Student Name: Jin Hoontae (A0243155L) **All the scripts are in the Appendix section**

Question 1

The local induced field for the signal-flow perceptron graph can be denoted as:

$$v = \sum_{k=1}^{m} w_k x_k + b \qquad [1]$$

To simplify the computation task later, $\sum_{k=1}^{m} w_k x_k$ can be denoted as a.

$$\sum_{k=1}^{m} w_k x_k = w_1x_1 + w_2x_2 + \cdots + w_mx_m = a$$

Hence, Eq1 can be written as:

$$v = a + b \qquad [2]$$

The assumption to be made to process the data is:

$$\varphi = \begin{cases} Class\ 1\ (C_1)\ if\ \varphi(v) > \xi \\ Class\ 2\ (C_2)\ if\ \varphi(v) < \xi \end{cases}$$

Each class is to be classified as 0 (Class 2) or 1 (Class 1).

Three activation functions are to be investigated whether they are capable of producing a hyper-plane. *The decision boundary decision can be determined by computing v in each activation function only when $\varphi(v) = \xi$.

1) Logistic function: $\varphi(v) = \frac{1}{1-e^{-v}}$

$$\varphi(v) = \frac{1}{1-e^{-v}} \to \xi = \frac{1}{1-e^{-v}} \to 1 - e^{-v} = \frac{1}{\xi} \to e^{-v} = 1 - \frac{1}{\xi} \to -v = \ln\left(1 - \frac{1}{\xi}\right) \to v = -\ln\left(\frac{\xi - 1}{\xi}\right)$$

Hence, we get:

$$v = -\ln\left(\frac{\xi-1}{\xi}\right) \qquad [3]$$

Substituting Eq 3 into Eq 2, we obtain:

$$-\ln\left(\frac{\xi-1}{\xi}\right) = a + b$$

Since $a = -\ln\left(\frac{\xi-1}{\xi}\right) - b$ and is constant, the resulting boundary decision is a hyper-plane.

2) Bell-shaped Gaussian function: $\varphi(v) = e^{\frac{(v-m)^2}{2}}$

$$\varphi(v) = e^{\frac{(v-m)^2}{2}} \to \xi = e^{\frac{(v-m)^2}{2}} \to \ln(\xi) = \frac{(v-m)^2}{2} \to 2\ln(\xi) = (v-m)^2 \to \pm\sqrt{2\ln(\xi)} = v - m$$

Hence, we get:

$$v = \pm\sqrt{2\ln(\xi)} + m \qquad [4]$$

Substituting Eq 4 into Eq 2, we obtain:

$$\pm\sqrt{2\ln(\xi)} + m = a + b$$

Unlike the previous logistic function, since $a = \pm\sqrt{2\ln(\xi)} + m - b$ and is not constant due to $\pm$ unless m is $\mp\sqrt{2\ln(\xi)}$, there exist two planes, which is against the nature of the single-flow perceptron.

3) Softsign function: $\boldsymbol{\varphi(v) = \frac{v}{1+|v|}}$

$$\varphi(v) = \frac{v}{1+|v|} \rightarrow \xi = \frac{v}{1+|v|} \rightarrow \xi(1+\mathrm{v}) = \mathrm{v} \rightarrow \xi = \mathrm{v}(1-\xi) \rightarrow \mathrm{v} = \frac{\xi}{(1-\xi)}$$

Hence, we get:

$$v = \frac{\xi}{(1-\xi)} \qquad [5]$$

Substituting Eq 5 into Eq 2, we obtain:

$$\frac{\xi}{(1-\xi)} = a + b$$

Since $a = \frac{\xi}{(1-\xi)} - b$ and is constant, the resulting boundary decision is a hyper-plane.

Question 2

To prove that the XOR problem is not linearly separable, *Proof by Contradiction* approach is needed. Hence, we need to make the following assumption: The XOR problem with the given data is linearly separable. Based on the assumption, we can obtain:

$$\varphi = \begin{cases} 1\ if\ v > 0 \\ 0\ if\ v < 0 \end{cases}$$

where φ is an activation function and $v = \sum_{k=1}^{2} w_k x_k + b = w_1 x_1 + w_2 x_2 + b$

Feeding the data given in the *Truth Table of XOR* into the equation, we obtain:

$$\varphi = \begin{cases} 0\ if\ 0w_1 + 0w_2 + b < 0 \\ 1\ if\ 1w_1 + 0w_2 + b > 0 \\ 1\ if\ 0w_1 + 1w_2 + b > 0 \\ 0\ if 1w_1 + 1w_2 + b < 0 \end{cases}$$

This can be re-written as:

$$\varphi = \begin{cases} 0\ if\ b < 0 \\ 1\ if\ w_1 + b > 0 \\ 1\ if\ w_2 + b > 0 \\ 0\ if\ w_1 + w_2 + b < 0 \end{cases} \qquad [1]$$

The error can be found when adding two sub-equations in Eq 1, which means:

When 1st and 4th sub-equations are added together, we get:

$$1w_1 + 1w_2 + 2b < 0 \qquad [2]$$

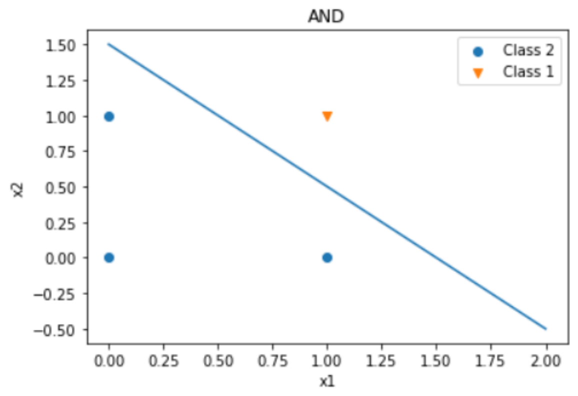
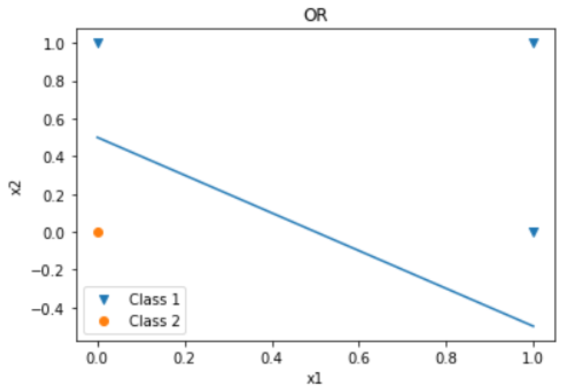
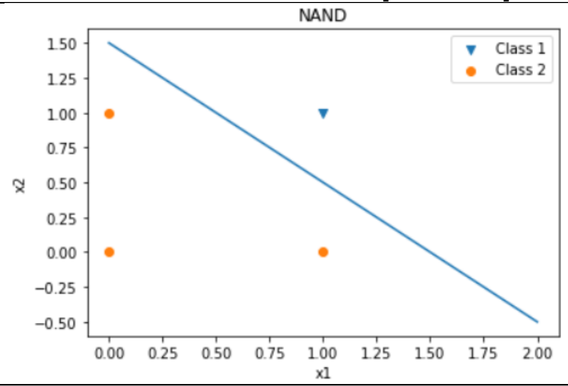
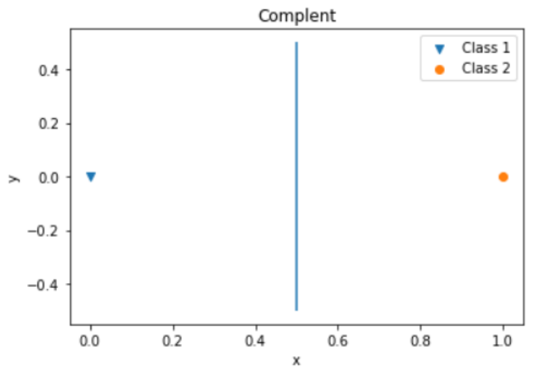
Likewise, when 2nd and 3rd sub-equations are added together, we get:

