size and number of layers can increase the change of this network to perform better. However, one should also be careful with vanishing gradients when deep fully connected layers are used. One other approach is to use convolutional layers instead of fully connected layers. However, implementation of these algorithms is not in the scope of this report.

References

[1] D. P. Bertsekas and J. N. Tsitsiklis, Introduction to Probability, Belmont: Athena Scientific, 2008.

[2] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, Chennai: The MIT Press, 2016.

[3] T. Cukur, "EEE 443 - Neural Network Lecture Notes - Chapter 2," Ankara, 2019.

[4] J. F. Wakerly, Digital Design Principles and Practices, New Jersey: Pearson Prentice Hall, 2007.

[5] T. Hastie, R. Tibshirani and J. Friedman, The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Stanford: Springer, 2008.

Appendix:

In the MATLAB program associated with this assignment, the questions are written in functions. For the sake of clarity, the question functions and their helper functions, if there are any are separately put in the appendix. For completeness, the main caller function is also given in the appendix.

Note that the complete MATLAB code is available as “hasanemre_erdemoglu_21401462_hw1.m” file.

Question 1 – MATLAB Code:

There is no MATLAB code associated with this question.

Question 2 – MATLAB Code:

```matlab
%% Question 2 - Initialization:
% Clear everyhing:
clear; close all; clc;

% ***********************************************************************
% Noise is N(0,0.2). Truth table for quadruple (x1, x2, x3, x4) is
% generated by hand. It is expanded with bias term -1, therefore we have
% 16x5 matrix.
%
% Then we generate the logic table for the neurons. Then we expand the
% logic table to have more samples and we add Gaussian noise.
%
% The ground truths for the XOR operation was calculated by hand. It is a
% very fundamental calculation which involves filling up the SOP logic
% table. Since this operation is fairly easy it is omitted from the
% report and the final results are hard coded here.
%
% The rest of the code just uses hidden layer bias/weights and output layer
% weights and biases to construct the neural realization of given logic
% circuit.
%
% Note that, as the question focuses on Hidden Layer and robustness through
% hidden layer, I omitted the robustness of output layer. Hidden layer
% realizes AND operations using triplets of inputs (with possible inverses)
% where in marginally stable case sum of weights + bias should be equal to
% zero at least. By doing that, bias enforces all inputs to be correct to
% fire the neuron. Output layer realizes OR gate and bias must be equal to
% weight of one of the neurons. (Since its a OR operation on binary inputs
% all weights can be equalized and normalized). However, this makes the
% neural network fragile to any kind of noise.
%
% Due to this reason, I assume if the hidden layer is fully robust,
% marginal case of [1 1 1 1 -1] will work just fine in the output layer.
% However, even when best support vectors are selected, the noise may
% perturb the outputs of the hidden layer. Even when we select the best
% support vectors for the output layer too, noise may still overcome the
% signal, if large enough. Therefore it is not possible to guarantee,
% perfect performance on the neural network, given noisy data.
%
% Summary for output robustness: The question didnt ask about it and so I
% assumed that I am not asked to implement robustness there.
%
% Self note: Normalization of weights could be nice,smaller coefficients
% imply smaller error terms.
% *************************************************************************


%% Part D:
mu = 0; sigma = 0.2; % Gaussian noise parameters

% Generate input samples:

% First create noise-free logic table: [X|-1]
x = [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];

% From input samples, generate neuron based logic tables:
