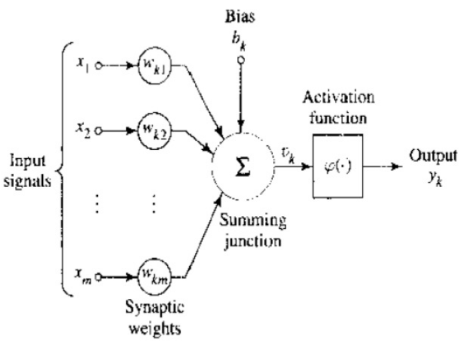
EE5904/ME5404 Neural Networks: Homework #1
XU NUO, A0313771H

Q1.



Consider the signal-flow graph of the perceptron shown in the above figure. The activation function, $\varphi(v)$, where v is the induced local field, can be designed by the user. If the activation function is chosen as hard limiter (i.e. step function), then it becomes the classical perceptron, and the decision boundary is shown to be a hyperplane. In this problem, let's explore other choices of the activation function, and its effect on the decision boundary. Let's assume that the classification decision made by the perceptron is simply a threshold defined as follows:

Observation vector $x = [x_1 \ x_2 \ \ldots . x_m]^T$ *belongs to class* C_1 *if the output* $y > \xi$, *where* ξ *is a user-defined threshold; otherwise,* x *belongs to class* C_2.

Consider the following three choices of activation function:

1) The activation function is the logistic function: $\varphi(v) = \frac{1}{1+e^{-v}}$;
2) The activation function is the Bell-shaped Gaussian function: $\varphi(v) = e^{-\frac{(v-m)^2}{2}}$;
3) The activation function is the Softsign function: $\varphi(v) = \frac{v}{1+|v|}$.

For each case, investigate whether the resulting decision boundary is a hyperplane or not.

A1.

v can be represented linearly by x since $x = [x_1 \ x_2 \ \ldots . x_m]^T$ and $v = w^T x + b$, so whether the decision boundary is a hyperplane or not fully depends on the relationship between the activation function $\varphi(v)$ and threshold ξ. In other words, the shape of the decision boundary can be decided merely by investigating the equation of the activation function $\varphi(v) = \xi$.

1) The logistic function $\varphi(v) = \frac{1}{1+e^{-v}}$ is a monotonically increasing sigmoid function mapping v into the range $(0,1)$. The decision rule is $\varphi(v) = \frac{1}{1+e^{-v}} > \xi$, then we rearrange it to be

$$v > \ln\left(\frac{\xi}{1-\xi}\right)$$

which clearly is a linear inequality with respect to v. The resulting decision boundary remains a hyperplane since v is a linear function of x.

2) The Bell-shaped Gaussian function $\varphi(v) = e^{-\frac{(v-m)^2}{2}}$ is symmetric about $v = m$, mapping v into the range $(0,1)$. The decision rule is $\varphi(v) = e^{-\frac{(v-m)^2}{2}} > \xi$, then we rearrange it to be

$$(v-m)^2 < -2\ln\xi$$

which is a quadratic inequality with respect to v. Hence the resulting decision boundary is not a hyperplane.

3) The Softsign function $\varphi(v) = \frac{v}{1+|v|}$ is a monotonically increasing function mapping v into the range $(-1,1)$. The decision rule is $\varphi(v) = \frac{v}{1+|v|} > \xi$, then we rearrange it to be

$$v > \frac{\xi}{1-\xi} \qquad for\, v > 0, \xi \geq 0$$

$$v > \frac{\xi}{\xi+1} \qquad for\, v \leq 0, \xi \leq 0$$

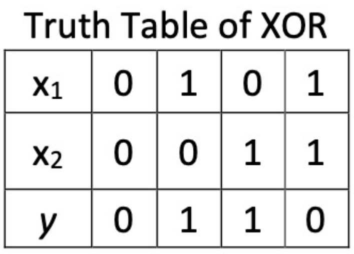
From the inequalities, we can discuss two cases:

If $\xi = 0$, for every $v > 0$, there is $\varphi(v) > 0$; for every $v \leq 0$, there is $\varphi(v) \leq 0$. Then $v = w^T x + b = 0$ is the resulting decision boundary, which apparently remains a hyperplane.

If $\xi \neq 0$, when $v \geq 0$ and $v < 0$, the values of the decision boundary are $\frac{\xi}{1-\xi}$ and $\frac{\xi}{\xi+1}$ respectively. The decision boundary is piecewise-defined, meaning it cannot be represented by a single hyperplane equation. In this case, the resulting decision boundary is not a hyperplane.

Q2.

Consider the logic function, EXCLUSIVE OR (XOR).



Truth Table of XOR

x_1	0	1	0	1
x_2	0	0	1	1
y	0	1	1	0

It is well known that the XOR problem is not linearly separable. It seems obvious by visual checking, which however cannot be accepted as mathematical proof. Therefore, please supply a rigorous mathematical proof for this statement.

A2.

Assume that the XOR problem is linearly separable. That is, there exists a linear decision boundary of the form $v = w^T x + b$ such that we can separate the two classed using a threshold ξ. We set observation vector $\mathrm{x} = [\mathrm{x}_1 \ \mathrm{x}_2]^{\mathrm{T}}$ belongs to class C_1(i.e., $y = 0$) if the induced local field $v < \xi$, where ξ is a threshold waiting to be defined; otherwise, x belongs to class C_2(i.e., $y = 1$). Then we should get 4 inequalities as follows:

$$\begin{cases} 0 \times w_1 + 0 \times w_2 + b < \xi & (1) \\ 1 \times w_1 + 0 \times w_2 + b \geq \xi & (2) \\ 0 \times w_1 + 1 \times w_2 + b \geq \xi & (3) \\ 1 \times w_1 + 1 \times w_2 + b < \xi & (4) \end{cases}$$

Summing (2) and (3) gives:

$$w_1 + w_2 + 2b \geq 2\xi \qquad (5)$$

Since (1) states $b < \xi$, we substitute that into (5), we can easily draw that

$$w_1 + w_2 + b > \xi \qquad (6)$$

Clearly, inequalities (4) and (6) contradict each other, which invalidates our initial assumption, proving the XOR problem is not linearly separable.