$$1w_1 + 1w_2 + 2b > 0 \qquad [3]$$

It shows that both LHS and RHS equations (Eq 2 and Eq 3, respectively) are the same even though the conditions ($LHS < 0 \text{ in Eq2 } and\ RHS > 0 \text{ in Eq3}$) are different. Hence, *Proof by Contradiction* proves that the assumption (The XOR problem with the given data is linearly separable) is not correct and not linearly separable.

Question 3

a) Demonstrate the implementation of the logic functions AND, OR, COMPLEMENT and NAND with selection of weights by off-line calculations.

x1 + x2 -1.5 = 0 and w = [-1.5, 1, 1]	x1 + x2 – 0.5 = 0 and w = [-0.5, 1, 1]
	
-x1 - x2 + 1.5 = 0 and w = [1.5, -1, -1]	-0.5x + 0.5 = 0 and w = [0.5, -0.5]
	

b) Demonstrate the implementation of the logic functions AND, OR, COMPLEMENT and NAND with selection of weights by learning procedure. Suppose initial weights are chosen randomly and learning rate is 1.0. Plot out the trajectories of the weights for each case. Compare the results with those obtained in (a). Try other learning rates, and report your observations with different learning rates.

b-1) Default Learning Rate = 1.0 was initially used to produce the plots. A row matrix composed of 1s was added to the first row of the input data, and a random value of bias was also added to the first index of the weight matrix to simplify the computation process. **Fig 1** illustrates how the logic functions are plotted and how the values (weights, bias) converge after a number of iterations.

Even though the graphs do not look exactly the same as the ones created by the off-line calculations, the written algorithm was able to draw a boundary decision line successfully by dividing each class clearly as shown in the Figure. As for the values of the weights and biases, they eventually start to converge after enough iteration.

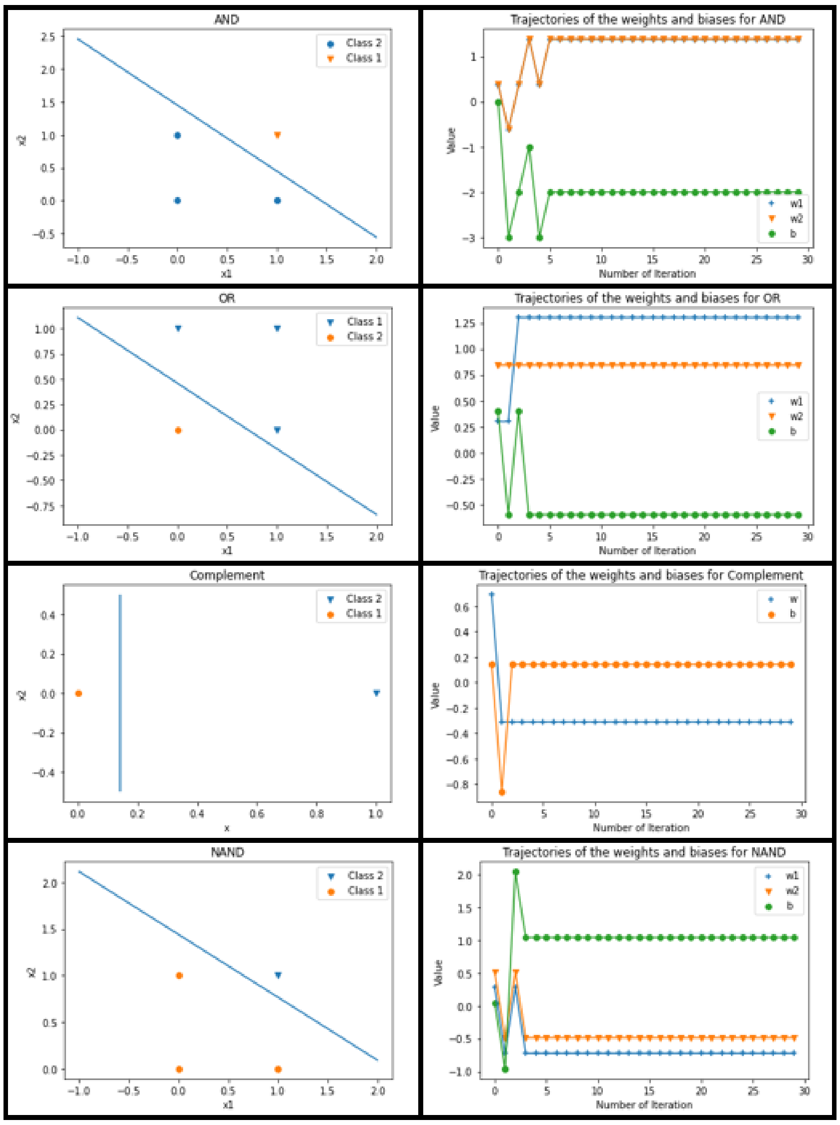


Figure 1. Graphs of the logic functions and trajectories of the weights and biases for each function

b-2) Four different learning values (0.1, 0.3, 0.5 and 0.7) were attempted to examine how they would affect the convergence of the parameters. As can be seen in **Fig 2**, each learning rate produces different trajectory plots, and their convergence lines also differ depending on the learning rate value. Additionally, it was found that the first random values of weights and biases also significantly affected the convergence, meaning that if the initial random values were close enough to the optimal values for the convergence, it did not take a long iteration to arrive at the convergence line. In conclusion, learning rate value and the initial values of weights and biases are critical factors for the fast convergence computation in this specific question.