x_in = zeros(4,5,16); % pre-init for fast computation
for i=1:size(x,1)
    % Realizes neuron 1's input from logic table, x:
    x_in(1,:,i) = [x(i,1), x(i,2), x(i,3), x(i,4), -1];
    % Realizes neuron 2's input from logic table, x:
    x_in(2,:,i) = [x(i,1), not(x(i,2)), x(i,3), x(i,4), -1];
    % Realizes neuron 3's input from logic table, x:
    x_in(3,:,i) = [not(x(i,1)), x(i,2), not(x(i,3)), x(i,4), -1];
    % Realizes neuron 4's input from logic table, x:
    x_in(4,:,i) = [not(x(i,1)), x(i,2), x(i,3), not(x(i,4)), -1];

end

% Generate output from XOR operation: A*notB + notA*B
% A = X1 + notX2;    B = notX3 + notX4; out = A*notB + notA*B
% out = (X1 + notX2) * not(notX3 + notX4)+not(X1 + notX2) * (notX3 + notX4)
y = zeros(16,1); y(4:7) = 1; y(12) = 1; y(16) = 1;

% Now repeat this 25 times:
xx = x_in; % debug
x_in = repmat(x_in, [1, 1, 25]);
y = repmat(y, [25, 1]);

% Now generate Gaussian noise vector for the inputs - 400x4 noise terms
noise = normrnd(mu, sigma, [4,5,400]); % only data samples are affected by noise.
noise(:,5,:) = 0; % exclude bias from noise

% Add noise to the input data:
x_in_noise = x_in + noise;

% Form the neural networks: Hidden layer weights taken from Part B and C:
% Due to implementing the logic inversions above, negative signs are
% omitted.
wh_b = [2,  0, 1.5, 2, 5; ...
    0, 3, 3,   4, 9; ...
    2,  8, 3,   0, 12; ...
    2,  8, 0,   3, 12];

% 3 neurons in each row: No noise --> 1+1+1-3=0 is marginally stable and
% correct. We have mean 0, std 0.2: AND operation multiplies.
% Multiplication of 3 Gaussians
wh_c = [1,  0, 1, 1, 2.717; ...
    0, 1, 1,   1, 2.717; ...
    1,  1, 1,   0, 2.717; ...
    1,  1, 0,   1, 2.717];

w_o = [1 1 1 1 -1]; % Weights for the output layer. Explained in report.
```

```
% Part b: - My weights:
part_b_output = realizeMyNetwork(x_in,wh_b,w_o);
part_b_noise_output = realizeMyNetwork(x_in_noise,wh_b,w_o);

perct_corr_b = perc_correct(y,part_b_output, size(y,1));
perct_corr_b_noise = perc_correct(y,part_b_noise_output, size(y,1));

% Display results:
disp('Percentage Correct for Part B Weights with no noise present:')
disp(perct_corr_b);
disp('Percentage Correct for Part B Weights with noise present:')
disp(perct_corr_b_noise);
disp('Therefore given weights on part B are not quite robust to noise.')

% Part c:
part_c_output = realizeMyNetwork(x_in,wh_c,w_o);
part_c_noise_output = realizeMyNetwork(x_in_noise,wh_c,w_o);

perct_corr_c = perc_correct(y,part_c_output, size(y,1));
perct_corr_c_noise = perc_correct(y,part_c_noise_output, size(y,1));

% Display results:
disp('********************************************************************')
disp('Percentage Correct for Part C Weights with no noise present:')
disp(perct_corr_c);
disp('Percentage Correct for Part C Weights with noise present:')
disp(perct_corr_c_noise);
% % disp('Therefore given weights on part B are not quite robust to noise.')


%% Helpers:
function out = not(in)
% Does boolean inversion for the inputs.
if in == 1
    out = 0;
else
    out = 1;
end
end

function [actv_sig_out_store] = realizeMyNetwork(x, w_hidden, w_out)
sampSize = size(x,3);
for i = 1:sampSize
    % Calculate activations for hidden layer:
    actv_sig_hl = diag(w_hidden*x(:,:,i)'); % diagonal entries for weighted sums

    % Step function applied to each neuron.
    actv_sig_hl(actv_sig_hl >= 0) = 1;
    actv_sig_hl(actv_sig_hl < 0) = 0;

    actv_sig_hl = [actv_sig_hl; 1]; % Inp to Out layer, bias ext'd.

    % Calculate activations for the output layer:
    actv_sig_out = w_out*actv_sig_hl;

    % Step function applied to output neuron:
    actv_sig_out(actv_sig_out >= 0) = 1;
    actv_sig_out(actv_sig_out < 0) = 0;

    % Store elements:
    actv_sig_out_store(i,1) = actv_sig_out;

end
```

```
end

function correct = perc_correct(y_act, y_pred, sample_size)
% Calculate percentage correct answers.
error = abs(y_act - y_pred); % make -1 and +1 errors the same

no_errors = sum(error);

correct = (sample_size - no_errors)/ sample_size;
end
```

