



**University School of Automation & Robotics  
GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY  
East Delhi Campus, Surajmal vihar  
Delhi - 110092**

## **Artificial Neural Network Lab**

**Lab File**

**Artificial Intelligence & Machine Learning  
(2022 - 2026)**

**Submitted by:**

Rohan saini  
04419051622  
AIML B1  
USAR, GGSIPU

**Submitted to:**

Ms. Kirti  
Assistant Professor

# INDEX

Lab No.	PROGRAM	Teacher's Signature
1	Write a program to understand basic plots in Python	
2	Write a program to perform the basic matrix operations	
3	Write a program to implement AND, OR, XOR gates to understand Linearly separable and non-linearly separable problems	
4	Write a program for implementation of different Activation functions to train Neural Network	
5	Write a program to study Weight and Bias effect on Output	
6	Write a program for implementation of different learning rules	
7	Write a program for implementation of Perceptron Networks	
8	Write a program to build an Artificial Neural Network by implementing the a) Backpropagation Algorithm b) Momentum Backpropagation Algorithm and test the same using appropriate data sets.	
9	Formulate a problem statement and implement ANN on an appropriate dataset and visualize the results	
10	a) Write a program to implement gradient ascent and descent algorithm b) Write a program to implement Newton Method	

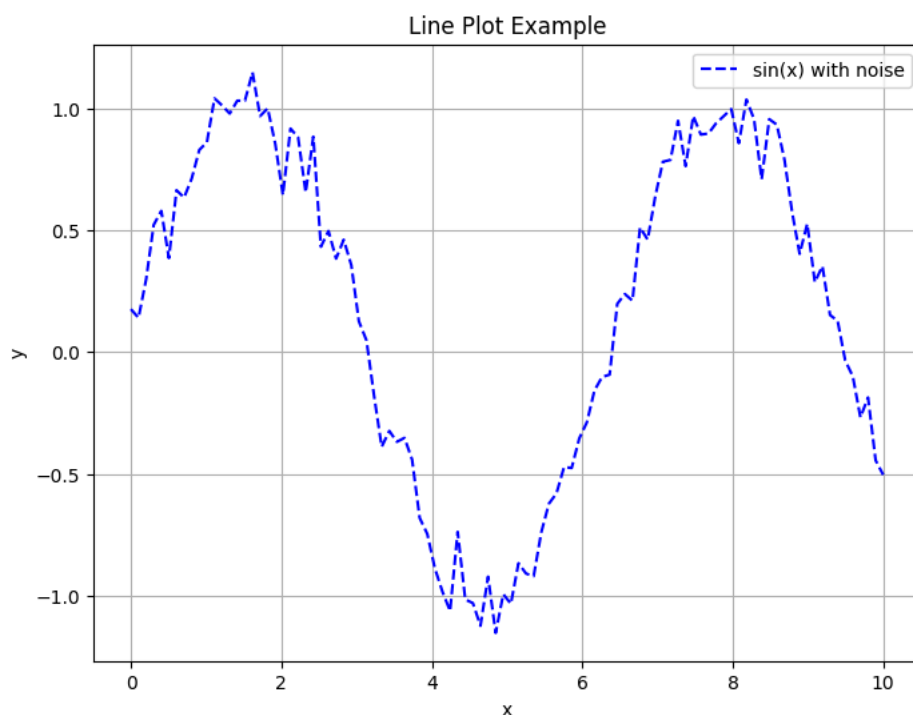
## ✓ 1. Write a program to understand basic plots in Python

1

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
np.random.seed(0)

x = np.linspace(0,10,100)
y=np.sin(x) + np.random.normal(0,0.1,100)

plt.figure(figsize=(8,6))
plt.plot(x,y,label='sin(x) with noise', color='blue', linestyle='--')
plt.title('Line Plot Example')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

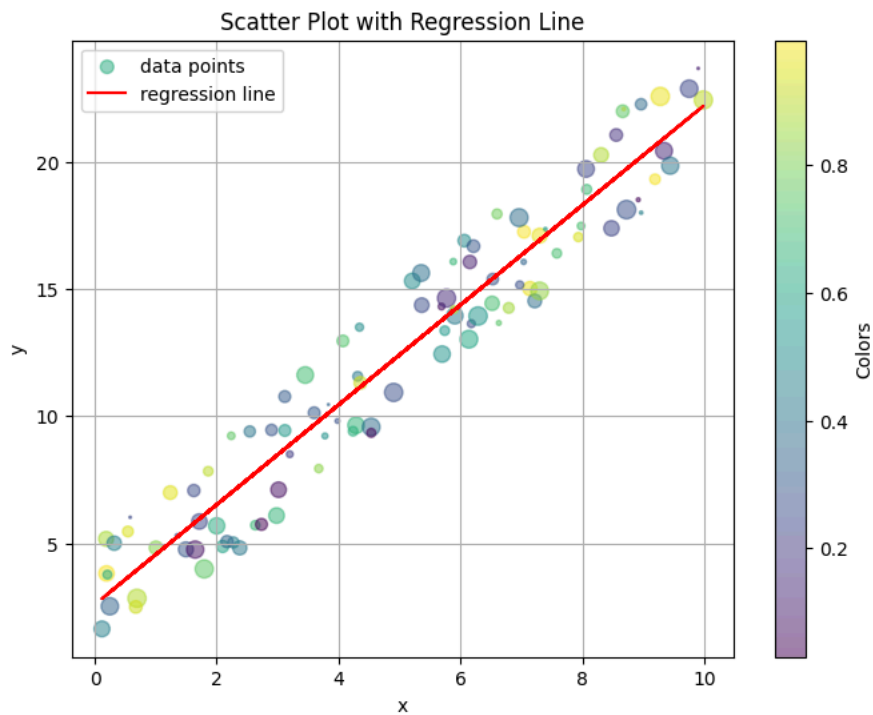


```
x = np.random.rand(100)*10
y = np.random.rand(100)*5 + 2*x
sizes = np.random.rand(100)*100
colors = np.random.rand(100)

coefficients = np.polyfit(x, y, 1)
poly_function = np.poly1d(coefficients)

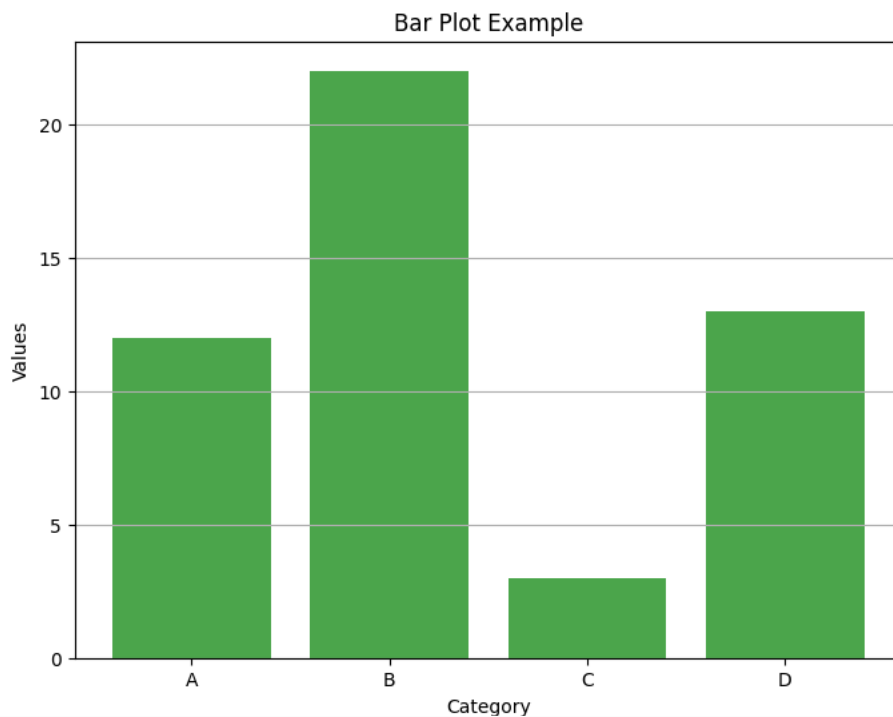
plt.figure(figsize=(8, 6))
plt.scatter(x, y, s=sizes, c=colors, alpha=0.5, label='data points')

plt.plot(x, poly_function(x), color='red', label='regression line')
plt.title('Scatter Plot with Regression Line')
plt.xlabel('x')
plt.ylabel('y')
plt.colorbar(label='Colors')
plt.legend()
plt.grid(True)
plt.show()
```



```
categories = ['A', 'B', 'C', 'D']
values = np.random.randint(1, 50, size=len(categories))
```