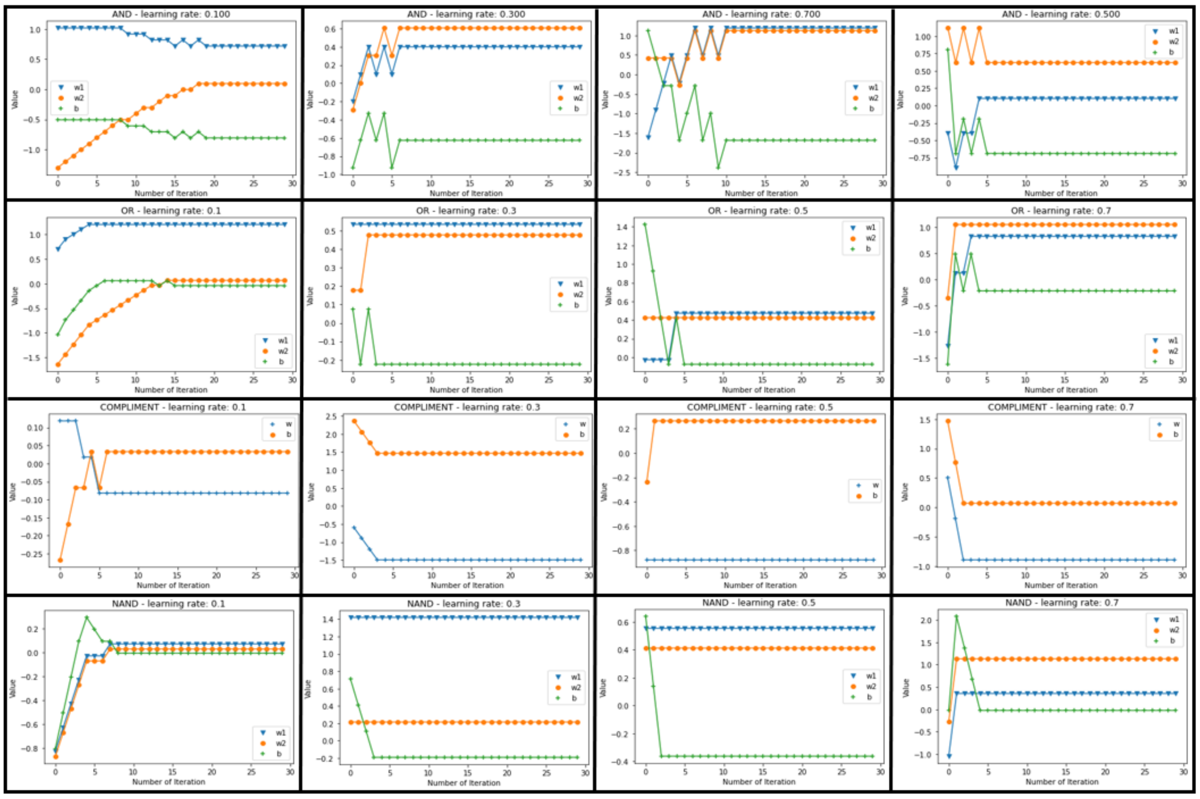


Figure 2. Trajectories of the weight and bias values for the functions with the learning rates (0.1, 0.3, 0.5 and 0.7)

c) What would happen if the perceptron is applied to implement the EXCLUSIVE OR function with selection of weights by learning procedure? Suppose initial weight is chosen randomly and learning rate is 1.0. Do the computer experiment and explain your finding.

As already demonstrated in Question 2, since the Exclusive OR function is not linearly separable with a single line, the weights and bias never converge but fluctuate constantly until the end of the iteration. The proof can be observed in **Figure 3**; that is, regardless of the number of iterations, the convergence would never be achieved due to its nature of being not linearly separable with one line.

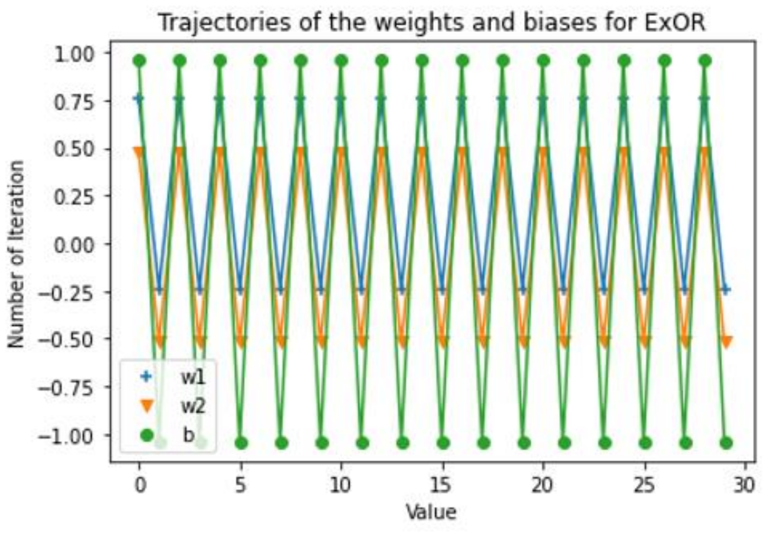


Figure 3. Weights and Bias trajectories for Exclusive OR function

Question 4

a) Find the solution of w and b using the standard linear least-squares (LLS) method. Plot out the fitting result.

Standard Linear Least Squares can be written as follows:

$$E(w) = \sum_{i=1}^{n} e(i)^2 = e^T e$$

$$e = d - Xw$$

where e is the error vector, d is the true output, X is the input data and w is the weight.

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial e}\frac{\partial e}{\partial w}$$

$$\frac{\partial E}{\partial e} = 2e^T \text{ and } \frac{\partial e}{\partial w} = -X$$

$$\frac{\partial E}{\partial w} = -2e^T X = 0$$

$$e^T X = (d - Xw)^T X = (d^T - w^T X^T)X = d^T X - w^T X^T X = 0$$

$$w^T X^T X = d^T X$$

$$X^T X w = X^T d$$

$$w = (X^T X)^{-1} X^T d \qquad [1]$$

With **Eq 1**, the linear regression can be drawn. Based on the calculation, the weight values obtained were 0.55543634 and 1.46995708. Hence, the line, *y=1.46995708x+0.55543634*, could be drawn as shown in **Fig 4**.