Q3.

The perceptron could be used to perform numerous logic functions, such as AND, OR, COMPLEMENT and NAND function, whose truth tables are tabulated as follows respectively.

x1	0	0	1	1
x2	0	1	0	1
y	0	0	0	1

AND

x1	0	0	1	1
x2	0	1	0	1
y	0	1	1	1

OR

x	0	1
y	1	0

COMPLEMENT

x1	0	0	1	1
x2	0	1	0	1
y	1	1	1	0

NAND

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

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 η 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.

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 η 1.0. Do the computer experiment and explain your finding.

A3.

a) As what we have discussed earlier, the mathematical representation of a perceptron is $y = \varphi(v) = \varphi(w^T x + b)$. We choose Step Function as the activation function for the logic functions listed. Let observation vector $\mathrm{x} = [\mathrm{x}_1\ \mathrm{x}_2]^\mathrm{T}$ belongs to class C_1(i.e., $y = 0$) if the induced local field $v \leq 0$, otherwise, x belongs to class C_2(i.e., $y = 1$).

1) AND:

Based on the truth table of AND, we can write the following 4 inequalities:

$$\begin{cases} 0 \times w_1 + 0 \times w_2 + b \leq 0 \\ 0 \times w_1 + 1 \times w_2 + b \leq 0 \\ 1 \times w_1 + 0 \times w_2 + b \leq 0 \\ 1 \times w_1 + 1 \times w_2 + b > 0 \end{cases}$$

My selection of weights and bias are $w_1 = 1 \quad w_2 = 1 \quad b = -1.5$. I choose these values because they ensure that the perceptron outputs the correct AND function for all input combinations. Substituting them into the inequalities gives us:

$$\begin{cases} (x_1, x_2) = (0,0), & v = -1.5 \leq 0, \quad y = 0 \\ (x_1, x_2) = (0,1), & v = -0.5 \leq 0, \quad y = 0 \\ (x_1, x_2) = (1,0), & v = -0.5 \leq 0, \quad y = 0 \\ (x_1, x_2) = (1,1), & v = 0.5 > 0, \quad y = 1 \end{cases}$$

which proves that the weights and bias chosen meet the requirements of the AND problem.

2) OR:

Based on the truth table of OR, we can write the following 4 inequalities:

$$\begin{cases} 0 \times w_1 + 0 \times w_2 + b \le 0 \\ 0 \times w_1 + 1 \times w_2 + b > 0 \\ 1 \times w_1 + 0 \times w_2 + b > 0 \\ 1 \times w_1 + 1 \times w_2 + b > 0 \end{cases}$$

My selection of weights and bias are $w_1 = 1.5 \quad w_2 = 1.5 \quad b = -1$. I choose these values because they ensure that the perceptron outputs the correct OR function for all input combinations. Substituting them into the inequalities gives us:

$$\begin{cases} (x_1, x_2) = (0,0), & v = -1 \le 0, \quad y = 0 \\ (x_1, x_2) = (0,1), & v = 0.5 > 0, \quad y = 1 \\ (x_1, x_2) = (1,0), & v = 0.5 > 0, \quad y = 1 \\ (x_1, x_2) = (1,1), & v = 2 > 0, \quad y = 1 \end{cases}$$

which proves that the weights and bias chosen meet the requirements of the OR problem.

3) COMPLEMENT:

Based on the truth table of COMPLEMENT, we can write the following 2 inequalities:

$$\begin{cases} 0 \times w + b > 0 \\ 1 \times w + b \le 0 \end{cases}$$

My selection of weights and bias are $w = -1.5 \quad b = 1$. I choose these values because they ensure that the perceptron outputs the correct COMPLEMENT function for all input combinations. Substituting them into the inequalities gives us:

$$\begin{cases} x = 0, & v = 1 > 0, \quad y = 1 \\ x = 1, & v = -0.5 \le 0, \quad y = 0 \end{cases}$$

which proves that the weights and bias chosen meet the requirements of the COMPLEMENT problem.

4) NAND:

Based on the truth table of NAND, we can write the following 4 inequalities:

$$\begin{cases} 0 \times w_1 + 0 \times w_2 + b > 0 \\ 0 \times w_1 + 1 \times w_2 + b > 0 \\ 1 \times w_1 + 0 \times w_2 + b > 0 \\ 1 \times w_1 + 1 \times w_2 + b \le 0 \end{cases}$$

My selection of weights and bias are $w_1 = -1 \quad w_2 = -1 \quad b = 1.5$. I choose these values because they ensure that the perceptron outputs the correct NAND function for all input combinations. Substituting them into the inequalities gives us:

$$\begin{cases} (x_1, x_2) = (0,0), & v = 1.5 > 0, \quad y = 0 \\ (x_1, x_2) = (0,1), & v = 0.5 > 0, \quad y = 0 \\ (x_1, x_2) = (1,0), & v = 0.5 > 0, \quad y = 0 \\ (x_1, x_2) = (1,1), & v = -0.5 \le 0, \quad y = 1 \end{cases}$$

which proves that the weights and bias chosen meet the requirements of the NAND problem.

b) I mainly utilize Python libraries such as Py*Torch*, *Numpy* and *Matplotlib* to demonstrate the implementation of the logic functions listed. A multi-layer perceptron is designed with 2 input features and 1 output feature for AND, OR and NAND functions, single input feature and single output feature for COMPLEMENT function. The Sigmoid activation function is chosen due to its effectiveness in binary classification problem. The model's weights are randomly initialized following a normal distribution and biases are set to zero.

For training, the mean squared error (MSE) function is used as the loss function, and stochastic gradient descend (SGD) is used as its optimizer, given that they are most commonly used and effective. During training, the model's weights and biases are recorded. Trajectories of the weights and biases and decision boundary are plotted once the training process is done.