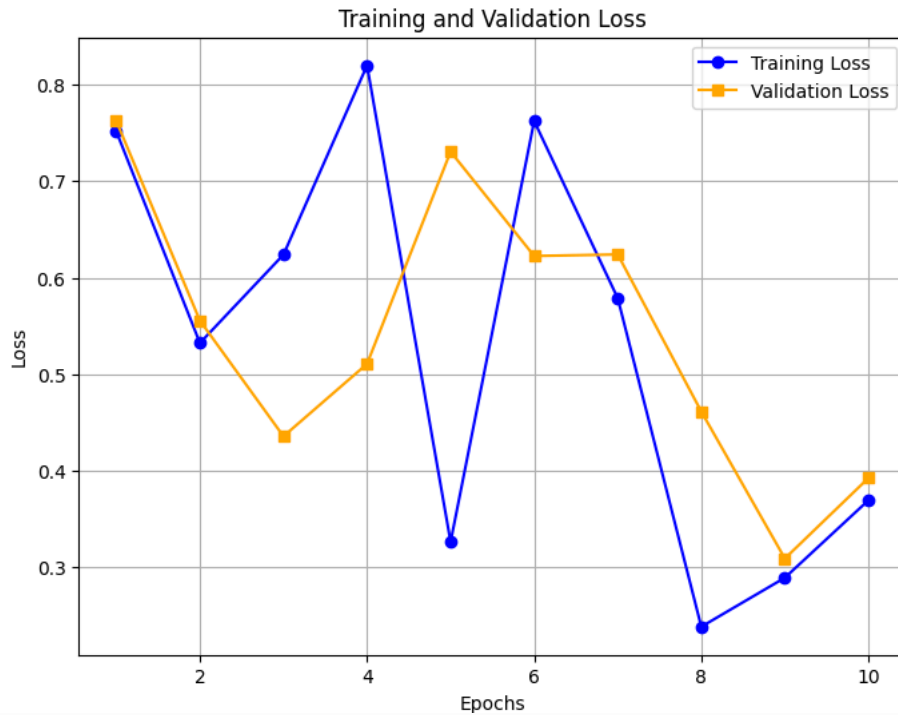
```
plt.figure(figsize=(8, 6))
plt.bar(categories, values, color='green', alpha=0.7)
plt.title('Bar Plot Example')
plt.xlabel('Category')
plt.ylabel('Values')
plt.grid(axis='y')
plt.show()
```



```
epochs = np.arange(1, 11)
train_loss = np.random.rand(10) * 0.5 + np.linspace(0.5, 0.1, 10)
val_loss = np.random.rand(10) * 0.5 + np.linspace(0.4, 0.05, 10)
```

```
# Plot the training and validation loss
plt.figure(figsize=(8, 6))
plt.plot(epochs, train_loss, label='Training Loss', color='blue', marker='o')
plt.plot(epochs, val_loss, label='Validation Loss', color='orange', marker='s')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
```

```
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

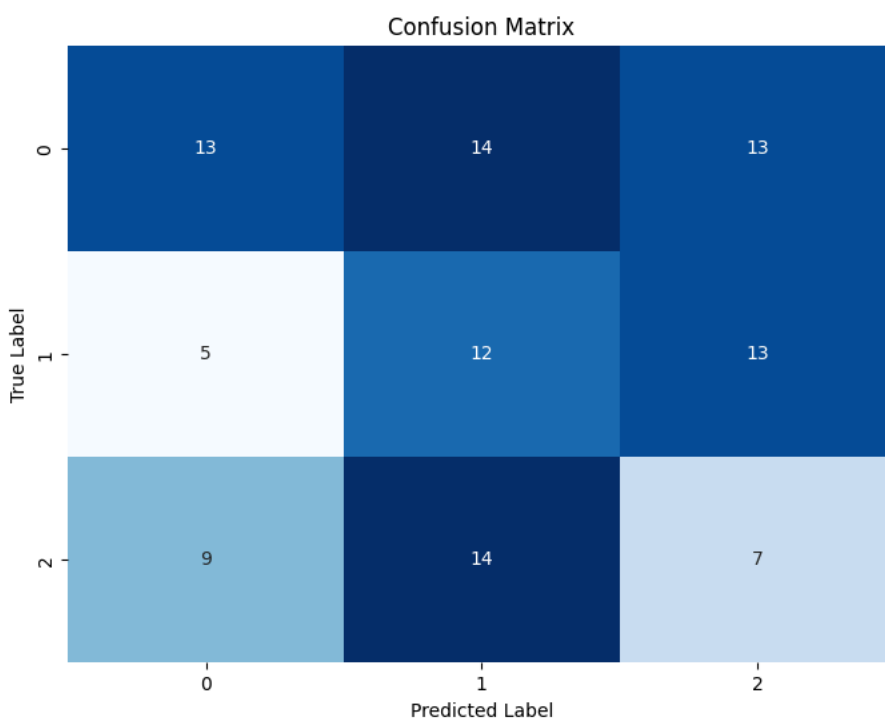


```
from sklearn.metrics import confusion_matrix
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

true_labels = np.random.randint(0, 3, size=100)
predicted_labels = np.random.randint(0, 3, size=100)

cm = confusion_matrix(true_labels, predicted_labels)
```

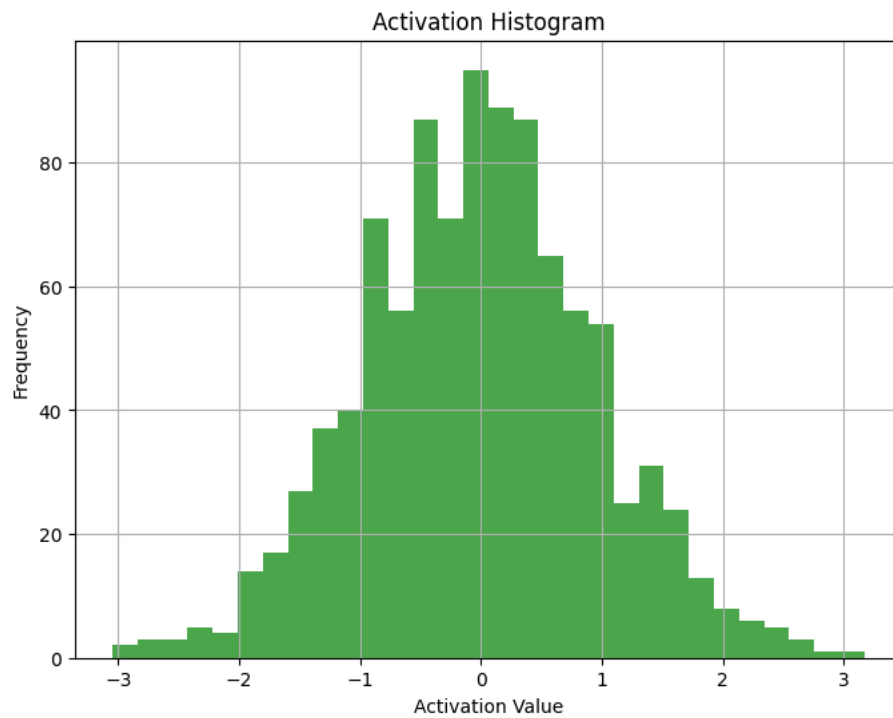
```
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```



```
import numpy as np
import matplotlib.pyplot as plt
```

```
activations = np.random.randn(1000)
```

```
plt.figure(figsize=(8, 6))
plt.hist(activations, bins=30, color='green', alpha=0.7)
plt.title('Activation Histogram')
plt.xlabel('Activation Value')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```



## ✓ 2. Write a program to perform the basic matrix operations

```
import numpy as np
A = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])

B= np.array([[10,11,12],[13,14,15],[16,17,18]])

print("Matrix A:\n",A,"\n")
print("Matrix B:\n",B)

↵ Matrix A:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Matrix B:
[[10 11 12]
 [13 14 15]
 [16 17 18]]

print("addition:\n",A+B)
print("\nsubtraction\n",A-B)
print("\nElement wise multiplication\n",A*B)
print("\ntranspose\n",A.T)
print("\nMatrix Multiplication\n",np.dot(A,B.T))
```

```
↵ addition:
[[115  47  85 176]
 [ 61 141 137 114]
 [ 99  58 114 159]]

subtraction
[[-51  43  -1   0]
 [-11  -5  13  12]
 [ 13   6 -72  21]]

Element wise multiplication
[[2656   90 1806 7744]
 [ 900 4964 4650 3213]
 [2408  832 1953 6210]]

transpose
[[32 25 56]
 [45 68 32]
 [42 75 21]
 [88 63 90]]

Matrix Multiplication
[[12296 11529 12524]
 [10980 13727 14165]
 [13535 10244 11403]]
```

## ✓ Batch Matrix Operations

Batch matrix multiplication (`np.dot(input_batch, weights)`) is used in the forward pass of neural networks

```
batch_size=16
input_features=10
output_features = 5
input_batch = np.random.randint(0,101,size=(batch_size,input_features))
weights = np.random.randint(0,101,size=(input_features,output_features))
output_batch = np.dot(input_batch,weights)

print("Input batch\n",input_batch)
print("\nOutput Batch\n",output_batch)
print("\nweights\n",weights)
```

```
↵ Input batch
[[ 94  41  14  98  70  73  88  20  22  19]
 [ 96  20   6  93  47  91  90  20  50  39]
 [ 68   2  62  18  37  55  58  91  19  93]
 [  7  14  64 100  76  82  92  54  88  90]
 [ 77   7  76  28  62  31  76  27   9  27]
 [ 47  32  10  81  26   6  76  68  11  19]
 [ 51  18  82  41  11  78  74  85  38  37]
 [ 16  97  24  83  90  80  56   8  40  88]]
```

```
[ 72 44 93 82 52 37 90 84 7 38]
[ 15 5 64 90 93 17 26 41 41 71]
[ 73 56 61 47 1 65 84 96 76 31]
[ 70 89 25 57 93 85 59 95 64 42]
[ 50 92 90 52 97 44 55 85 13 64]
[ 65 80 89 58 88 84 94 98 5 94]
[ 65 13 76 54 69 25 9 81 68 36]
[ 75 60 79 76 29 65 88 21 96 5]]
```

Output Batch

```
[[22049 29001 25462 29012 26167]
[26416 30757 23943 28346 26818]
[28095 22496 25167 26548 23430]
[31472 32269 28903 31545 27744]
[20169 15497 23550 23223 21476]
[11366 17926 18150 22440 20117]
[26502 22876 22714 26738 26725]
[24975 35699 27898 28352 20102]
[25083 26279 30291 33724 30592]
[20787 21664 22449 23113 17571]
[27509 28690 26602 32494 30712]
[27341 35502 32935 36574 29017]
[26315 30600 34666 35316 26991]
[34003 36529 38875 40210 33147]
[24077 21999 24051 26950 22352]
[27396 29541 27024 31520 29959]]
```

weights

```
[[78 63 67 79 74]
[ 6 99 67 70 27]
[81 10 58 49 58]
[23 90 11 50 51]
[ 6 13 83 50 9]
[90 83 5 17 37]
[ 7 0 66 60 77]
[16 21 41 70 62]
[58 48 30 44 37]
[99 85 56 39 7]]
```

## ✓ Matrix Concatenation and Splitting

Matrix concatenation (`np.concatenate((A, B), axis=1)`) and splitting (`np.split(concatenated_matrix, 2, axis=1)`) can be used in neural network architectures that involve concatenating feature maps or splitting inputs/outputs for parallel processing or multiple branches within the network.