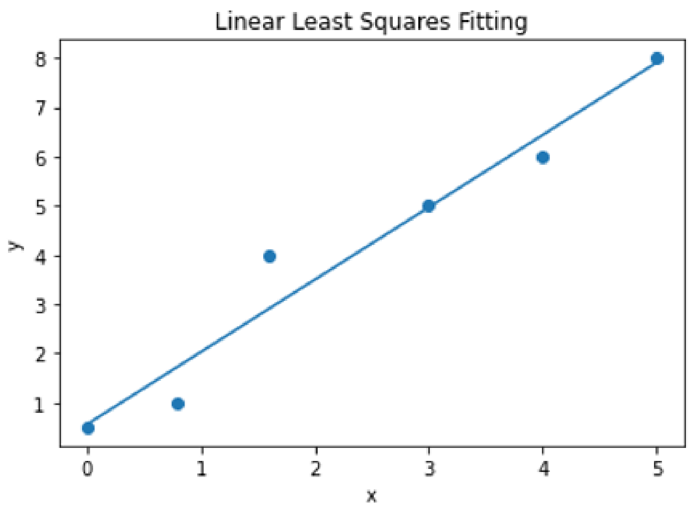
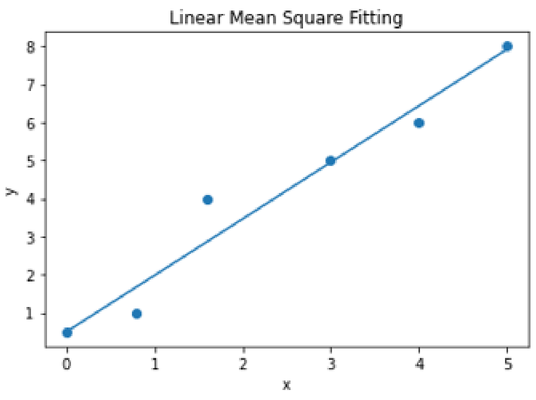


Figure 4. Linear Least Squares (LLS) Fitting

b) Suppose that initial weight is chosen randomly and learning rate is 0.01. Find the solution of w and b using the least-mean-square (LMS) algorithm for 100 epochs. Plot out the fitting result and the trajectories of the weights versus learning steps. Will the weights converge?

The weight values obtained using the LMS algorithm were 0.49035409 and 1.48468083, which were different from those obtained previously in **4-a**. However, it was observed that their values were extremely close. The line, *y=1.48468083x+0.49035409, is shown in **Fig 5**. It shows* a fairly accurate fitting line displaying the low error.

When 0.01 is used as a learning rate with an epoch of 100, as shown in the figure, they start to form a converging trend as the number of epochs increases.



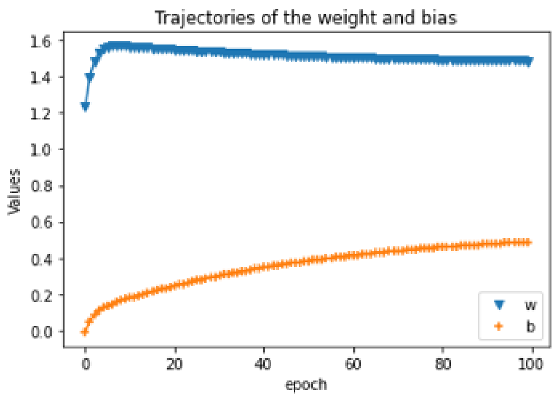


Figure 5. LMS Fitting (left) and Trajectories of the weight and bias (right)

c) Compare the results obtained by LLS and the LMS methods.

As discussed in **4-b**, each value of LLS and LMS was fairly close. This analysis could further be observed with a better visualization by plotting two lines. As shown in **Fig 6**, it looks as though the lines overlap each other almost completely, which explains that the LLS and LSM methods work great for this problem.

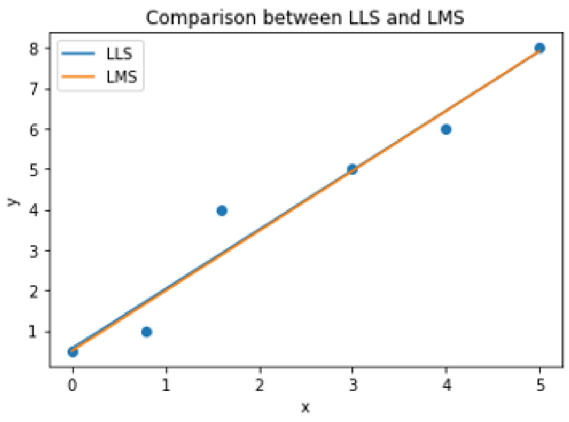


Figure 6. LLS and LMS Fitting Lines

d) Repeat the simulation study in b) with different learning rates, and explain your findings.

Four different learning rates (0.0001, 0.001, 0.1 and 1) were attempted to investigate how they would behave with respect to the number of epochs. **Fig 7** demonstrates how the value of a learning rate affects the computation of weights and biases in this question. When the learning rate is 0.0001, the weight and bias do not converge at all until the end of the iteration. Hence, it can be said that a greater number of epochs is required for the parameters (weight and bias) to converge when the learning rate is 0.0001 (extremely low). When the learning rate is 0.001, the parameters seem to converge slightly better when the final epoch is reached. The better convergence can be observed when the learning rate is 0.1. The complete failure of the convergence is found when 1.0 is used as the learning rate. This phenomenon occurs because the Taylor series only holds for sufficiently small learning rates.

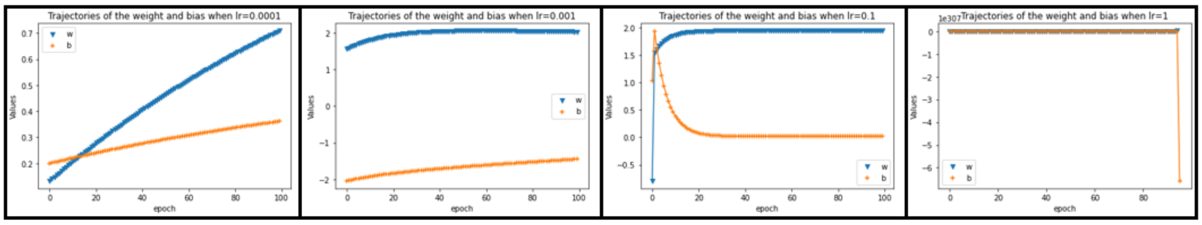


Figure 7. Trajectories of weights and bias with respect to different learning rates

Question 5

The weighted factors, $r(i)$, for each output error $e(i)$, can be expressed as a diagonal matrix:

$$R_{n\times n} = r(i) = \begin{bmatrix} r(1) & 0 & \cdots & 0 & 0 \\ 0 & r(2) & 0 & 0 & 0 \\ \vdots & 0 & \ddots & 0 & \vdots \\ 0 & 0 & 0 & r(n-1) & 0 \\ 0 & 0 & \cdots & 0 & r(n) \end{bmatrix}$$

The error function can be written as:

$$e(i) = d(i) - y(i) \rightarrow e = d - y$$

where d is the desired output ($d = [d(1)\ d(2)\ d(3) \cdots d(n)]^T$)

y is the computed output ($y = [y(1)\ y(2)\ y(3) \cdots y(n)]^T$)

Since $y(i) = w^T x(i) = x(i)^T w$, it can be expressed as:

$$y = \begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ y(i) \end{bmatrix} = \begin{bmatrix} x(1)^T w \\ x(2)^T w \\ \vdots \\ x(i)^T w \end{bmatrix} = \begin{bmatrix} x(1)^T \\ x(2)^T \\ \vdots \\ x(i)^T \end{bmatrix} w = Xw$$

where $X = \begin{bmatrix} x(1)^T \\ x(2)^T \\ \vdots \\ x(n)^T \end{bmatrix}$ (regression matrix)

Hence, we obtain the error function ($e = d - Xw$).