Question 3 – MATLAB Code:

```
%% Question 3 - Initialization & Explanation:
% Clear everyhing:
clear; close all; clc;

% Load the dataset
load assign1_data1.mat

% Normalize images:
trainims = double(trainims) / 255;
testims = double(testims) / 255;

% Dataset Explanation: *****************************************************
% Training: 28x28 images - 5200 samples
% Test: 28x28 images - 1300 samples
%
% Images are kept in unsigned 8 bit integers.
% Color is grayscale, scaled between 0 and 255.
%
% Labels for the sets are kept seperately.
% Image and label indices match for both sets.
%
% There are 26 classes available in the dataset, sorted, groupped &
% ordered from 1 to 26.
%
% The images contain English letters, 1 being 'A' and 26 being 'Z'.
% These are deduced using observation on data.
% *************************************************************************

%% Part A:
% We have 26 classes, 28x28 pixels shown on a 6x5 matrix.
classes = 26; pxls = 28;  x_ax = 5;  y_ax = 6;

% Visualize sample image from each class:
% Since no specification given, I will pick first index of the given
% class, from training images.

% Sample image pre-allocation:
smpl_ims = zeros(pxls, pxls, classes);
for i = 1:classes
    find_im_idx = find(trainlbls==i,1);
    smpl_ims(:,:,i) = trainims(:,:,find_im_idx);
    smpl_im = trainims(:,:,find_im_idx);

    subplot(x_ax,y_ax,i); imshow(smpl_im);
    colormap(gray);
end

% Beautify plot by adding title: -- Works after 2018b, comment if fails.
sgtitle('Alphabet Letters Labeled From 1-26 (Left to Right, Up to Down)');

% Correlation Coefficients Calculation:
corr_vals = zeros(classes,classes);
for i = 1:classes
    for j=1:classes
        corr_vals(i,j) = corr2(smpl_ims(:,:,i),smpl_ims(:,:,j));
    end
end

% Print Correlation values in matrix format:sgtitle('Alphabet Letters Labeled From 1-26 (Left to
Right, Up to Down)');
% disp('Correlation Values in Matrix Form');
% disp(corr_vals); % display correlation in matrix form
figure; imagesc(corr_vals); colormap(gray); colorbar; %display image form
title('Correlation Matrix of Sample Letters')
% clc; close all; % ******************************************************* debug

%% Part B:
mu = 0; sigma = 0.01; % Given Gaussian variables.
```

```
its = 10000; % Given iteration rate
rng default; % for reproducibility in MATLAB, adviced by documentation.

% Initialize w & b: NN total number of parameters 28x28
% - flattened for my ease, I will work with vectors rather than matrices
% note that we actually have [w|b] with given custom function.
% Rows: Connecting neuron index, Columns: Weights to next neuron
[w,b] = init_with_gauss(mu, sigma, classes, ...
    pxls, pxls);

% One hot-encode training and test data labels: To be used later:
% Rows: Labels, Columns: Neuron index for the label
trainlbls_onehot = encode_onehot(trainlbls);
testlbls_onehot = encode_onehot(testlbls);

% Flatten the input images:
% Columns: Image samples, Rows: Data for given column (image sample)
trainims = double(reshape(trainims, [size(trainims,1)*size(trainims,2) ...
    , size(trainims,3)]));
% trainims = [trainims; -ones(1,5200)];

testims = double(reshape(testims, [size(testims,1)*size(testims,2) ...
    , size(testims,3)]));
% testims = [testims; -ones(1,1300)];

% Learning & Tuning for Learning Rate:
% Online learning will be used. (update every sample)
% Keep the record in each iteration in memory, specifically, w, b and MSE
% values.
lr = 0:0.05:2;
for l = 1:size(lr,2)
    [w_lr(:,:,l),b_lr(:,l),mse_lr(:,l)] = ...
        learn_weights(trainims,trainlbls_onehot,w, b, lr(l), its);
end

% Pick best learning rate: -- take minimizinng MSE over iterations
[minMSE, best_lr_idx] = min(sum(mse_lr));

% Display results:
disp('Best Learning Rate Found:'); lr_star = lr(best_lr_idx);
disp(lr_star);
%%
figure();
plot(lr,sum(mse_lr)/10000);
title('MSE vs Learning Rate'); xlabel('Learning Rate'); ylabel('MSE');
hold on; plot(lr(best_lr_idx),(mse_lr(best_lr_idx))/10,'*r')
legend('lr vs MSE','Best point')

%% Show network weights for the best learning rate available:
figure;
for i = 1:26
    best_w(i,:,:) = reshape(w_lr(i,:,best_lr_idx), [28, 28]);
    subplot(x_ax,y_ax,i);
    imagesc(squeeze(best_w(i,:,:))); colormap(gray);
    %   imshow(squeeze(best_w(i,:,:))*10); %scaled for better visual
end
sgtitle('Finalized Weights of Neuron from 1 to 26');

%% Part C:
lr_low = 1/100 * lr_star;
lr_high = 100 * lr_star;

lr_c = [lr_low, lr_star, lr_high];