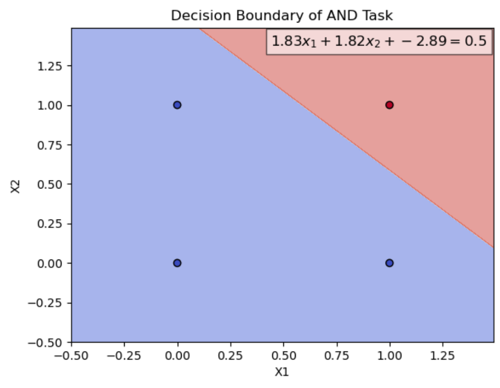
Since the implementation of AND, OR and NAND functions follows a similar structure, only the complete code for the AND task is presented here for brevity:

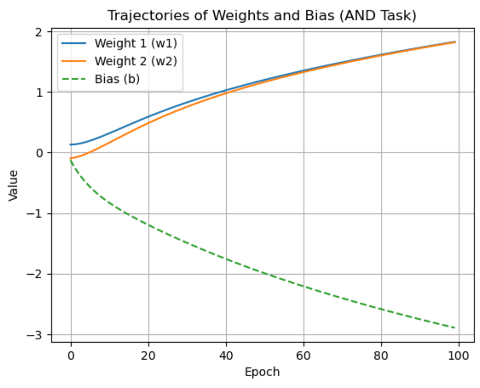
```python
# Import necessary packages
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from sympy import false

# Define a Multi-Layer Perceptron class
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.out = nn.Linear(2,1)
        ''' Define a full connected linear layer
        with 2 input features (x1,x2) and 1 output feature (y)'''
        self.sigmoid = nn.Sigmoid() # Sigmoid as activation function

        nn.init.normal_(self.out.weight, mean = 0.0, std = 0.1) # Randomly initialize
weights
        nn.init.zeros_(self.out.bias) # Set initial biases to be 0

    def forward(self,x):
        x = self.sigmoid(self.out(x))
        ''' Pass input through the Linear layer
        then apply the activation function'''
        return x

# Create an instance of MLP class
mlp=MLP()

# Define input features and expected output
# Float32 to meet the requirements of torch.nn.Linear and gradient calculation
X=torch.tensor([(0,0),(0,1),(1,0),(1,1)], dtype = torch.float32)
Y=torch.tensor([[0],[0],[0],[1]], dtype = torch.float32)

# Record weights and biases change
weight_trajectories = []
bias_trajectories = []

# Define hyperparameters
learning_rate = 1.0
epochs = 100
loss_function = nn.MSELoss() # Mean-Square Error function as loss function
optimizer = torch.optim.SGD(mlp.parameters(), lr = learning_rate) # Stochastic gradient
descend as optimizer
```

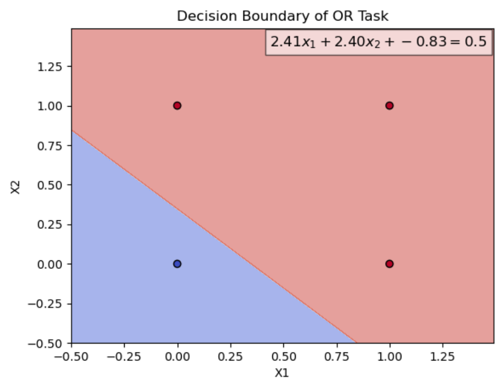
```python
# Model training
for epoch in range(epochs):
    optimizer.zero_grad() # Clear gradient
    y_predict = mlp(X) # Forward propagation
    loss = loss_function(y_predict,Y) # Loss calculation
    loss.backward() # Back propagation
    optimizer.step() # Weight update
    print(f"Epoch {epoch + 1}, Loss: {loss.item()}") # Print epoch and loss respectively

    # Add the updated weights and biases to the list
    weight_trajectories.append(mlp.out.weight.detach().numpy().flatten().copy())
    bias_trajectories.append(mlp.out.bias.detach().numpy().copy())

weight_trajectories = np.array(weight_trajectories)
bias_trajectories = np.array(bias_trajectories)

# Plot the trajectories of weights and biases
plt.figure()
plt.plot(weight_trajectories[:, 0], label = "Weight 1 (w1)")
plt.plot(weight_trajectories[:, 1], label = "Weight 2 (w2)")
plt.plot(bias_trajectories, label = "Bias (b)", linestyle = "dashed")
plt.title("Trajectories of Weights and Bias (AND Task)")
plt.xlabel("Epoch")
plt.ylabel("Value")
plt.legend()
plt.grid()
plt.show(block = false)

# Plot the decision boundary
x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
res = 0.01
xx, yy = np.meshgrid(np.arange(x_min, x_max, res), np.arange(y_min, y_max, res))
grid_points = np.c_[xx.ravel(), yy.ravel()].astype(np.float32)
''' After flattening the two-dimensional matrix into a one-dimensional array,
it is then concatenated into an array of two-dimensional coordinates'''
grid_tensor = torch.tensor(grid_points) # Numpy array to Pytorch tensor

with torch.no_grad(): # Disable gradient calculation
    zz = mlp(grid_tensor).numpy() # Put grid tensor into trained model for prediction
    zz = (zz > 0.5).astype(np.float32).reshape(xx.shape)
    ''' Binarization: zz greater than 0.5 predicted as Class 1 (y = 1),
    other wise Class 0 (y = 0)'''
```