The given cost function J(w) is below:

$$J(w) = \frac{1}{2}\sum_{i=1}^{n} r^2(i)\, e(i)^2 + \frac{1}{2}\lambda||w||^2 = \frac{1}{2}\sum_{i=1}^{n} r^2(i)(d(i) - y(x(i)))^2 + \frac{1}{2}\lambda w^T w$$

$$J(w) = \frac{1}{2}(e^T R^T R e + \lambda w^T w) = \frac{1}{2}(e^T R^T R e + \lambda (X^{-1}(\mathrm{d} - \mathrm{e}))^T (X^{-1}(\mathrm{d} - \mathrm{e}))$$

$$\begin{aligned} w^T w = \left(X^{-1}\,(d-e)\right)^T\left(X^{-1}\,(d-e)\right) &= (X^{-1}d - X^{-1}e)^T(X^{-1}d - X^{-1}e) \\ &= \{(X^{-1}d)^T - (X^{-1}e)^T\}(X^{-1}d - X^{-1}e) = (d^T X^{-T} - e^T X^{-T})(X^{-1}d - X^{-1}e) \\ &= d^T X^{-T} X^{-1} d - d^T X^{-T} X^{-1} e - e^T X^{-T} X^{-1} d + e^T X^{-T} X^{-1} e \end{aligned}$$

Hence, the cost function can be written as:

$$J(w) = \frac{1}{2}(e^T R^T R e + \lambda\{d^T X^{-T} X^{-1} d - d^T X^{-T} X^{-1} e - e^T X^{-T} X^{-1} d + e^T X^{-T} X^{-1} e\})$$

$$\frac{\partial J}{\partial e} = \frac{1}{2}\left(2e^T R^T R + \lambda(-X^{-T} X^{-1} d - X^{-T} X^{-1} d + 2e^T X^{-T} X^{-1})\right) = e^T R^T R + \frac{1}{2}\lambda(-2X^{-T} X^{-1} d + 2e^T X^{-T} X^{-1})$$

$$\frac{\partial e}{\partial w} = -X$$

By chain rule, we obtain:

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial e}\frac{\partial e}{\partial w} = \{e^T R^T R + \lambda(-X^{-T} X^{-1} d + e^T X^{-T} X^{-1})\}(-X) = 0$$

Replacing e with $d - Xw$, we obtain:

$$\{(d - Xw)^T R^T R + \lambda(-X^{-T}X^{-1}d + (d - Xw)^T X^{-T}X^{-1})\}(-X) = 0$$

$$\{(d^T - w^T X^T)R^T R - \lambda X^{-T}X^{-1}d + \lambda(d^T - w^T X^T)X^{-T}X^{-1}\}(-X) = 0$$

$$(d^T R^T R - w^T X^T R^T R - \lambda X^{-T}X^{-1}d + \lambda d^T X^{-T}X^{-1} - \lambda w^T X^T X^{-T}X^{-1})(-X) = 0$$

$$-d^T R^T RX + w^T X^T R^T RX + \lambda X^{-T}X^{-1}dX - \lambda d^T X^{-T}X^{-1}X + \lambda w^T X^T X^{-T}X^{-1}X = 0$$

$$-d^T R^T RX + w^T X^T R^T RX + \lambda X^{-T}X^{-1}dX - \lambda d^T X^{-T} + \lambda w^T X^T X^{-T} = 0$$

$$w^T(X^T R^T RX + \lambda X^T X^{-T}) = (d^T R^T RX - \lambda X^{-T}X^{-1}dX + \lambda d^T X^{-T}) \qquad [1]$$

Transposing Eq1, we obtain:

$$(X^T R^T RX + \lambda X^T X^{-T})^T w = (d^T R^T RX - \lambda X^{-T}X^{-1}dX + \lambda d^T X^{-T})^T$$

$$(X^T R^T RX + \lambda)^T w = (d^T R^T RX - \lambda X^{-T}X^{-1}dX + \lambda d^T X^{-T})^T$$

$$(X^T R^T RX + \lambda)w = (X^T R^T Rd - \lambda X^T d^T X^{-T}X^{-1} + \lambda X^{-1}d)$$

$$w = (X^T R^T RX + \lambda)^{-1}(X^T R^T Rd - \lambda X^T d^T X^{-T}X^{-1} + \lambda X^{-1}d) \qquad [2]$$

Thus, the optimal parameter (w^*) is eq 2, and then the cost function $J(w)$ is minimized.

Appendix

Question 3-a)

```python
# Question 3
import numpy as np
import matplotlib.pyplot as plt
```

```python
#a)
# AND
x1_AND = np.array([0, 0, 1, 1]);
x2_AND = np.array([0, 1, 0, 1]);
y_AND = np.array([0, 0, 0, 1]);
x = np.arange(0, 3)
y = -x +1.5

plt.scatter(x1_AND[:3], x2_AND[:3])
plt.scatter(x1_AND[3], x2_AND[3], marker = 'v')
plt.legend(labels=["Class 2","Class 1"]) #Class 1 when y>=0 and Class 2 when y<0
plt.plot(x,y)
plt.title("AND")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()

# OR
x1_OR = np.array([0, 0, 1, 1]);
x2_OR = np.array([0, 1, 0, 1]);
y_OR = np.array([0, 1, 1, 1]);
x = np.arange(0, 2)
y = -x + 0.5

plt.scatter(x1_OR[1:], x2_OR[1:], marker = 'v')
plt.scatter(x1_OR[0], x2_OR[0])
plt.legend(labels=["Class 1","Class 2"]) #Class 1 when y>=0 and Class 2 when y<0
plt.plot(x,y)
plt.title("OR")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
```

```python
# Complement
x_Complement = np.array([0, 1]);
y_Complement = np.array([1, 0]);

plt.scatter(x_Complement[1], y_Complement[1], marker = 'v')
plt.scatter(x_Complement[0],y_Complement[1])
plt.legend(labels=["Class 2","Class 1"]) #Class 1 when y>=0 and Class 2 when y<0
plt.vlines(0.5,-0.5, 0.5)
plt.title("Complement")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()

# NAND
x1_NAND = np.array([0, 0, 1, 1]);
x2_NAND = np.array([0, 1, 0, 1]);
y_NAND = np.array([1, 1, 1, 0]);
x = np.arange(0, 3)
y = -x +1.5

plt.scatter(x1_NAND[3], x2_NAND[3], marker = 'v')
plt.scatter(x1_NAND[:3], x2_NAND[:3])
plt.legend(labels=["Class 2","Class 1"]) #Class 1 when y>=0 and Class 2 when y<0
plt.plot(x,y)
plt.title("NAND")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
```