```
A=np.random.randint(1,101,size=(3,4))
B=np.random.randint(1,101,size=(3,4))

concatenatedmatrix= np.concatenate((A,B),axis=1)
print("concatenated matrix\n",concatenatedmatrix)

p1,p2 = np.split(concatenatedmatrix,2,axis=1)

print("\npart1\n",p1,"\n\npart2\n",p2)

print("\nShape of matrix A:",A.shape)
print("Shape of matrix A:",A.shape)
print("Shape of concatenated matrix",concatenatedmatrix.shape)
print("Shape of matrix p1 after splitting:",p1.shape)
print("Shape of matrix p2 after splitting:",p2.shape)
```

```
concatenated matrix
[[32 45 42 88 83 2 43 88]
[25 68 75 63 36 73 62 51]
[56 32 21 90 43 26 93 69]]
```

```
part1
[[32 45 42 88]
[25 68 75 63]
[56 32 21 90]]
```

```
part2
[[83 2 43 88]
[36 73 62 51]
[43 26 93 69]]
```

```
Shape of matrix A: (3, 4)
Shape of matrix A: (3, 4)
Shape of concatenated matrix (3, 8)
Shape of matrix p1 after splitting: (3, 4)
```



Shape of matrix p2 after splitting: (3, 4)

## ✓ Eigen value and eigen matrix

```
A = np.array([[1,2],[3,4]])  
  
rankA=np.linalg.matrix_rank(A)  
eigenVal,eigenVect=np.linalg.eig(A)  
A_inv = np.linalg.inv(A)
```

```
print("Rank of A:",rankA)  
print("\nEigenvalue:",eigenVal)  
print("\nEigenVector",eigenVect)  
print("\nInverse of matrix",A_inv)
```



Rank of A: 2

Eigenvalue: [-0.37228132 5.37228132]

EigenVector [[-0.82456484 -0.41597356]  
[ 0.56576746 -0.90937671]]

Inverse of matrix [[-2. 1. ]  
[ 1.5 -0.5]]

### 3. Write a program to implement AND, OR, XOR gates to understand Linearly separable and non-linearly separable problems

```
import numpy as np
import matplotlib.pyplot as plt

# AND gate data
X_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_and = np.array([0, 0, 0, 1])

# OR gate data
X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_or = np.array([0, 1, 1, 1])

# XOR gate data
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([0, 1, 1, 0])

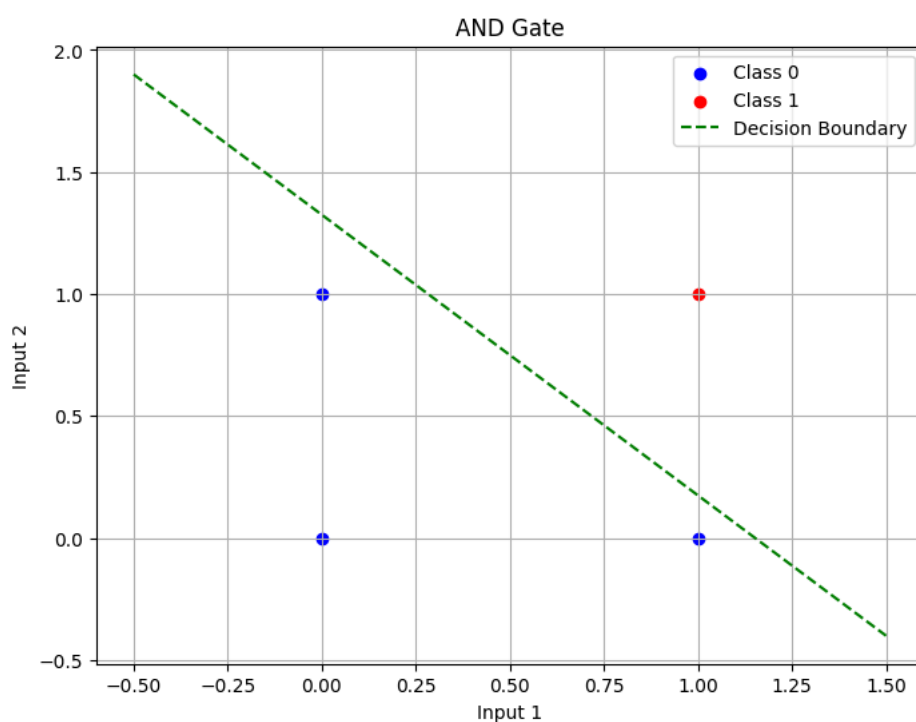
def plot_data_and_boundary(X, y, gate_type):
    plt.figure(figsize=(8, 6))

    plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='blue', label='Class 0')
    plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='red', label='Class 1')

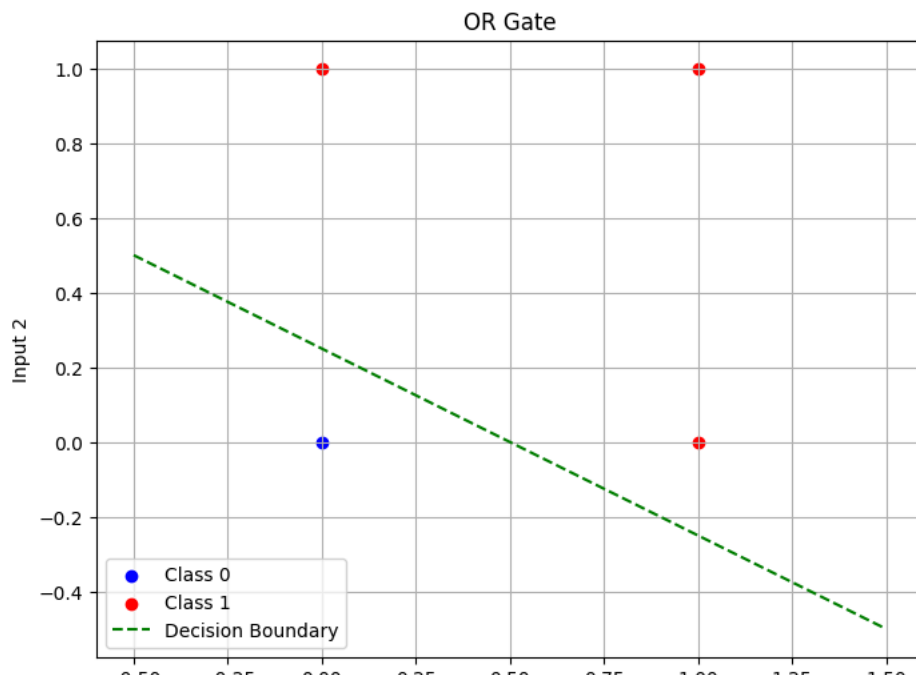
    if gate_type == 'AND':
        plt.plot([-0.5, 1.5], [1.9, -0.4], color='green', linestyle='--', label='Decision Boundary')
    elif gate_type == 'OR':
        plt.plot([-0.5, 1.5], [0.5, -0.5], color='green', linestyle='--', label='Decision Boundary')
    elif gate_type == 'XOR':
        plt.plot([-0.5, 1.5], [1.9, -0.4], color='green', linestyle='--', label='Decision Boundary 1')
        plt.plot([-0.5, 1.5], [0.5, -0.5], color='purple', linestyle='--', label='Decision Boundary 2')

    plt.title(f'{gate_type} Gate')
    plt.xlabel('Input 1')
    plt.ylabel('Input 2')
    plt.legend()
    plt.grid(True)
    plt.show()

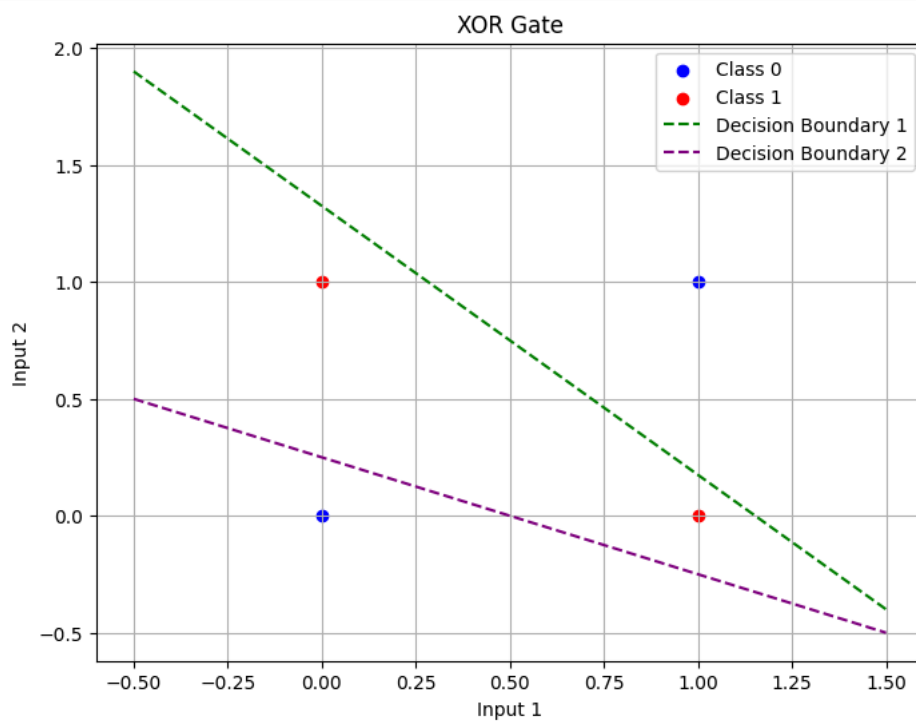
plot_data_and_boundary(X_and, y_and, gate_type='AND')
```



```
plot_data_and_boundary(X_or, y_or, gate_type='OR')
```

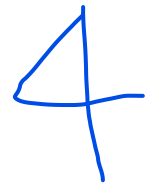


```
plot_data_and_boundary(X_xor, y_xor, gate_type='XOR')
```



## 4. Write a program for implementation of different Activation functions to train Neural Network

### ✓ Apply Sigmoid, tanh, ReLU Activation Functions



```
X1 = [1, 1, -1, -1]
print(X1)
```

```
X2 = [1, -1, 1, -1]
print(X2)
```

```
y = [1, -1, -1, -1]
print(y)
```

```
↩ [1, 1, -1, -1]
   [1, -1, 1, -1]
   [1, -1, -1, -1]
```

### ✓ Tanh X function

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

```
import numpy as np
def tanh_function(z):
    return (np.exp(z) - np.exp(-z)) / (np.exp(z) + np.exp(-z))
```

```
w1 = 10
w2 = 11
```

```
y_pred = [0, 0, 0, 0]
for i in range(4):
    y_pred[i] = (X1[i] * w1) + (X2[i] * w2)
print(y_pred)
```

```
↩ [21, -1, 1, -21]
```

```
for i in y_pred:
    print(tanh_function(i))
```

```
↩ 1.0
   -0.7615941559557649
   0.7615941559557649
   -1.0
```

### ✓ Sigmoid

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

### ✓ ReLU

```
def relu(x):
    return max(0, x)

def test(y, y_p):
    flag = "Correct"
    index = -1
    for i in range(4):
        if y[i] != y_p[i]:
            flag = "Incorrect"
            index = i
            break
    return flag, index
```

```
print(test(y, y_pred))
```

```
↗ ('Incorrect', 0)
```

## ✓ Effect of Sigmoid and Tanh on input

```
x1 = [1, -10, 0, 15, -2]
tanhOutput = [0, 0, 0, 0, 0]
SigmoidOutput = [0, 0, 0, 0, 0]
ReLUOutput = [0, 0, 0, 0, 0]
```

```
for i in range(len(x1)):
    tanhOutput[i] = tanh_function(x1[i])
    SigmoidOutput[i] = sigmoid(x1[i])
```