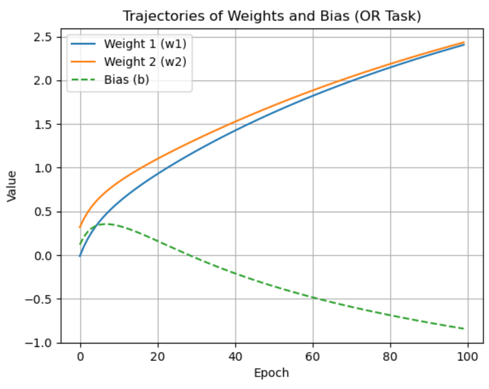
```python
# Expression of decision equation
w1, w2 = mlp.out.weight[0].detach().numpy()
b = mlp.out.bias[0].detach().numpy()
decision_eq = f"${w1:.2f}x_1 + {w2:.2f}x_2 + {b:.2f} = 0.5$"

plt.figure()
plt.text(0.985, 0.98, decision_eq, fontsize=12, color="black",
         transform=plt.gca().transAxes,
         verticalalignment='top', horizontalalignment='right',
         bbox=dict(facecolor="white", alpha=0.6))
plt.contourf(xx, yy, zz, alpha = 0.5, cmap = "coolwarm")
plt.scatter(X[:, 0], X[:, 1], c = Y.numpy(), edgecolors = "k", marker = "o", cmap =
"coolwarm")
plt.title("Decision Boundary of AND Task")
plt.xlabel("X1")
plt.ylabel("X2")
plt.show()
```

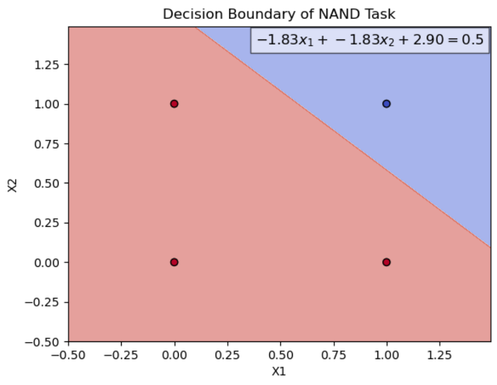
For AND, OR and NAND functions, the decision boundaries and the trajectories of weights and biases are plotted respectively as shown below:

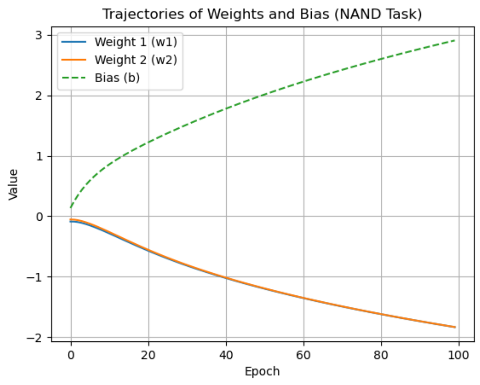












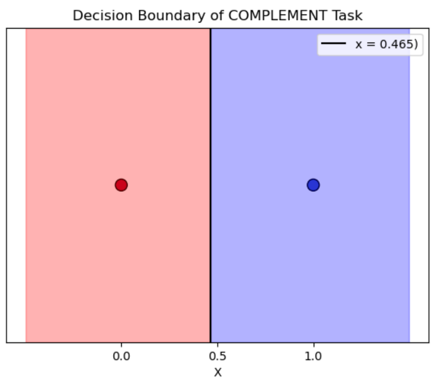
The complete code for the COMLEMENT task is presented as shown below:

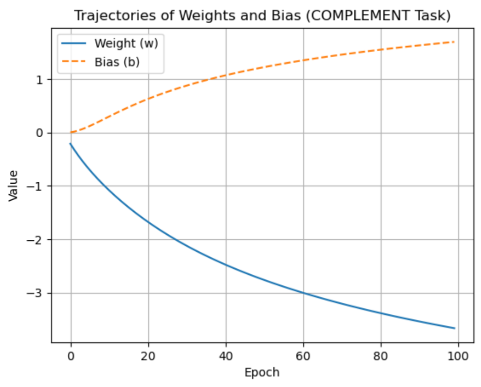
```
# Import necessary packages
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from sympy import false

# Define a Multi-Layer Perceptron class for COMPLEMENT task
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.out = nn.Linear(1,1)
        ''' Define a full connected linear layer
        with 1 input feature (x) and 1 output feature (y)'''
        self.sigmoid = nn.Sigmoid() # Sigmoid as activation function

        nn.init.normal_(self.out.weight, mean = 0.0, std = 0.1) # Randomly initialize
weights
        nn.init.zeros_(self.out.bias) # Set initial biases to be 0

    def forward(self,x):
        x = self.sigmoid(self.out(x))
        ''' Pass input through the Linear layer
        then apply the activation function'''
        return x

# Create an instance of MLP class
mlp=MLP()

# Define input features and expected output
# Float32 to meet the requirements of torch.nn.Linear and gradient calculation
```

```
X=torch.tensor([[0],[1]], dtype = torch.float32)
Y=torch.tensor([[1],[0]], dtype = torch.float32)

# Record weights and biases change
weight_trajectories = []
bias_trajectories = []

# Define hyperparameters
learning_rate = 1.0
epochs = 100
loss_function = nn.MSELoss() # Mean-Square Error function as loss function
optimizer = torch.optim.SGD(mlp.parameters(), lr = learning_rate) # Stochastic gradient
descend as optimizer

# Model training
for epoch in range(epochs):
    optimizer.zero_grad() # Clear gradient
    y_predict = mlp(X) # Forward propagation
    loss = loss_function(y_predict,Y) # Loss calculation
    loss.backward() # Back propagation
    optimizer.step() # Weight update
    print(f"Epoch {epoch + 1}, Loss: {loss.item()}") # Print epoch and loss respectively

    # Add the updated weights and biases to the list
    weight_trajectories.append(mlp.out.weight.detach().numpy().flatten().copy())
    bias_trajectories.append(mlp.out.bias.detach().numpy().copy())

weight_trajectories = np.array(weight_trajectories)
bias_trajectories = np.array(bias_trajectories)

# Plot the trajectories of weights and biases
plt.figure()
plt.plot(weight_trajectories, label = "Weight (w)")
plt.plot(bias_trajectories, label = "Bias (b)", linestyle = "dashed")
plt.title("Trajectories of Weights and Bias (COMPLEMENT Task)")
plt.xlabel("Epoch")
plt.ylabel("Value")
plt.legend()
plt.grid()
plt.show(block = false)

# Plot the decision boundary
x_range = np.linspace(-0.5, 1.5, 200).reshape(-1, 1) # Input range from -0.5 to 1.5,
reshape to (100,1) dimension
```

```
tensor_range = torch.tensor(x_range, dtype=torch.float32) # Numpy array to Torch tensor

with torch.no_grad():
    y_predict = mlp(tensor_range).numpy()  # 计算 x_range 上的模型预测值

plt.figure()
plt.scatter(X.numpy(), np.zeros_like(X.numpy()), c=Y.numpy(), edgecolors="k",
marker="o", cmap="coolwarm", s=100)
decision_boundary = tensor_range[np.abs(y_predict - 0.5).argmin()].item()
plt.axvspan(-0.5, decision_boundary, color="red", alpha=0.3)
plt.axvspan(decision_boundary, 1.5, color="blue", alpha=0.3)
plt.axvline(decision_boundary, color="black", label=f"x = {decision_boundary:.3f})")
plt.title("Decision Boundary of COMPLEMENT Task")
plt.xlabel("X")
plt.xticks([0, 0.5, 1])
plt.yticks([])
plt.legend()
plt.show()
```