Question 3-b)

```
#b)
#AND
x = np.array([[0, 0, 1, 1],[0, 1, 0, 1]]) # Input data
y_AND = np.array([0, 0, 0, 1]) # Output data

# The fixed input, x0, is set to be +1 and the weight, w0, is b.
x0 = np.ones((1,x.shape[1]))
x = np.concatenate((x0,x))
w = np.random.rand(1,x.shape[0]) # Bias in 1st column and Weight in 2nd, 3rd columns

# Tracking weights and biases for plotting
w1_track = []
w2_track = []
b_track = []

# Customised setup for running the algorithm
lr = 1.0 # Learning Rate
iteration = 30 # Number of iteration to update weight and bias values
i = 0

# Plotting the predicted boundary decision line
while i < iteration:
    # record weights
    w1_track.append(w[0,1])
    w2_track.append(w[0,2])
    b_track.append(w[0,0])

    v = np.dot(w, x)
    y = np.array(v>=0)
    e = y_AND - y

    #update
    w = w + lr * np.dot(e, x.transpose())
    i += 1

x = np.arange(-1,3)
line = -w[0,2]/w[0,1] * x - w[0,0]/w[0,1]
plt.scatter([0, 0, 1], [0, 1, 0])
plt.scatter([1], [1], marker = 'v')
plt.plot(x, line)
plt.title("AND")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()

num_iter = np.array(range(iteration))
plt.scatter(num_iter,w1_track, label = 'w1', marker = '+')
plt.plot(num_iter,w1_track)
plt.scatter(num_iter,w2_track,label = 'w2', marker = 'v')
plt.plot(num_iter,w2_track)
plt.scatter(num_iter,b_track, label = 'b', marker = 'o')
plt.plot(num_iter,b_track)
plt.title('Trajectories of the weights and biases for AND')
plt.xlabel("Number of Iteration")
plt.ylabel("Value")
plt.legend()
plt.show()
```

```python
#OR
x = np.array([[0, 0, 1, 1],[0, 1, 0, 1]]) # Input data
y_OR = np.array([0, 1, 1, 1]) # Output data
# The fixed input, x0, is set to be +1 and the weight, w0, is b.
x0 = np.ones((1,x.shape[1]))
x = np.concatenate((x0,x))
w = np.random.rand(1,x.shape[0]) # Weight

# Tracking weights and biases for plotting
w1_track = []
w2_track = []
b_track = []

# Customised setup for running the algorithm
lr = 1.0 # Learning Rate
iteration = 30 # Number of iteration to update weight and bias values
i = 0

# Plotting the predicted boundary decision line
while i < iteration:
    # record weights
    w1_track.append(w[0,1])
    w2_track.append(w[0,2])
    b_track.append(w[0,0])

    v = np.dot(w, x)
    y = np.array(v>=0)
    e = y_OR - y

    #update
    w = w + lr * np.dot(e, x.transpose())
    i += 1

x = np.arange(-1,3)
line = -w[0,2]/w[0,1] * x - w[0,0]/w[0,1]
plt.scatter([0, 1, 1], [1, 0, 1], marker = 'v')
plt.scatter([0], [0])
plt.plot(x, line)
plt.title("OR")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()

num_iter = np.array(range(iteration))
plt.scatter(num_iter,w1_track, label = 'w1', marker = '+')
plt.plot(num_iter,w1_track)
plt.scatter(num_iter,w2_track,label = 'w2', marker = 'v')
plt.plot(num_iter,w2_track)
plt.scatter(num_iter,b_track, label = 'b', marker = 'o')
plt.plot(num_iter,b_track)
plt.title('Trajectories of the weights and biases for OR')
plt.xlabel("Number of Iteration")
plt.ylabel("Value")
plt.legend()
plt.show()
```

```python
#NAND
x = np.array([[0, 0, 1, 1],[0, 1, 0, 1]]) # Input data
y_NAND = np.array([1, 1, 1, 0]) # Output data
# The fixed input, x0, is set to be +1 and the weight, w0, is b.
x0 = np.ones((1,x.shape[1]))
x = np.concatenate((x0,x))
w = np.random.rand(1,x.shape[0])
# Tracking weights and biases for plotting
w1_track = []
w2_track = []
b_track = []

# Customised setup for running the algorithm
lr = 1.0 # Learning Rate
iteration = 30 # Number of iteration to update weight and bias values
i = 0

# Plotting the predicted boundary decision line
while i < iteration:
    # record weights
    w1_track.append(w[0,1])
    w2_track.append(w[0,2])
    b_track.append(w[0,0])

    v = np.dot(w, x)
    y = np.array(v>=0)
    e = y_NAND - y

    #update
    w = w + lr * np.dot(e, x.transpose())
    i += 1

x = np.arange(-1,3)
line = -w[0,2]/w[0,1] * x - w[0,0]/w[0,1]
plt.scatter([0, 0, 1], [0, 1, 0], marker = 'v')
plt.scatter([1], [1])
plt.plot(x, line)
plt.title("NAND")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()

num_iter = np.array(range(iteration))
plt.scatter(num_iter,w1_track, label = 'w1', marker = '+')
plt.plot(num_iter,w1_track)
plt.scatter(num_iter,w2_track,label = 'w2', marker = 'v')
plt.plot(num_iter,w2_track)
plt.scatter(num_iter,b_track, label = 'b', marker = 'o')
plt.plot(num_iter,b_track)
plt.title('Trajectories of the weights and biases for NAND')
plt.xlabel("Number of Iteration")
plt.ylabel("Value")
plt.legend()
plt.show()
```

```
#Complement
x = np.array([[0,1]]) # Input data
y_Complement = np.array([[1,0]]) # Output data

# The fixed input, x0, is set to be +1 and the weight, w0, is b.
x0 = np.ones((1,x.shape[1]))
x = np.concatenate((x0,x))
w = np.random.rand(1,x.shape[0])


# Tracking weights and biases for plotting
w_track = []
b_track = []

# Customised setup for running the algorithm
lr = 1.0 # Learning Rate
iteration = 30 # Number of iteration to update weight and bias values
i = 0

# Plotting the predicted boundary decision line
while i < iteration:
    # record weights
    w_track.append(w[0,1])
    b_track.append(w[0,0])

    v = np.dot(w, x)
    y = np.array(v>0, dtype= float)
    e = y_Complement - y

    #update
    w = w + lr * np.dot(e, x.T)
    i += 1

plt.scatter([0], [0], marker = 'v')
plt.scatter([1], [0])
plt.vlines(w[0,0], -0.5,0.5)
plt.title("Complement")
plt.xlabel("x")
plt.ylabel("x2")
plt.show()

num_iter = np.array(range(iteration))
plt.scatter(num_iter,w_track, label = 'w', marker = '+')
plt.plot(num_iter,w_track)
plt.scatter(num_iter,b_track, label = 'b', marker = 'o')
plt.plot(num_iter,b_track)
plt.title('Trajectories of the weights and biases for Complement')
plt.xlabel("Number of Iteration")
plt.ylabel("Value")
plt.legend()
plt.show()
```

```python
#b) Different Learning Rates (AND)
iteration = 30
lr_multi = [0.1, 0.3, 0.5, 0.7]

x = np.array([[0, 0, 1, 1],[0, 1, 0, 1]]) # Input data
y_AND = np.array([0, 0, 0, 1]) # Output data
# The fixed input, x0, is set to be +1 and the weight, w0, is b.
x0 = np.ones((1,x.shape[1]))
x = np.concatenate((x0,x))
# Tracking weights and biases for plotting
w1_track = np.zeros([len(lr_multi),iteration])
w2_track = np.zeros([len(lr_multi),iteration])
b_track = np.zeros([len(lr_multi),iteration])

# Plotting the predicted boundary decision line
for num, lr_val in enumerate(lr_multi):
    i = 0
    w = np.random.randn(1,x.shape[0])
    while i < iteration:
        # record weights
        w1_track[num, i] = (w[0,1])
        w2_track[num, i] = (w[0,2])
        b_track[num, i] = (w[0,0])

        v = np.dot(w, x)
        y = np.array(v>0)
        error = y_AND - y
        #update
        w = w + lr_val * np.dot(error, x.T)
        i += 1
num_iter = np.array(range(iteration))

for i, lr in enumerate(lr_multi):
    plt.scatter(num_iter,w1_track[i, :], label = 'w1', marker = 'v')
    plt.plot(num_iter,w1_track[i, :])
    plt.scatter(num_iter,w2_track[i, :],label = 'w2', marker = 'o')
    plt.plot(num_iter,w2_track[i, :])
    plt.scatter(num_iter,b_track[i, :], label = 'b', marker = '+')
    plt.plot(num_iter,b_track[i, :])
    plt.title("AND - learning rate: %1.3f" % lr)
    plt.xlabel("Number of Iteration")
    plt.ylabel("Value")
    plt.legend()
    plt.show()
```

```
#b) Different Learning Rates (OR)
iteration = 30
lr_multi = [0.1, 0.3, 0.5, 0.7]

x = np.array([[0, 0, 1, 1],[0, 1, 0, 1]]) # Input data
y_OR = np.array([0, 1, 1, 1]) # Output data
# The fixed input, x0, is set to be +1 and the weight, w0, is b.
x0 = np.ones((1,x.shape[1]))
x = np.concatenate((x0,x))
# Tracking weights and biases for plotting
w1_track = np.zeros([len(lr_multi),iteration])
w2_track = np.zeros([len(lr_multi),iteration])
b_track = np.zeros([len(lr_multi),iteration])

# Plotting the predicted boundary decision line
for num, lr_val in enumerate(lr_multi):
    i = 0
    w = np.random.randn(1,x.shape[0])
    while i < iteration:
        # record weights
        w1_track[num, i] = (w[0,1])
        w2_track[num, i] = (w[0,2])
        b_track[num, i] = (w[0,0])

        v = np.dot(w, x)
        y = np.array(v>0)
        error = y_OR - y
        #update
        w = w + lr_val * np.dot(error, x.T)
        i += 1
num_iter = np.array(range(iteration))

for i, lr in enumerate(lr_multi):
    plt.scatter(num_iter,w1_track[i, :], label = 'w1', marker = 'v')
    plt.plot(num_iter,w1_track[i, :])
    plt.scatter(num_iter,w2_track[i, :],label = 'w2', marker = 'o')
    plt.plot(num_iter,w2_track[i, :])
    plt.scatter(num_iter,b_track[i, :], label = 'b', marker = '+')
    plt.plot(num_iter,b_track[i, :])
    plt.title("OR - learning rate: %1.1f" % lr)
    plt.xlabel("Number of Iteration")
    plt.ylabel("Value")
    plt.legend()
    plt.show()
```

```python
#b) Different Learning Rates (COMPLIMENT)
iteration = 30
lr_multi = [0.1, 0.3, 0.5, 0.7]

x = np.array([[0,1]]) # Input data
y_Complement = np.array([[1,0]]) # Output data

# The fixed input, x0, is set to be +1 and the weight, w0, is b.
x0 = np.ones((1,x.shape[1]))
x = np.concatenate((x0,x))
w = np.random.rand(1,x.shape[0])


# Tracking weights and biases for plotting
w_track = np.zeros([len(lr_multi),iteration])
b_track = np.zeros([len(lr_multi),iteration])

# Plotting the predicted boundary decision line
for num, lr_val in enumerate(lr_multi):
    i = 0
    w = np.random.randn(1,x.shape[0])
    while i < iteration:
        w_track[num,i] = (w[0,1])
        b_track[num,i] = (w[0,0])

        v = np.dot(w, x)
        y = np.array(v>0, dtype= float)
        e = y_Complement - y

        #update
        w = w + lr_val * np.dot(e, x.T)
        i += 1
num_iter = np.array(range(iteration))
for i, lr in enumerate(lr_multi):
    num_iter = np.array(range(iteration))
    plt.scatter(num_iter,w_track[i,:], label = 'w', marker = '+')
    plt.plot(num_iter,w_track[i,:])
    plt.scatter(num_iter,b_track[i,:], label = 'b', marker = 'o')
    plt.plot(num_iter,b_track[i,:])
    plt.title("COMPLIMENT - learning rate: %1.1f" % lr)
    plt.xlabel("Number of Iteration")
    plt.ylabel("Value")
    plt.legend()
    plt.show()
```

```python
#b) Different Learning Rates (NAND)
iteration = 30
lr_multi = [0.1, 0.3, 0.5, 0.7]

x = np.array([[0, 0, 1, 1],[0, 1, 0, 1]]) # Input data
y_NAND = np.array([0, 1, 1, 1]) # Output data
# The fixed input, x0, is set to be +1 and the weight, w0, is b.
x0 = np.ones((1,x.shape[1]))
x = np.concatenate((x0,x))
# Tracking weights and biases for plotting
w1_track = np.zeros([len(lr_multi),iteration])
w2_track = np.zeros([len(lr_multi),iteration])
b_track = np.zeros([len(lr_multi),iteration])

# Plotting the predicted boundary decision line
for num, lr_val in enumerate(lr_multi):
    i = 0
    w = np.random.randn(1,x.shape[0])
    while i < iteration:
        # record weights
        w1_track[num, i] = (w[0,1])
        w2_track[num, i] = (w[0,2])
        b_track[num, i] = (w[0,0])

        v = np.dot(w, x)
        y = np.array(v>0)
        error = y_OR - y
        #update
        w = w + lr_val * np.dot(error, x.T)
        i += 1
num_iter = np.array(range(iteration))

for i, lr in enumerate(lr_multi):
    plt.scatter(num_iter,w1_track[i, :], label = 'w1', marker = 'v')
    plt.plot(num_iter,w1_track[i, :])
    plt.scatter(num_iter,w2_track[i, :],label = 'w2', marker = 'o')
    plt.plot(num_iter,w2_track[i, :])
    plt.scatter(num_iter,b_track[i, :], label = 'b', marker = '+')
    plt.plot(num_iter,b_track[i, :])
    plt.title("NAND - learning rate: %1.1f" % lr)
    plt.xlabel("Number of Iteration")
    plt.ylabel("Value")
    plt.legend()
    plt.show()
```