```
print("Tanh Output:", [float(x) for x in tanhOutput])
print("Sigmoid Output:", [float(x) for x in SigmoidOutput])
```

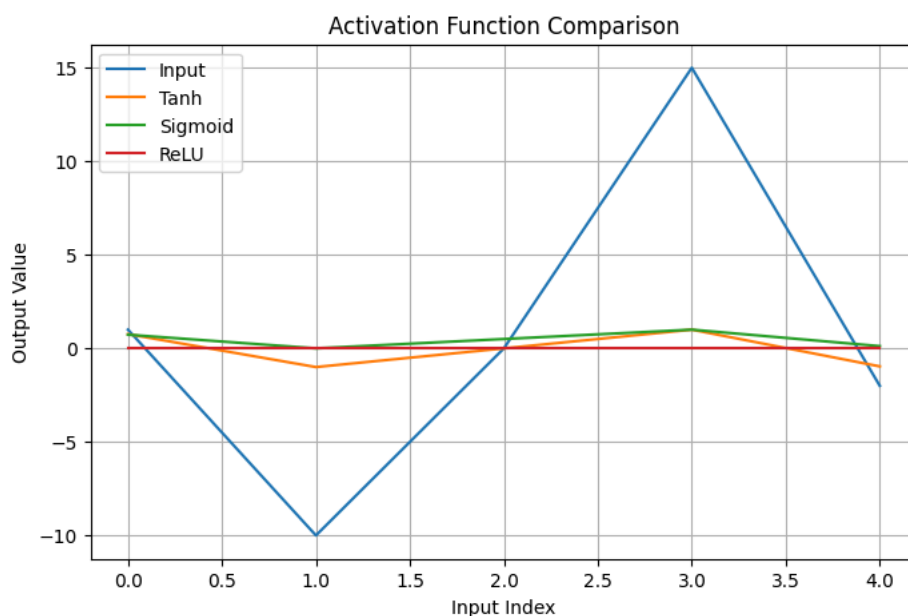
```
↗ Tanh Output: [0.7615941559557649, -0.9999999958776926, 0.0, 0.999999999998128, -0.964027580075817]
Sigmoid Output: [0.7310585786300049, 4.5397868702434395e-05, 0.5, 0.999999694097773, 0.11920292202211755]
```

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

```
plt.figure(figsize=(8, 5))
plt.plot(x1, label='Input')
plt.plot(tanhOutput, label='Tanh')
plt.plot(SigmoidOutput, label='Sigmoid')
plt.plot(ReLUOutput, label='ReLU')
```

```
plt.legend(loc="upper left")
plt.title("Activation Function Comparison")
plt.xlabel("Input Index")
plt.ylabel("Output Value")
plt.grid(True)
plt.show()
```

```
↗
```



## ✓ Multilayer perceptron for XOR

```

X1=[0,0,1,1]
X2=[0,1,0,1]
y=[0,1,1,0]
w1=0
w2=1
w3=1
w4=0
w5=1
w6=-1
y_pred = [0, 0, 0, 0]
z1 = [0, 0, 0, 0]
z2 = [0, 0, 0, 0]

for i in range(4):
    z1[i] = (X1[i] * w1) + (X2[i] * w3) # First weighted sum
    z2[i] = (X2[i] * w4) + (X1[i] * w2) # Second weighted sum

    # Conditional prediction
    if ((z1[i] * w5) + (z2[i] * w6) < 0):
        y_pred[i] = 1
    else:
        y_pred[i] = (z1[i] * w5) + (z2[i] * w6) # Original value if ≥ 0

print("z1:", z1)
print("z2:", z2)
print("Predictions:", y_pred)
print(test(y, y_pred))

```

↩

```

z1: [0, 1, 0, 1]
z2: [0, 0, 1, 1]
Predictions: [0, 1, 1, 0]
('Correct', -1)

```

## 5 ✓ Weights and Bias effect on Output

Effect of weight on network

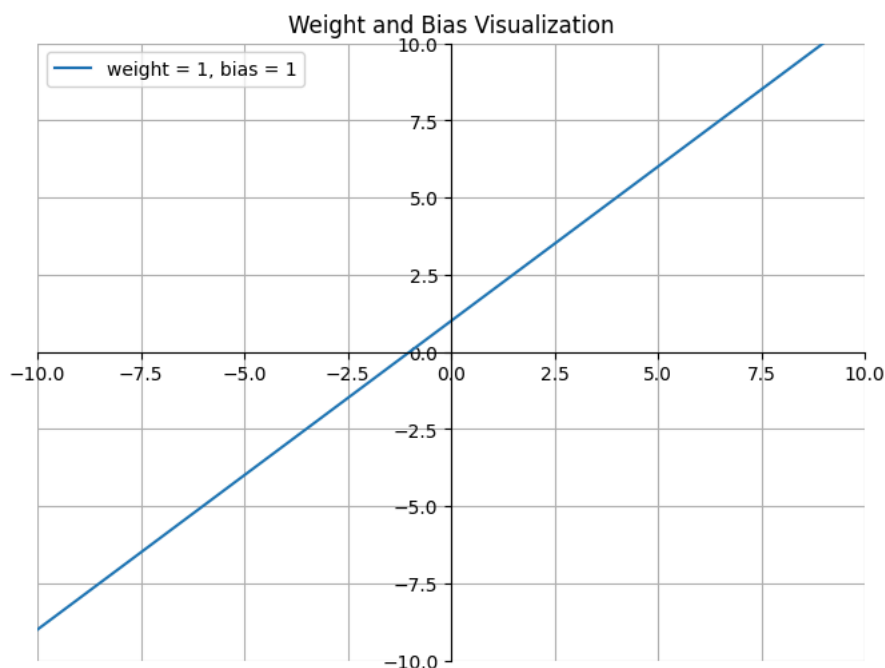
```
import matplotlib.pyplot as plt
weight = 1
bias = 1
x = range(-10, 11)
y = [weight * i + bias for i in x]

legend_label = f"weight = {weight}, bias = {bias}"

plt.figure(figsize=(8, 6)) # Added for better visualization
plt.plot(x, y, label=legend_label)
plt.xlim(-10, 10)
plt.ylim(-10, 10)

# Move spines to center
ax = plt.gca()
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none')

plt.title("Weight and Bias Visualization")
plt.legend()
plt.grid(True)
plt.show()
```



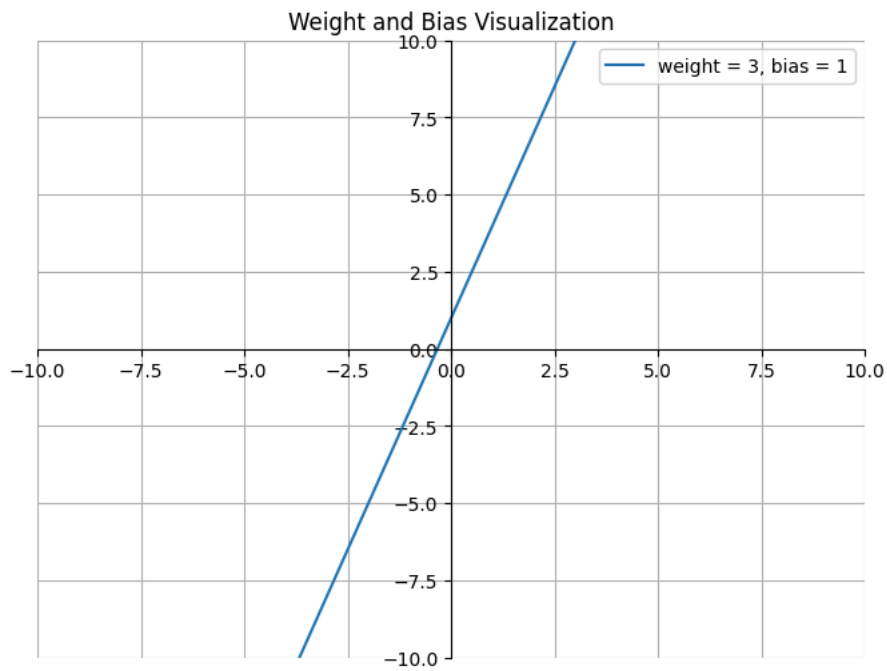
```
weight = 3
bias = 1
x = range(-10, 11)
y = [weight * i + bias for i in x] # Fixed variable name (1 → i)

legend_label = f"weight = {weight}, bias = {bias}"

# Plot configuration
plt.figure(figsize=(8, 6))
plt.plot(x, y, label=legend_label)
plt.xlim(-10, 10)
plt.ylim(-10, 10)

# Center the axes
ax = plt.gca()
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none') # Hide right spine
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none') # Hide top spine
```

```
# Add labels and grid
plt.title("Weight and Bias Visualization")
plt.legend()
plt.grid(True)
plt.show()
```



```
# Define parameters and generate data
weight = -2
bias = 1
x = range(-10, 11)
y = [weight * i + bias for i in x]

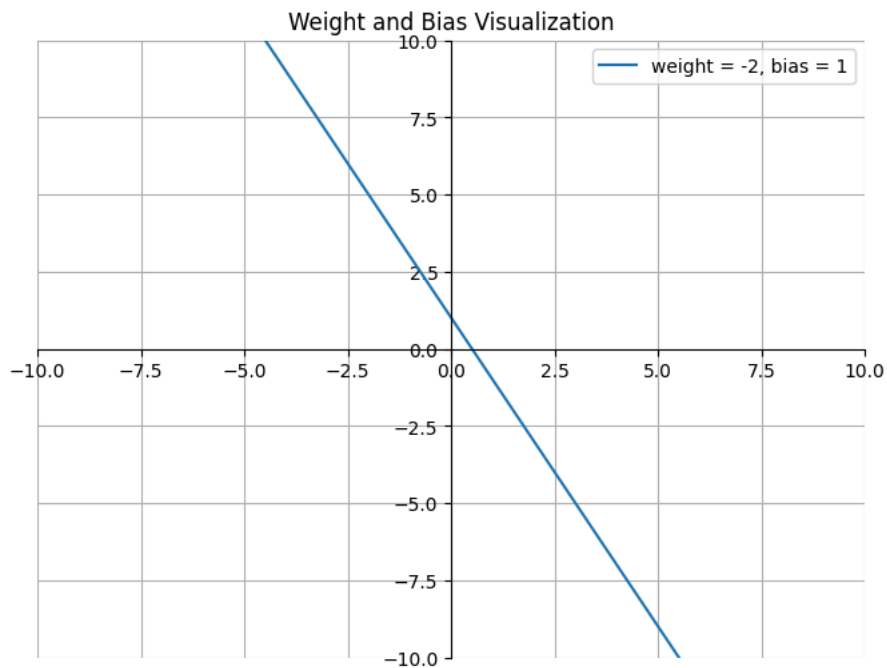
# Corrected f-string (was using parentheses instead of curly braces)
legend_label = f"weight = {weight}, bias = {bias}"

# Plot configuration
plt.figure(figsize=(8, 6))
plt.plot(x, y, label=legend_label)
plt.xlim(-10, 10)
plt.ylim(-10, 10)

# Center the axes
ax = plt.gca()
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none') # Hide right spine
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none') # Hide top spine