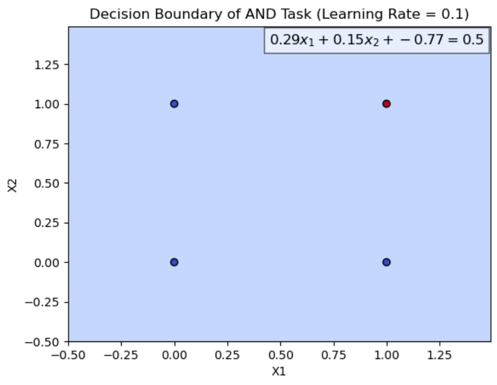
For COMPLEMENT function, the decision boundary and the trajectories of weights and bias are plotted as shown below:

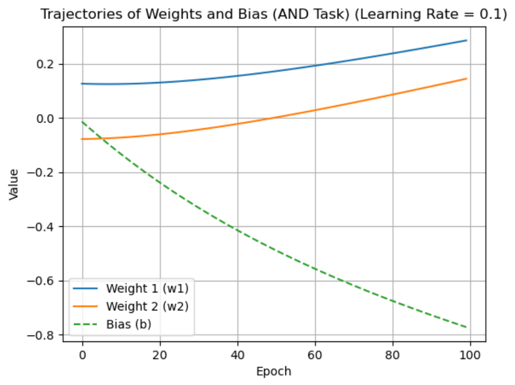


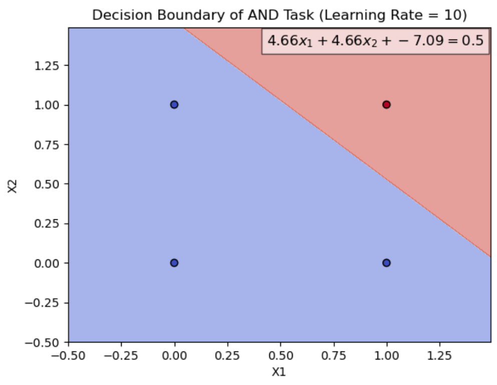


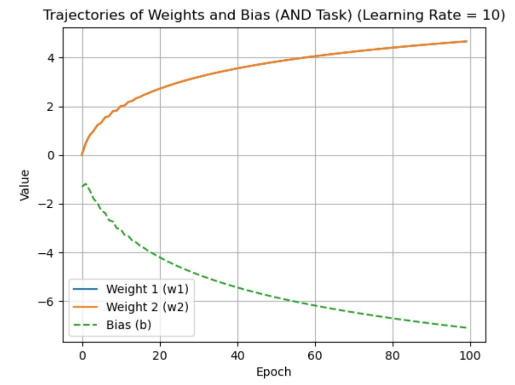
Compared with the results obtained in a)., the decision boundaries calculated by the two methods have the same direction, and their slopes and intercepts are also close. They can both correctly complete their respective classification tasks, proving that the model training is successful.

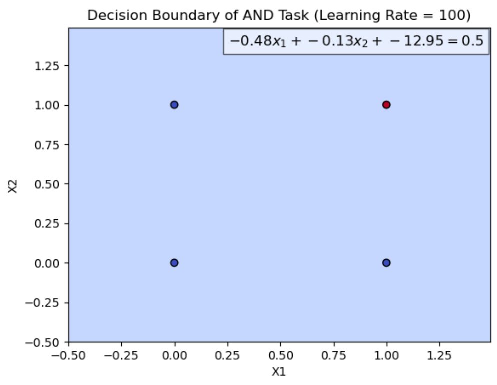
With the number of training epochs fixed at 100, different learning rates have a significant impact on the training results. Taking the AND function as an example, the decision boundaries and the corresponding weights/biases obtained with learning rates of 0.1, 10, and 100 are shown in the figures below:

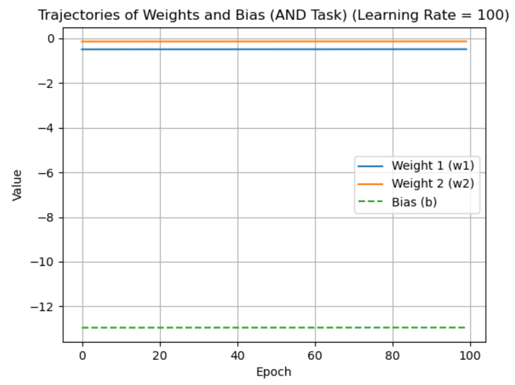






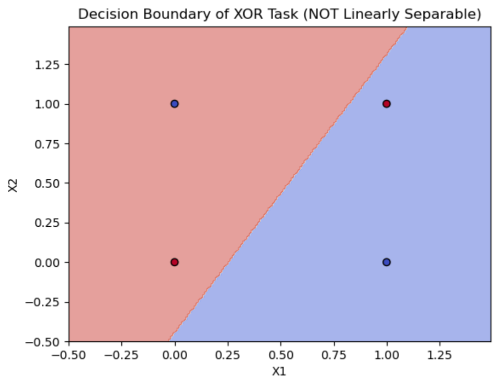


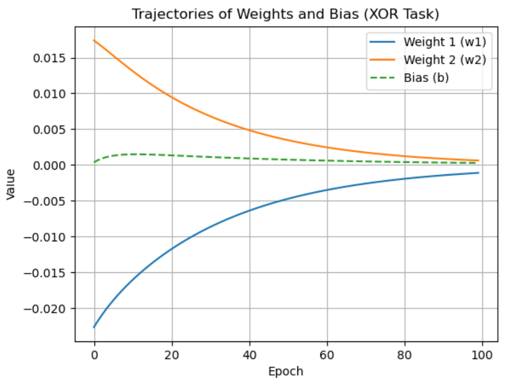




From the figures above, it can be observed that when the learning rate is set to 0.1, the model fails to correctly complete the classification task. This is because the learning rate is too low, resulting in very small update steps, making weight adjustments extremely slow and requiring a significantly larger number of training iterations to converge. When the learning rate is set to 10, the model successfully classifies the data, and I notice that the corresponding model loss is much lower than that of the model trained with a learning rate of 1.0. This indicates that the chosen learning rate is appropriate, allowing the model to converge faster and train more stably. However, when the learning rate is set to 100, the model fails to classify correctly due to excessively large update steps, which may cause it to overshoot the optimal solution or lead to gradient explosion, preventing convergence. Therefore, selecting an appropriate learning rate is crucial for effective model training.

c) Since the implementation of AND, OR, NAND and XOR functions follows a similar structure, only the decision boundary and the trajectories of weights and bias of XOR are presented here for brevity:

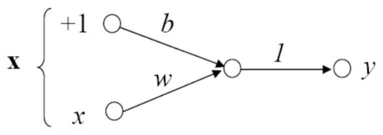




From the figures above, it can be observed that when using a single-layer perceptron, the model can never perfectly classify the XOR task. This indicates that XOR is not linearly separable, meaning that no single linear decision boundary can correctly separate the two classes. A single-layer perceptron can only learn linear decision boundaries; therefore, regardless of how it is trained, it cannot reduce the classification error of XOR to 0%. To solve the XOR problem, a multi-layer perceptron with hidden layers and nonlinear activation functions must be used.

Q4.

Single layer perceptron with pure linear activation function can be used to fit a linear model to a set of input-output pairs. Suppose that we are given the following pairs: $\{(0,0.5), (0.8,1), (1.6,4), (3,5), (4.0,6), (5.0,8)\}$ and a single linear neuron as shown in the following figure.



a). Find the solution of w and b using the standard linear least-squares (LLS) method. Plot out the fitting result.
b). Suppose that initial weight is chosen randomly and learning rate is η 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?
c). Compare the results obtained by LLS and the LMS methods.
d). Repeat the simulation study in b) with different learning rates, and explain your findings.

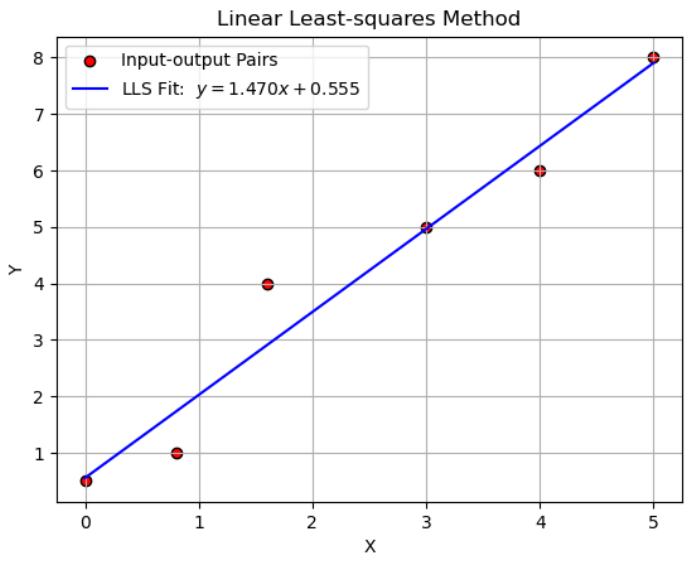
A4.

a) The Python code for solving w and b using the linear least-squares method is shown below:

```
# Import necessary packages
import numpy as np
import matplotlib.pyplot as plt

# Define given input-output pairs
X = np.array([0, 0.8, 1.6, 3, 4.0, 5.0])
Y = np.array([0.5, 1, 4, 5, 6, 8])
```

```python
# Define necessary intermediate variables
sum_x = 0
sum_y = 0
sum_xy = 0
sum_x_square = 0
n = len(X)

for i in range(n):
    sum_x +=  X[i]
    sum_y += Y[i]
    sum_x_square += X[i] ** 2
    sum_xy += X[i] * Y[i]

avg_x = sum_x / n
avg_y = sum_y / n
w = (sum_xy - n * avg_x * avg_y) / (sum_x_square - n * avg_x ** 2)
b = avg_y - w * avg_x
X_fit = np.linspace(min(X), max(X), 100)
Y_fit = w * X_fit + b

# Plot scatter plots and the fit line
plt.figure()
plt.scatter(X, Y, color="red", label="Input-output Pairs", edgecolors="black")
plt.plot(X_fit, Y_fit, color="blue", label=f"LLS Fit:  $y={w:.3f}x + {b:.3f}$")
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Linear Least-squares Method")
plt.legend()
plt.grid(True)
plt.show()
```

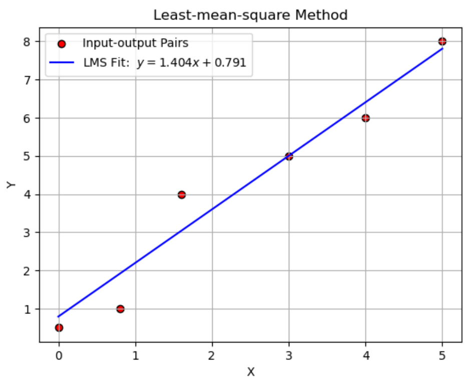
The fitting result is plotted as shown below, from which we can see that $w = 1.470$ and $b = 0.555$.