Question 3-c)

```python
#c)
#NAND
x = np.array([[0, 0, 1, 1],[0, 1, 0, 1]]) # Input data
y_ExOR = np.array([0, 1, 1, 0]) # Output data
# The fixed input, x0, is set to be +1 and the weight, w0, is b.
x0 = np.ones((1,x.shape[1]))
x = np.concatenate((x0,x))
w = np.random.rand(1,x.shape[0])
# Tracking weights and biases for plotting
w1_track = []
w2_track = []
b_track = []

# Customised setup for running the algorithm
lr = 1.0 # Learning Rate
iteration = 30 # Number of iteration to update weight and bias values
i = 0

# Plotting the predicted boundary decision line
while i < iteration:
    # record weights
    w1_track.append(w[0,1])
    w2_track.append(w[0,2])
    b_track.append(w[0,0])

    v = np.dot(w, x)
    y = np.array(v>=0)
    e = y_ExOR - y

    #update
    w = w + lr * np.dot(e, x.transpose())
    i += 1

num_iter = np.array(range(iteration))
plt.scatter(num_iter,w1_track, label = 'w1', marker = '+')
plt.plot(num_iter,w1_track)
plt.scatter(num_iter,w2_track,label = 'w2', marker = 'v')
plt.plot(num_iter,w2_track)
plt.scatter(num_iter,b_track, label = 'b', marker = 'o')
plt.plot(num_iter,b_track)
plt.title('Trajectories of the weights and biases for ExOR')
plt.xlabel("Value")
plt.ylabel("Number of Iteration")
plt.legend()
plt.show()
```