# Add labels and grid
plt.title("Weight and Bias Visualization")
plt.legend()
plt.grid(True)
plt.show()
```



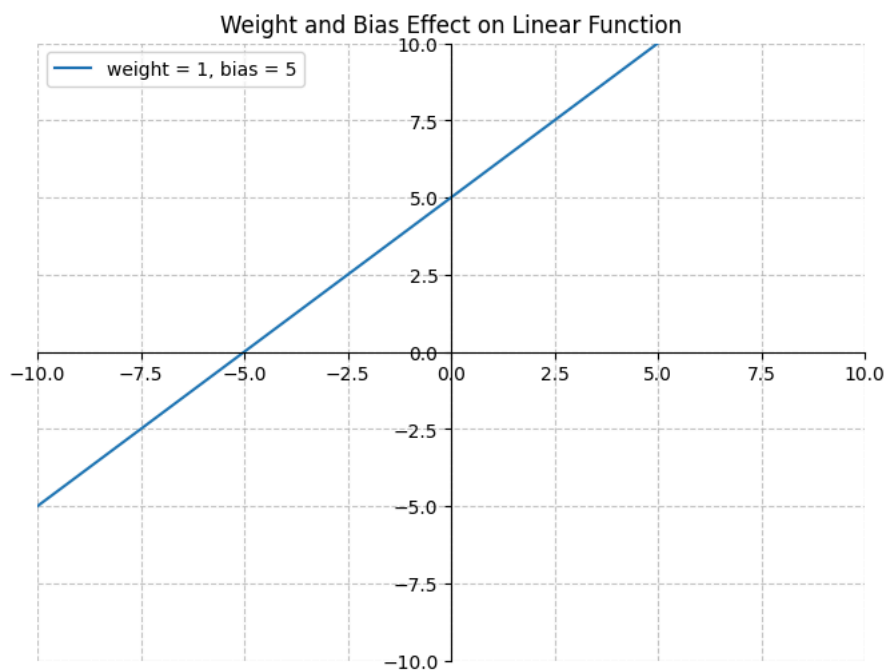


```
weight = 1
bias = 5
x = range(-10, 11)
y = [weight * i + bias for i in x] # Linear equation: y = 1x + 5
```

```
# Plot configuration
plt.figure(figsize=(8, 6))
plt.plot(x, y, label=f"weight = {weight}, bias = {bias}")
plt.xlim(-10, 10)
plt.ylim(-10, 10)
```

```
# Axis customization
ax = plt.gca()
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none') # Hide right spine
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none') # Hide top spine
```

```
# Labels and styling
plt.title("Weight and Bias Effect on Linear Function")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.show()
```



```
# Parameters and linear function definition
```

```

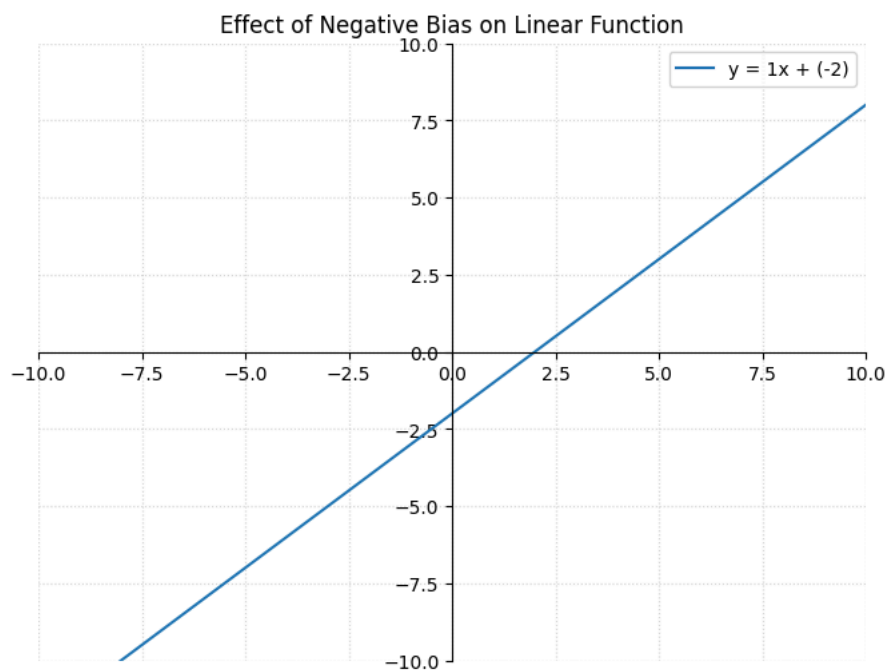
weight = 1
bias = -2
x = range(-10, 11)
y = [weight * i + bias for i in x] # y = 1x - 2

# Create plot
plt.figure(figsize=(8, 6))
plt.plot(x, y, label=f"y = {weight}x + ({bias})") # Improved label format
plt.xlim(-10, 10)
plt.ylim(-10, 10)

# Center axes and customize spines
ax = plt.gca()
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none') # Hide right spine
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none') # Hide top spine

# Add labels and grid
plt.title("Effect of Negative Bias on Linear Function")
plt.legend()
plt.grid(True, linestyle=':', alpha=0.5) # Dotted grid lines
plt.show()

```



## Conclusion

- Weights determine the relative importance of each input feature by adjusting/shifting the slope of the decision line. As on decreasing the weights its slope increases and on negative weights it becomes negative.
- Bias acts as a constant threshold. On increasing and decreasing the bias, the decision line moves upwards or downwards.

## 6. Implementation of different learning rules

6

### Learning Rules in ANN

#### 5 types of Learning Rules:

1. Hebbian Learning Rule
2. Error Correction
3. Memory Based
4. Competitive
5. Boltzmann

#### 1. Hebbian Learning Rule

- Set all weights to zero, ( $w_i = 0$ ) for ( $i = 1$ ) to ( $n$ ), and bias to zero.
- For each input vector, ( $S$ ) (input vector) : ( $t$ ) (target output pair), repeat steps 3-5.
- Set activations for input units with the input vector ( $X_i = S_i$ ) for ( $i = 1$ ) to ( $n$ ).
- Set the corresponding output value to the output neuron, i.e., ( $y = t$ ).
- Update weight and bias by applying the Hebb rule for all ( $i = 1$ ) to ( $n$ ):

$$w_i(\text{new}) = w_i(\text{old}) + x_i y$$

$$b(\text{new}) = b(\text{new}) + y$$

#### OR GATE using Hebbian Rule

```
import numpy as np
x1 = np.array([1,1,-1,-1])
x2 = np.array([1,-1,1,-1])
t=np.array([1,1,1,-1])
w1=0
w2=0
bias=0
def hebbian_learning(x1, x2, t, w1, w2, bias):
    # Update weights
    w1 += np.dot(x1, t)
    w2 += np.dot(x2, t)
    bias += np.sum(t)
    return w1, w2, bias

print("x1\tx2\tt\tty_pred")
for i in range(len(x1)):
    y_pred = bias + w1 * x1[i] + w2 * x2[i]
    print(x1[i], "\t", x2[i], "\t", t[i], "\t", y_pred)
    if y_pred != t[i]:
        # print("Incorrect prediction, updating weights...")
        w1 = w1 + x1[i] * t[i]
        w2 = w2 + x2[i] * t[i]
        bias = bias + t[i]
    print("Updated w1:", w1, " Updated w2:", w2, " Updated bias:", bias)
```

```
↗ x1      x2      t      y_pred
1         1         1         0
Updated w1: 1  Updated w2: 1  Updated bias: 1
1        -1         1         1
-1         1         1         1
-1        -1        -1        -1
```

#### AND GATE using Hebbian Rule

```
import numpy as np
x1 = np.array([1, 1, -1, -1])
x2 = np.array([1, -1, 1, -1])
t = np.array([1, -1, -1, -1])
w1 = 0
w2 = 0
bias = 0
```

```

def threshold(y_pred):
    for i in range(len(y_pred)):
        if y_pred[i] < 0:
            y_pred[i] = -1
        else:
            y_pred[i] = 1

def hebbian_learning(x1, x2, t, w1, w2, bias):
    # Update weights
    w1 += np.dot(x1, t)
    w2 += np.dot(x2, t)
    bias += np.sum(t)
    return w1, w2, bias

for epoch in range(2):
    print("Epoch", epoch)
    print("x1\tx2\tt\tty_pred")
    for i in range(len(x1)):
        y_pred = bias + w1 * x1[i] + w2 * x2[i]
        print(x1[i], "\t", x2[i], "\t", t[i], "\t", y_pred)
        if y_pred != t[i]:
            w1 = w1 + x1[i] * t[i]
            w2 = w2 + x2[i] * t[i]
            bias = bias + t[i]
        print("Updated w1:", w1, " Updated w2:", w2, " Updated bias:", bias)

# Convert predictions to binary
y_pred = bias + w1 * x1 + w2 * x2
threshold(y_pred)
print("\nPredictions after thresholding:", y_pred)

```

```

↔ Epoch 0
x1      x2      t      y_pred
1        1        1        0
Updated w1: 1 Updated w2: 1 Updated bias: 1
1       -1       -1        1
Updated w1: 0 Updated w2: 2 Updated bias: 0
-1        1       -1        2
Updated w1: 1 Updated w2: 1 Updated bias: -1
-1       -1       -1       -3
Updated w1: 2 Updated w2: 2 Updated bias: -2
Epoch 1
x1      x2      t      y_pred
1        1        1        2
Updated w1: 3 Updated w2: 3 Updated bias: -1
1       -1       -1       -1
-1        1       -1       -1
-1       -1       -1       -7
Updated w1: 4 Updated w2: 4 Updated bias: -2

Predictions after thresholding: [ 1 -1 -1 -1]

```

## Adaline Learning Rule

Step 1: Initialize weight not zero but small random values are used. Set learning rate  $\alpha$ .

Step 2: While the stopping condition is False do steps 3 to 7.

Step 3: For each training set perform steps 4 to 6.

Step 4: Set activation of input unit  $x_i = s_i$  for ( $i = 1$  to  $n$ ).

Step 5: Compute net input to output unit  $y_{in} = \sum(w_i * x_i) + b$

```
# Here, b is the bias and n is the total number of neurons.
```

Step 6: Update the weights and bias for  $i = 1$  to  $n$   $w_i(\text{new}) = w_i(\text{old}) + \eta * (t - y_{in}) * x_i$   $b(\text{new}) = b(\text{old}) + \alpha * (t - y_{in})$

```

# and calculate the error:
error = (t - y_in)^2
# When the predicted output and the true value are the same, then the weight will not change.

```

Step 7: Test the stopping condition.