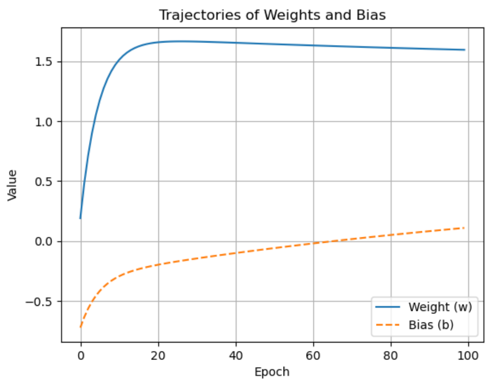


b) The Python code for solving w and b using the least-mean-square method is shown below:

```
# Import necessary packages
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from sympy import false

# Define given input-output pairs
# Float32 to meet the requirements of torch.nn.Linear and gradient calculation
X = torch.tensor([[0], [0.8], [1.6], [3], [4.0], [5.0]], dtype = torch.float32)
Y = torch.tensor([[0.5], [1], [4], [5], [6], [8]], dtype = torch.float32)

# Randomly initialize weight and bias and record their change
w = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)
weight_trajectories = []
bias_trajectories = []

# Define hyperparameters
learning_rate = 0.01
epochs = 100
loss_function = nn.MSELoss() # Mean-Square Error function as loss function

# Model training
for epoch in range(epochs):
    y_predict = w * X + b
    loss = loss_function(y_predict, Y) # Loss calculation
    loss.backward() # Back propagation
    with torch.no_grad():
        w -= learning_rate * w.grad # Gradient calculation
        b -= learning_rate * b.grad

    w.grad.zero_() # Clear gradient
    b.grad.zero_()

    print(f"Epoch {epoch+1}, Loss: {loss.item()}")

    # Add the updated weights and biases to the list for every epoch
    weight_trajectories.append(w.detach().numpy().flatten().copy())
    bias_trajectories.append(b.detach().numpy().copy())

weight_trajectories = np.array(weight_trajectories)
bias_trajectories = np.array(bias_trajectories)
w_value = w.detach().numpy()
b_value = b.detach().numpy()

X_fit = np.linspace(min(X), max(X), 100)
Y_fit = w_value * X_fit + b_value

# Plot the trajectories of weights and biases
plt.figure()
plt.plot(weight_trajectories, label = "Weight (w)")
plt.plot(bias_trajectories, label = "Bias (b)", linestyle = "dashed")
plt.title("Trajectories of Weights and Bias")
plt.xlabel("Epoch")
plt.ylabel("Value")
plt.legend()
plt.grid()
plt.show(block = false)

# Plot scatter plots and the fit line
plt.figure()
plt.scatter(X, Y, color="red", label="Input-output Pairs", edgecolors="black")
plt.plot(X_fit, Y_fit, color="blue", label=f"LMS Fit:  $y={w_value[0]:.3f}x +
{b_value[0]:.3f}$")
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Least-mean-square Method")
plt.legend()
plt.grid(True)
plt.show()
```

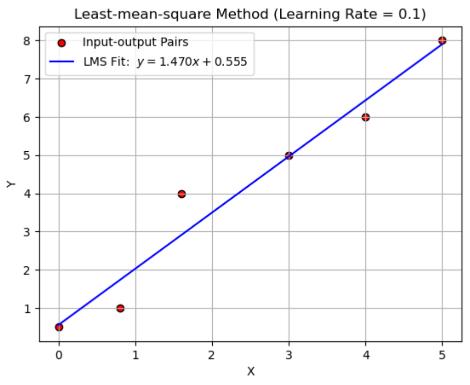
The fitting result and the trajectories of weights and bias versus learning steps using least-mean-square method are plotted as shown below. As can be seen from the figure, the weights will converge.