Question 4-a)

```
# Question 4
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt
```

```
#a)
x = np.array([[1, 1, 1, 1, 1, 1], [0, 0.8, 1.6, 3, 4.0, 5.0]]) # Input data
d = np.array([0.5, 1, 4, 5, 6, 8]) # True output data
x = x.T
w_LLS = np.dot(inv(np.dot(x.T,x)),np.dot(x.T,d))

i = np.arange(0, 6)
j = w_LLS[1]*i + w_LLS[0] # Predicted output data
plt.scatter(x[:,1], d)
plt.plot(i, j)
plt.title("Linear Least Squares Fitting")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
print(w_LLS)
```

Question 4-b)

```
#b)
#b-1) Fitting Graph
lr = 0.01
epochs = 100
w_LMS = np.random.randn(1,x.shape[1]).flatten()
w_track = []
b_track = []

for i in range(epochs):
    w_track.append(w_LMS[1])
    b_track.append(w_LMS[0])
    for j, label in enumerate(d):
        e = d[j] - np.dot(x[j, :],w_LMS)
        w_LMS = w_LMS + lr * e * x[j]

i = np.arange(0, 6)
j = w_LMS[1]*i + w_LMS[0]
plt.scatter(x[:,1], d)
plt.plot(i, j)
plt.title("Linear Mean Square Fitting")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
print(w_LMS)

#b-2) Trajectories
epc = np.array(range(epochs))
plt.scatter(epc,w_track, label = 'w', marker = 'v')
plt.plot(epc,w_track)
plt.scatter(epc,b_track, label = 'b', marker = '+')
plt.plot(epc,b_track)
plt.title('Trajectories of the weight and bias')
plt.xlabel("epoch")
plt.ylabel("Values")
plt.legend()
plt.show()
```

Question 4-c)

```
#c)
i = np.arange(0, 6)
LLS = w_LLS[1]*i + w_LLS[0]
LMS = w_LMS[1]*i + w_LMS[0]
plt.scatter(x[:,1], d)
plt.plot(i, LLS)
plt.plot(i, LMS)
plt.legend(['LLS','LMS'])
plt.title("Comparison between LLS and LMS")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

Question 4-d)

```
#d)
#b-1) Fitting Graph
lr_multi = [0.0001, 0.001, 0.1, 1]
epochs = 100
w_track = np.zeros([len(lr_multi),epochs])
b_track = np.zeros([len(lr_multi),epochs])

for order, lr_new in enumerate(lr_multi):
    w_LMS = np.random.randn(1,x.shape[1]).flatten()
    for i in range(epochs):
        w_track[order, i] = w_LMS[1]
        b_track[order, i] = w_LMS[0]
        for j, label in enumerate(d):
            e = d[j] - np.dot(x[j, :],w_LMS)
            w_LMS = w_LMS + lr_new*e*x[j]


#b-2) Trajectories
epc = np.array(range(epochs))
for i, lr in enumerate(lr_multi):
    plt.scatter(epc,w_track[i,:], label = 'w', marker = 'v')
    plt.plot(epc,w_track[i,:])
    plt.scatter(epc,b_track[i,:], label = 'b', marker = '+')
    plt.plot(epc,b_track[i,:])
    plt.title("Trajectories of the weight and bias when lr=%lf" % lr)
    plt.xlabel("epoch")
    plt.ylabel("Values")
    plt.legend()
    plt.show()
```