```
# The stopping condition may be when the weight changes at a low rate or no change.
```

## ✓ OR GATE using Adaline Learning Rule

```
import numpy as np

# Bipolar OR gate input patterns and expected outputs
x1 = np.array([1, 1, -1, -1])
x2 = np.array([1, -1, 1, -1])
t = np.array([1, 1, 1, -1])

# Initialize weights, bias, learning rate, total_error, and iteration
w1 = 0.1
w2 = 0.1
b = 0.1
eta = 0.1
total_error = 0
iteration = 0

# OR Gate using Addline Learning rate
print("Iteration\tInput\tTarget\tYin\tError\tw1\tw2\tBias\tFinal Error\tTotal Error")
for j in range(3):
    total_error = 0
    for i in range(4):
        y_in = b + w1 * x1[i] + w2 * x2[i]
        error = t[i] - y_in
        final_error = error**2
        total_error += final_error
        w1 += eta * error * x1[i]
        w2 += eta * error * x2[i]
        b += eta * error

    print(f"{iteration+1}\t{x1[i], x2[i]}\t{t[i]}\t{y_in:.4f}\t{error:.4f}\t{w1:.4f}\t{w2:.4f}\t{b:.4f}\t{final_error:.4f}\t{total_error}")
    iteration += 1
    if total_error <= 2:
        break

print("\nFinal Weights and Bias:")
print(f"w1: {w1:.4f}")
print(f"w2: {w2:.4f}")
print(f"Bias: {b:.4f}")
```



Iteration	Input	Target	Yin	Error	W1	W2	Bias	Final Error	Total Error
1	(np.int64(1), np.int64(1))	1	0.3000	0.7000	0.1700	0.1700	0.4900	0.4900	
2	(np.int64(1), np.int64(1))	1	0.5100	0.4900	0.2190	0.2190	0.2401	0.2401	
3	(np.int64(1), np.int64(1))	1	0.6570	0.3430	0.2533	0.2533	0.1176	0.1176	

```
Final Weights and Bias:
w1: 0.2533
w2: 0.2533
Bias: 0.2533
```

## ✓ AND GATE using ADALINE Learning Rule

```
import numpy as np

# Bipolar OR gate input patterns and expected outputs
x1 = np.array([1, 1, -1, -1])
x2 = np.array([1, -1, 1, -1])
t = np.array([1, -1, -1, -1])
w1 = 0.1
w2 = 0.1
b = 0.1
eta = 0.1

total_error = 0
iteration = 0

# AND Gate using Adaline Learning rate
print("Iteration\tInput\tTarget\tYin\tError\tw1\tw2\tBias\tFinal Error\tTotal Error")
for j in range(3):
    total_error = 0
    for i in range(4):
        y_in = b + w1 * x1[i] + w2 * x2[i]
        error = t[i] - y_in
        final_error = error**2
        total_error += final_error
        w1 += eta * error * x1[i]
        w2 += eta * error * x2[i]
        b += eta * error
```

```

    print(f"{iteration+1}\t{x1[i], x2[i]}\t{t[i]}\t{y_in:.4f}\t{error:.4f}\t{w1:.4f}\t{w2:.4f}\t{b:.4f}\t{final_error:.4f}\t{total_
iteration += 1
if total_error <= 2:
    break

print("\nFinal Weights and Bias:")
print(f"w1: {w1:.4f}")
print(f"w2: {w2:.4f}")
print(f"Bias: {b:.4f}")

```

Iteration	Input	Target	Yin	Error	W1	W2	Bias	Final Error	Total Error
1	(np.int64(1), np.int64(1))	1	0.3000	0.7000	0.1700	0.1700	0.1700	0.1700	0.4900
1	(np.int64(1), np.int64(-1))	-1	0.1700	-1.1700	0.0530	0.2870	0.0530	0.0530	1.3689
1	(np.int64(-1), np.int64(1))	-1	0.2870	-1.2870	0.1817	0.1583	-0.0757	1.6564	3.5153
1	(np.int64(-1), np.int64(-1))	-1	-0.4157	-0.5843	0.2401	0.2167	-0.1341	0.3414	3.8567
2	(np.int64(1), np.int64(1))	1	0.3227	0.6773	0.3079	0.2845	-0.0664	0.4587	0.4587
2	(np.int64(-1), np.int64(-1))	-1	-0.0430	-0.9570	0.2122	0.3802	-0.1621	0.9158	1.3745
2	(np.int64(-1), np.int64(1))	-1	0.0059	-1.0059	0.3127	0.2796	-0.2627	1.0118	2.3864
2	(np.int64(-1), np.int64(-1))	-1	-0.8550	-0.1450	0.3272	0.2941	-0.2772	0.0210	2.4074
3	(np.int64(1), np.int64(1))	1	0.3441	0.6559	0.3928	0.3597	-0.2116	0.4302	0.4302
3	(np.int64(1), np.int64(-1))	-1	-0.1784	-0.8216	0.3107	0.4418	-0.2938	0.6750	1.1052
3	(np.int64(-1), np.int64(1))	-1	-0.1626	-0.8374	0.3944	0.3581	-0.3775	0.7012	1.8064
3	(np.int64(-1), np.int64(-1))	-1	-1.1300	0.1300	0.3814	0.3451	-0.3645	0.0169	1.8233

```

Final Weights and Bias:
w1: 0.3814
w2: 0.3451
Bias: -0.3645

```

### 3. Memory based learning

Memory-based learning in artificial neural networks (ANNs) involves storing and utilizing past experiences or training examples directly rather than learning explicit parameters. This approach is also known as instance-based learning or lazy learning. One of the most popular memory-based learning algorithms is the k-nearest neighbors (k-NN) algorithm.

```

x1 = [0, 0, 0, 1, 1, 1]
x2 = [0, 0, 1, 1, 0, 1]
x3 = [0, 1, 0, 1, 0, 1]

y = [0, 0, 0, 1, 0, 1]

x_test = [1, 0, 1]

import matplotlib.pyplot as plt

# Separate the data points based on their labels
class_0_x1 = [x1[i] for i in range(len(x1)) if y[i] == 0]
class_0_x2 = [x2[i] for i in range(len(x2)) if y[i] == 0]
class_0_x3 = [x3[i] for i in range(len(x3)) if y[i] == 0]

class_1_x1 = [x1[i] for i in range(len(x1)) if y[i] == 1]
class_1_x2 = [x2[i] for i in range(len(x2)) if y[i] == 1]
class_1_x3 = [x3[i] for i in range(len(x3)) if y[i] == 1]

# Plot the data points
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(class_0_x1, class_0_x2, class_0_x3, c='blue', label='Class 0')
ax.scatter(class_1_x1, class_1_x2, class_1_x3, c='red', label='Class 1')
ax.scatter(x_test[0], x_test[1], x_test[2], c='green', marker='x', label='Test Point')

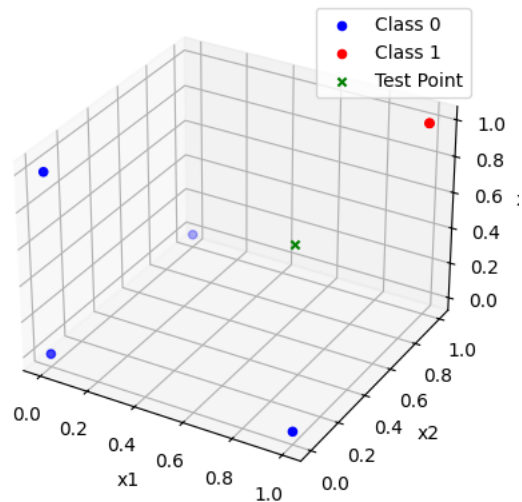
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('x3')

plt.legend()
plt.title('Data Points')
plt.show()

```



Data Points



```

n = len(x1)
y_eucli = [0,0,0,0,0,0]
for i in range(n):
    y_eucli[i] = ((x1[i]-x_test[0])**2 + (x2[i]-x_test[1])**2 + (x3[i]-x_test[2])**2)**0.5

# Combine distances with labels
combined_data = list(zip(y, y_eucli))

# Sort the combined data based on distances
sorted_data = sorted(combined_data, key=lambda x: x[1])

k = 3
nearest_neighbors = sorted_data[:k]

# Extract coordinates of k nearest neighbors
nearest_neighbor_indices = [x[0] for x in nearest_neighbors]
nearest_neighbor_coords = [(x1[i], x2[i], x3[i]) for i in nearest_neighbor_indices]

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(x1, x2, x3, c=y, cmap='coolwarm', label='Data Points')
ax.scatter(x_test[0], x_test[1], x_test[2], c='green', marker='x', label='Test Point')

for i in range(k):
    ax.scatter(nearest_neighbor_coords[i][0], nearest_neighbor_coords[i][1], nearest_neighbor_coords[i][2],
               c='red', s=100, label=f'Neighbor {i+1}')
    ax.plot([x_test[0], nearest_neighbor_coords[i][0]],
            [x_test[1], nearest_neighbor_coords[i][1]],
            [x_test[2], nearest_neighbor_coords[i][2]], c='black')

ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('x3')

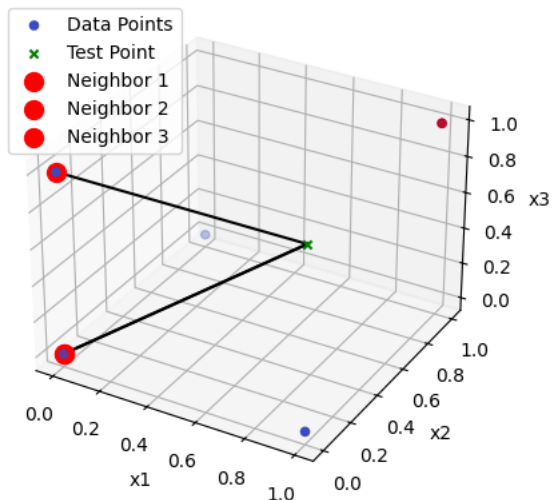
plt.legend()
plt.title('Data Points with Nearest Neighbors and Connecting Lines')
plt.show()

# Print distances
print("y", "y_euclidean")
for label, distance in sorted_data:
    print(label, distance)

```



## Data Points with Nearest Neighbors and Connecting Lines



```

y y_euclidean
0 1.0
1 1.0
0 1.0
1 1.0
0 1.4142135623730951
1 1.7320508075688772

```

```
import plotly.graph_objects as go
```

```
# Create traces for data points, test point, and connecting lines
```

```
data_points = go.Scatter3d(x=x1, y=x2, z=x3, mode='markers', marker=dict(color=y, size=8))
```

```
test_point = go.Scatter3d(x=[x_test[0]], y=[x_test[1]], z=[x_test[2]], mode='markers', marker=dict(color='green', size=10, symbol='x'))
```

```
neighbors = []
```

```
for i, coord in enumerate(nearest_neighbor_coords):
```

```
    neighbors.append(go.Scatter3d(x=[x_test[0], coord[0]], y=[x_test[1], coord[1]], z=[x_test[2], coord[2]],
                                mode='lines', line=dict(color='black')))
```