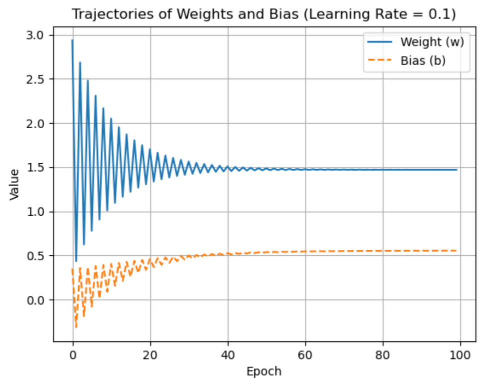


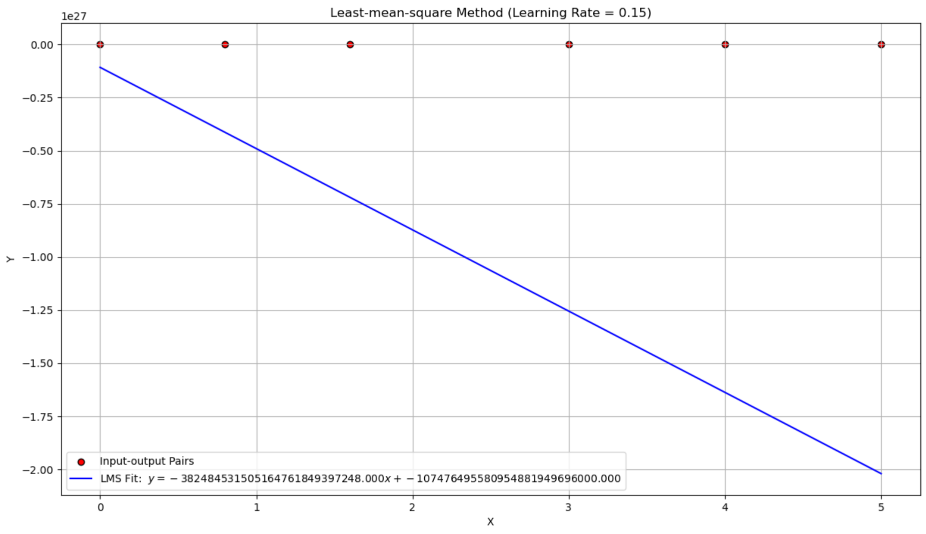


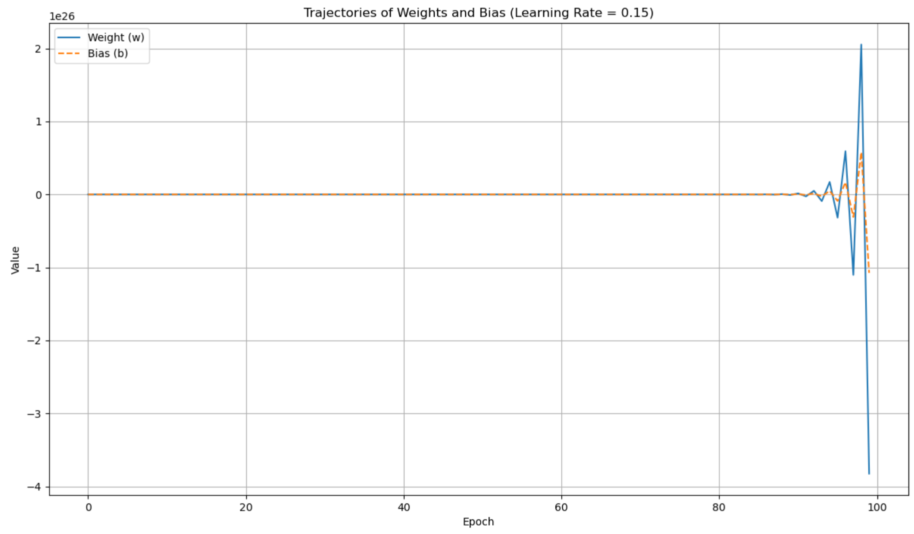
c) The LLS method directly computes the optimal solution for w and b using a fixed formula, meaning it can find out the optimal solution immediately. But LMS method on the other hand, requires multiple iterations to adjust the weights using gradient descent. The final solution depends on the number of epochs and the learning rate.

d) With the number of training epochs fixed at 100, different learning rates have a significant impact on the training results. The fitted straight line and the corresponding weights/biases obtained with learning rates of 0.1 and 0.15 are shown in the figures below:









As can be seen from the figures, when the learning rate is 0.1, although the initial updates of weights and biases oscillate, the training eventually stabilizes and reaches the optimal solution that is exactly the same as the fitted line calculated by the LLS method. LMS can approximate the LLS solution given enough training iterations and a property learning rate. But when the learning rate increases to 0.15, the slope and intercept of the fitted line become extreme. w and b trajectories show oscillations and diverge violently in the later stages. It is because each update step is too large, causing the parameters to be excessively corrected and potentially overshooting the optimal point. If sufficiently large, the next update may overshoot the optimal point, leading to oscillations. If the oscillations persist and the values gradually increase, it will eventually result in parameter explosion.

Q5.

Consider that we are trying to fit a linear model to a set of input-output pairs $(x(1), d(1))$, $(x(2), d(2))$ … $(x(n), d(n))$ observed in an interval of duration n, where input x is m-dimensional vector, $x = [x_1 \ x_2 \ \dots . x_m]^T$. The linear model takes the following form:

$$y(x) = w_1 x_1 + w_2 x_2 + \cdots w_m x_m = w^T x$$

Derive the formula to calculate the optimal parameter w^* such that the following cost function $J(w)$ is minimized.

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

Where $r(i) > 0$ are the weighting factors for each output error $e(i)$, and $\lambda > 0$ is the regularization factor.

A5.

Since $x = [x_1 \ x_2 \ \dots . x_m]^T$ and $d(i)$ is the output of corresponding $x(i)$, we can get the matrix representation of X and d:

$$X = \begin{bmatrix} x(1)^T \\ x(2)^T \\ \dots \\ x(3)^T \end{bmatrix} \quad (n \times m \ matrix)$$

$$d = \begin{bmatrix} d(1) \\ d(2) \\ \dots \\ d(n) \end{bmatrix} \quad (n \times 1 \ matrix)$$

Since $r(i)$ are the weighting factors for each output error $e(i)$, we can get the matrix representation of R:

$$R = \begin{bmatrix} r(1) & 0 & 0 & 0 \\ 0 & r(2) & 0 & 0 \\ \dots & \dots & \ddots & \dots \\ 0 & 0 & 0 & r(n) \end{bmatrix} \quad (n \times n \ diagonal \ matrix)$$

Using the above representation of matrices, we can rewrite $J(w)$ as:

$$J(w) = \frac{1}{2}\sum_{i=1}^{n} r^2(i) \left(d(i) - y(x(i))\right)^2 + \frac{1}{2}\lambda w^T w = \frac{1}{2}(d - Xw)^T R^2 (d - Xw) + \frac{1}{2}\lambda w^T w$$

Because minimizing the loss function requires the derivative of $J(w)$ with respect to w to be equal to 0, then

$$J'(w^*) = -X^T R^2 (d - Xw^*) + \lambda w^* = 0$$

$$(X^T R^2 X + \lambda I) w^* = x^T R^2 d$$

If $(X^T R^2 X + \lambda I)$ is a full rank matrix, then the formula to calculate the optimal parameter w^* is:

$$w^* = (X^T R^2 X + \lambda I)^{-1} X^T R^2 d$$