% Do the training for given set of learning rates
for k = 1:3
    [w_lrc(:,:,k),b_lrc(:,k),mse_lrc(:,k)] = ...
        learn_weights(trainims,trainlbls_onehot,w, b, lr_c(k), its);
end

%% Plot MSE curves:
```

```
figure(); xlabel('Iterations'); ylabel('MSE values'); hold on;
plot(1:its,mse_lrc(:,1)); plot(1:its,mse_lrc(:,2));
plot(1:its,mse_lrc(:,3));
legend('lr-low', 'lr-best', 'lr-high'); grid on;
title('MSE evolution over iterations')
%%
% Display total average mean squared error:
averages = 1/10000 * sum(mse_lrc);
disp('Average MSE of lr_low:');
disp(averages(1));

disp('Average MSE of lr_best:');
disp(averages(2));

disp('Average MSE of lr_high:');
disp(averages(3));

%% Part D:
% Do the training for given set of learning rates.
% Using the weights from part C: 'w_lrc'
w_dlow = w_lrc(:,:,1); w_dstar = w_lrc(:,:,2); w_dhigh = w_lrc(:,:,3);
b_dlow = b_lrc(:,1); b_dstar = b_lrc(:,2); b_dhigh = b_lrc(:,3);

% Calculate the output given the input
out_dlow = sigmoid((w_dlow*testims)-repmat(b_dlow, [1,1300]));
out_dstar = sigmoid((w_dstar*testims)-repmat(b_dstar, [1,1300]));
out_dhigh = sigmoid((w_dhigh*testims)-repmat(b_dhigh, [1,1300]));

% Winner takes all: May the best neuron win: Get winning idx per each
% sample.
[vals_dlow, indices_dlow] = max(out_dlow);
[vals_dstar, indices_dstar] = max(out_dstar);
[vals_dhigh, indices_dhigh] = max(out_dhigh);

% Now do check classification error using original labels, one hot enode
% not needed as comparison from indices are easy:
perc_corr_dlow = (1 - sum(abs(indices_dlow ~= testlbls')) / 1300);
perc_corr_dstar = (1 - sum(abs(indices_dstar ~= testlbls')) / 1300);
perc_corr_dhigh = (1 - sum(abs(indices_dhigh ~= testlbls')) / 1300);

% Display the results:
disp('Percentage of output correct for lr: 0.0005 - Low Learning Rate');
disp(perc_corr_dlow)
disp('*************')
disp('Percentage of output correct for lr: 0.05 - Best Learning Rate');
disp(perc_corr_dstar)
disp('*************')
disp('Percentage of output correct for lr: 5 - High Learning Rate');
disp(perc_corr_dhigh)


%% Misc Functions Q3: - Used to simplify code and promote re-use.
function out = sigmoid(z)
% Calculates sigmoid function given a matrix value set.
% Used to simplify code in Question 3 - Part B.
out = 1./(1+exp(-z));
end
function [w,b] = init_with_gauss(mu, sigma, layerSize, imSize_x, imSize_y)
% Initialize weights and bias by gaussian distribution, given layer size
% Customized to convert 2D image connections, flattening them to vectors
w = normrnd(mu, sigma, [layerSize,imSize_x*imSize_y]); % includes bias term
b = normrnd(mu, sigma, [layerSize, 1]);
end
function [encoded] = encode_onehot(labels)
% Encode 1-26 labels to one hot version.
encoded = zeros(size(labels,1),26);
for i=1:size(labels,1)
    encoded(i,labels(i)) = 1;
end
end
function [w, b, mse] = ...
```

```
    learn_weights(images_flat,labels_onehot,wb_init,b_init,lr, iterations)
% This function does the learning of the weights, bias and MSE and also
% updates weights. Specifically written to ease read in Parts B,C,D.

w = wb_init; b = b_init; mse = zeros(1,iterations);
for it = 1:iterations
    % Pick a random sample: - Already normalized. Flattened to ease my code
    % Image converted to double to allow matrix calculations with w.
    idx = ceil(size(images_flat,2)*rand);
    x = images_flat(:,idx); % flattened img selected, with bias added

    % Do feedforward operation with sigmoid:
    output = sigmoid(w*x-b)';
    error = (labels_onehot(idx,:)-output);

    % MSE loss at this stage
    mse(it) = 1/26 .* error * error';

    % Do weight update: - Gradient Descent w/ sigmoid:
    dw = -2*(error .* ((output) .* (1-(output)))) .* x;
    db = -2*(error .* ((output) .* (1-(output))));

    % Optimizes the parameters, -- Namely, update w and b
    w = w - lr*dw';
    b = b - lr*db';

    % Returns MSE for 10000 iterations and final w and b
end
end
```

Assignment 1 – Main Code:

This part only has the main code available. If you like to see the code in its entirety, please check the code submission separately.

```
function hasanemre_erdemoglu_21401462_hw1(question)
clc
close all

switch question
    case '1'
        disp('Question 1:')
        % This question normally does not have a MATLAB counterpart.
        question1()
    case '2'
        disp('Question 2:')
        % question 2 code goes here
        question2()

    case '3'
        disp('Question 3:')
        % question 3 code goes here
        question3()
end
end
```