```
fig = go.Figure(data=[data_points, test_point] + neighbors)
```

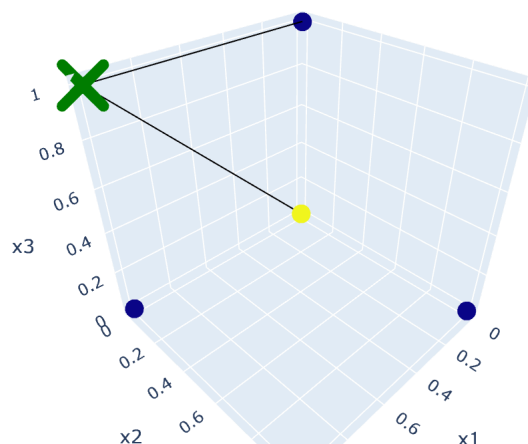
```
# Update layout
```

```
fig.update_layout(scene=dict(xaxis_title='x1', yaxis_title='x2', zaxis_title='x3'), title='3D k-NN Visualization')
```

```
fig.show()
```



## 3D k-NN Visualization



## Competitive Learning Rule



```
c_x1 = [0.2, 0.6, 0.4, 0.9, 0.2]
c_x2 = [0.3, 0.5, 0.7, 0.6, 0.8]
x1 = 0.3
x2 = 0.4
```

```
d = [0 for i in range(len(c_x1))] # list representing distance square d**2
```

```
for i in range(len(c_x1)):
    d[i] = ((c_x1[i] - x1)**2 + (x2 - c_x2[i])**2)
```

```
print(d)
min_value = min(d)
min_index = d.index(min_value)
```

```
print("Minimum value:", min_value)
print("Cluster of minimum value:", min_index + 1)
```

```
→ [0.020000000000000004, 0.09999999999999999, 0.09999999999999996, 0.4000000000000001, 0.17000000000000004]
    Minimum value: 0.020000000000000004
    Cluster of minimum value: 1
```

```
eta = 0.3
c_x1[min_index] = c_x1[min_index] + eta * (x1 - c_x1[min_index])
c_x2[min_index] = c_x2[min_index] + eta * (x2 - c_x2[min_index])
```

```
print(c_x1)
print(c_x2)
```

```
→ [0.23, 0.6, 0.4, 0.9, 0.2]
    [0.33, 0.5, 0.7, 0.6, 0.8]
```

```
d = [0 for i in range(len(c_x1))] # list representing distance square d**2
```

```
for i in range(len(c_x1)):
    d[i] = ((c_x1[i] - x1)**2 + (x2 - c_x2[i])**2)
```

```
print(d)
min_value = min(d)
min_index = d.index(min_value)
```

```
print("Minimum value:", min_value)
print("Cluster of minimum value:", min_index + 1)
```

```
→ [0.020000000000000004, 0.09999999999999999, 0.09999999999999996, 0.4000000000000001, 0.17000000000000004]
```

## ✓ 7. Implementation of Perceptron Networks.

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(9)

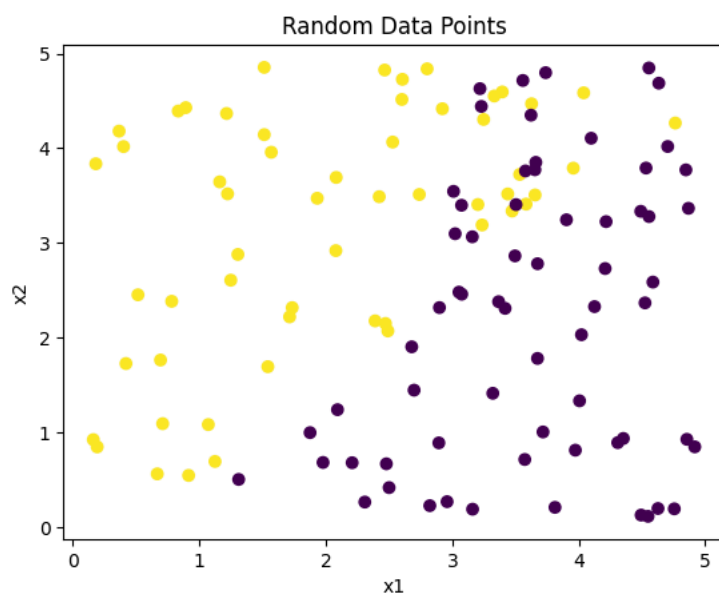
N = 100
X = np.random.rand(N, 2) * 5

y = np.sign(X[:, 1] - X[:, 0] + 0.5)

outlier_ratio = 0.2
outlier_x = np.random.rand(int(N * outlier_ratio), 2) * 2 + 3
outlier_y = np.ones(int(N * outlier_ratio)) * -1
X = np.vstack((X, outlier_x))
y = np.concatenate((y, outlier_y))

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')

plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Random Data Points')
plt.show()
```

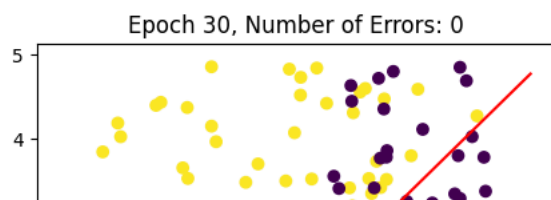
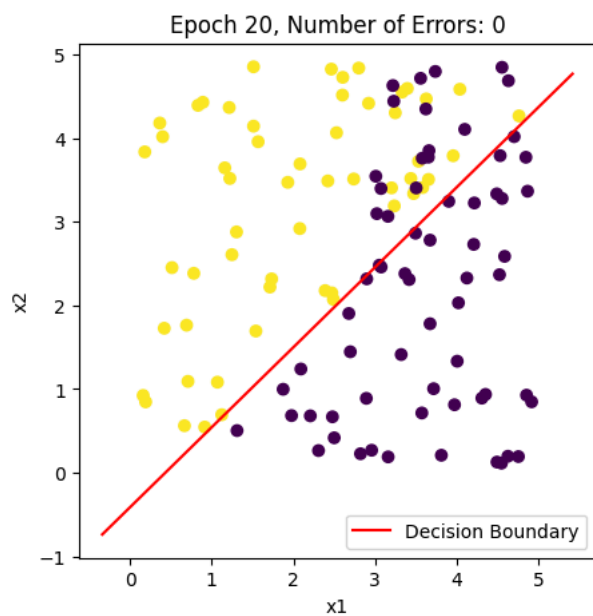
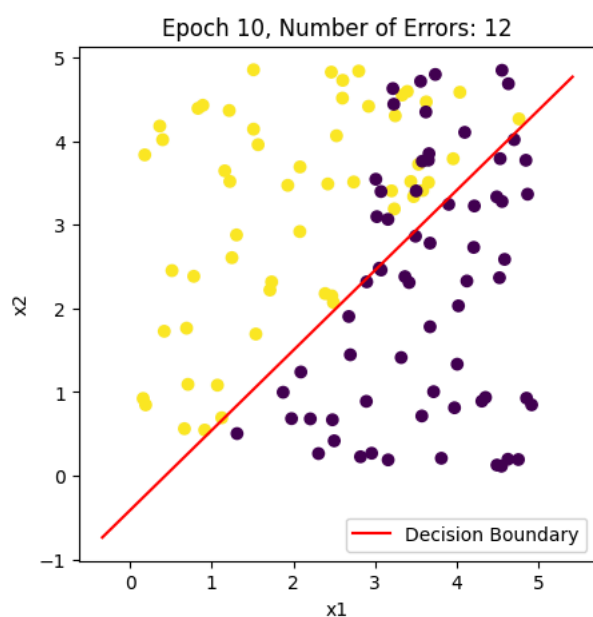
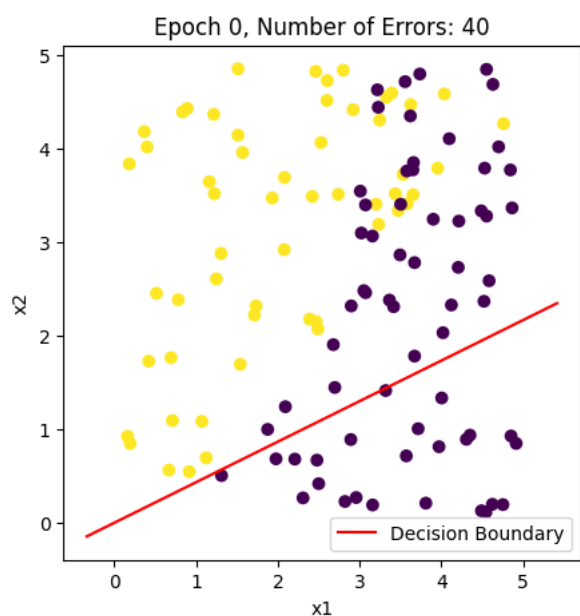


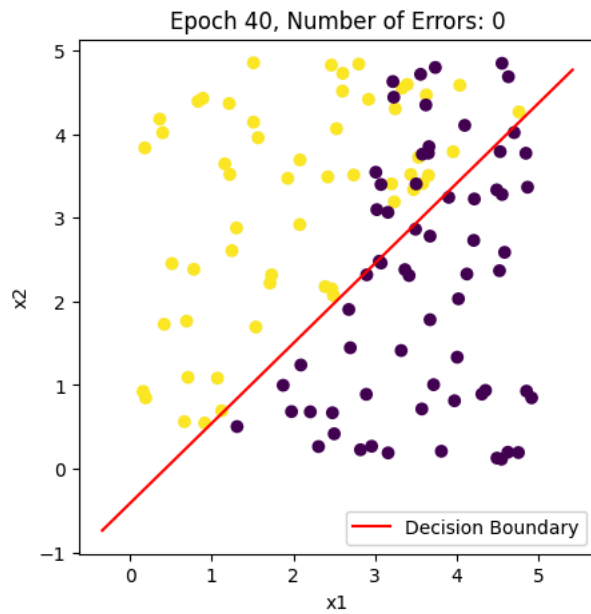
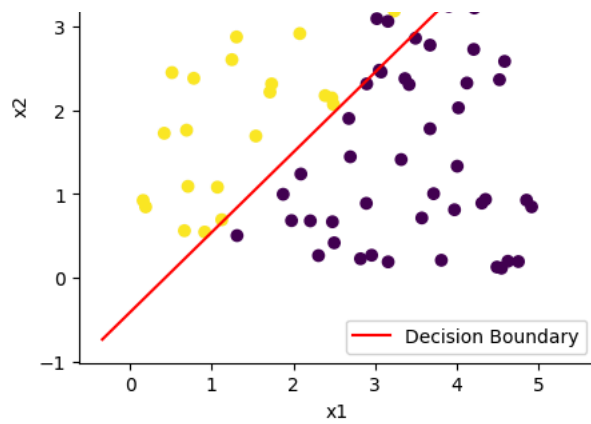
```
eta = 0.1
epochs = 50
w = np.random.rand(2)
bias = 0

for epoch in range(epochs):
    error_count = 0
    for i in range(N):
        activation = np.dot(w, X[i]) + bias
        prediction = np.sign(activation)
        if prediction != y[i]:
            error_count += 1
            w += eta * y[i] * X[i]
            bias += eta * y[i]

    if epoch % 10 == 0:
        plt.figure(figsize=(5, 5))
        plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')
        x_span = np.linspace(min(X[:, 0]) - 0.5, max(X[:, 0]) + 0.5)
        y_span = -(bias + w[0] * x_span) / w[1]

        plt.plot(x_span, y_span, color='red', label='Decision Boundary')
        plt.xlabel('x1')
        plt.ylabel('x2')
        plt.title(f'Epoch {epoch}, Number of Errors: {error_count}')
        plt.legend()
        plt.show()
```





## Conclusion

This Perceptron networks classify linearly seperable data by plotting decision boundary.

Perceptron train by each data point based on error it update weights and adust decision boundary.

## 8. Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

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

```
x1 = 0.35
x2 = 0.9
t = 0.5
w13 = 0.1
w14 = 0.4
w23 = 0.8
w24 = 0.6
w45 = 0.9
w35 = 0.3
```

```
v3 = x1 * w13 + x2 * w23
y3 = sigmoid(v3)
```

```
v4 = x1 * w14 + x2 * w24
y4 = sigmoid(v4)
```

```
v5 = y3 * w35 + y4 * w45
y5 = sigmoid(v5)
```

```
error = t - y5
```

```
print("Activations:")
print(f"y3 = {y3}, y4 = {y4}, y5 = {y5}")
print("Error:")
print(f"Error = {error}")
```

```
↗ Activations:
y3 = 0.6802671966986485, y4 = 0.6637386974043528, y5 = 0.6902834929076443
Error:
Error = -0.19028349290764435
```

```
delta5 = y5 * (1 - y5) * error
```

```
delta3 = y3 * (1 - y3) * (delta5 * w35)
delta4 = y4 * (1 - y4) * (delta5 * w45)
```

```
print("Local Gradients:")
print(f"delta3 = {delta3}, delta4 = {delta4}, delta5 = {delta5}")
```

```
↗ Local Gradients:
delta3 = -0.002654489030884742, delta4 = -0.00817164506412987, delta5 = -0.04068112511233903
```

```
eta = 1
w35 += eta * y3 * delta5
w45 += eta * y4 * delta5
w13 += eta * x1 * delta3
w14 += eta * x1 * delta4
w23 += eta * x2 * delta3
w24 += eta * x2 * delta4
```

```
print("Updated Weights:")
print(f"w13 = {w13}, w14 = {w14}, w23 = {w23}, w24 = {w24}, w35 = {w35}, w45 = {w45}")
```

```
↗ Updated Weights:
w13 = 0.09907092883919034, w14 = 0.39713992422755456, w23 = 0.7976109598722038, w24 = 0.592645519442283, w35 = 0.27232596506128215,
```

```
v3 = x1 * w13 + x2 * w23
y3 = sigmoid(v3)
```



```
v4 = x1 * w14 + x2 * w24
y4 = sigmoid(v4)
```

```
v5 = y3 * w35 + y4 * w45
y5 = sigmoid(v5)
```

```
error = t - y5
```

```
print("Epoch 2 Activations:")
print(f"y3 = {y3}, y4 = {y4}, y5 = {y5}")
print("Epoch 2 Error:")
print(f"Error = {error}")
```

```
Epoch 2 Activations:
y3 = 0.6797285672285043, y4 = 0.662035862797593, y5 = 0.6820185832642942
Epoch 2 Error:
Error = -0.1820185832642942
```

```
def forward_pass(x1, x2, w13, w14, w23, w24, w35, w45):
    v3 = x1 * w13 + x2 * w23
    y3 = sigmoid(v3)
```

```
    v4 = x1 * w14 + x2 * w24
    y4 = sigmoid(v4)
```

```
    v5 = y3 * w35 + y4 * w45
    y5 = sigmoid(v5)
```

```
    return y3, y4, y5
```

```
def calculate_error(y5, t):
    error = t - y5
    return error
```

```
def calculate_local_gradients(y3, y4, y5, w35, w45, error):
    delta5 = y5 * (1 - y5) * error
    delta3 = y3 * (1 - y3) * (delta5 * w35)
    delta4 = y4 * (1 - y4) * (delta5 * w45)

    return delta3, delta4, delta5
```

```
def update_weights(x1, x2, delta3, delta4, delta5, learning_rate, w13, w14, w23, w24, w35, w45, y3, y4):
    w35 += learning_rate * y3 * delta5
    w45 += learning_rate * y4 * delta5
    w13 += learning_rate * x1 * delta3
    w14 += learning_rate * x1 * delta4
    w23 += learning_rate * x2 * delta3
    w24 += learning_rate * x2 * delta4

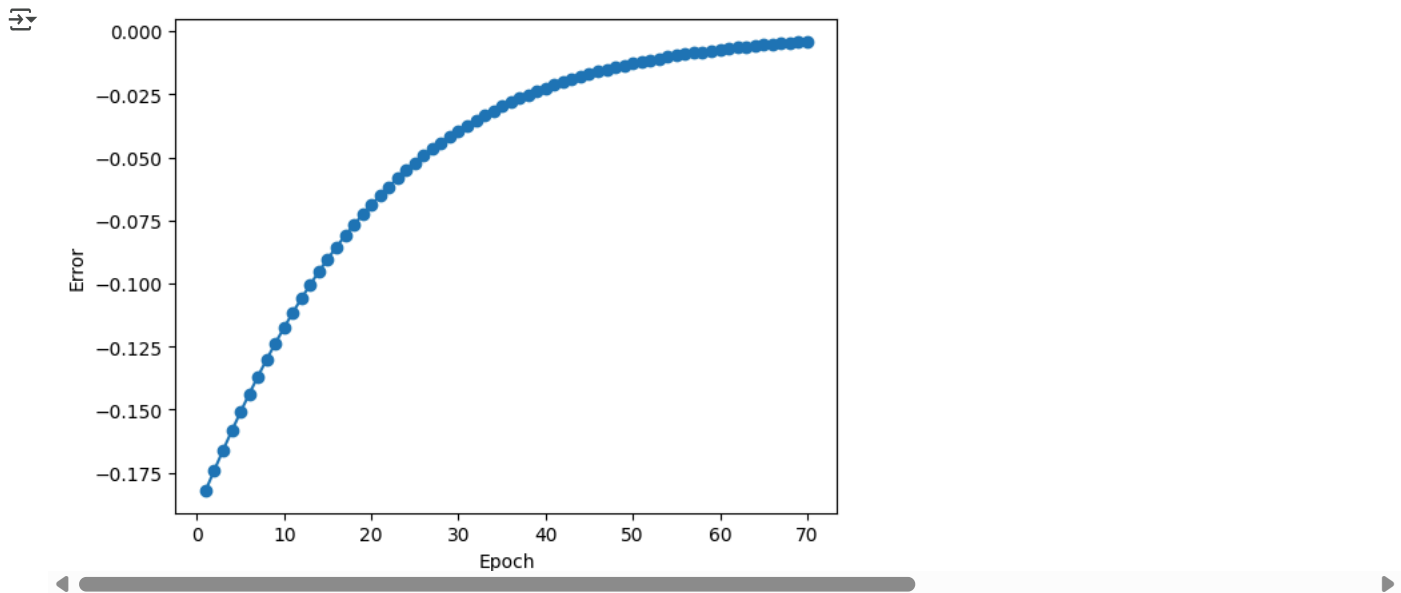
    return w13, w14, w23, w24, w35, w45
```

```
errors = []
```

```
def train_one_epoch(x1, x2, w13, w14, w23, w24, w35, w45, t, eta):
    y3, y4, y5 = forward_pass(x1, x2, w13, w14, w23, w24, w35, w45)
    error = calculate_error(y5, t)
    delta3, delta4, delta5 = calculate_local_gradients(y3, y4, y5, w35, w45, error)
    w13, w14, w23, w24, w35, w45 = update_weights(x1, x2, delta3, delta4, delta5, eta, w13, w14, w23, w24, w35, w45, y3, y4)
    return error, w13, w14, w23, w24, w35, w45
```

```
for epoch in range(70):
    error, w13, w14, w23, w24, w35, w45 = train_one_epoch(x1, x2, w13, w14, w23, w24, w35, w45, t, eta)
    errors.append(error)
```

```
plt.plot(range(1, 71), errors, marker='o')
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.show()
```



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

```
x1 = 0.35
x2 = 0.9
w13 = 0.1
w14 = 0.4
w23 = 0.8
w24 = 0.6
w35 = 0.3
w45 = 0.9
t = 0.5
learning_rate = 1
alpha = 0.5
```

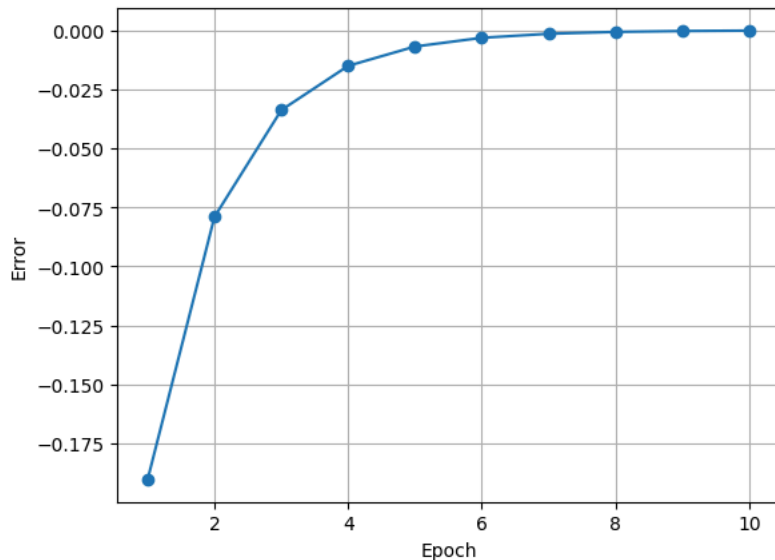
```
def update_weights(x1, x2, delta3, delta4, delta5, learning_rate, w13, w14, w23, w24, w35, w45, y3, y4, alpha):
    w35 = alpha * w35 + learning_rate * y3 * delta5
    w45 = alpha * w45 + learning_rate * y4 * delta5
    w13 = alpha * w13 + learning_rate * x1 * delta3
    w14 = alpha * w14 + learning_rate * x1 * delta4
    w23 = alpha * w23 + learning_rate * x2 * delta3
    w24 = alpha * w24 + learning_rate * x2 * delta4
    return w13, w14, w23, w24, w35, w45
```

```
errors = []
```

```
def train_one_epoch(x1, x2, w13, w14, w23, w24, w35, w45, t, eta, alpha):
    y3, y4, y5 = forward_pass(x1, x2, w13, w14, w23, w24, w35, w45)
    error = calculate_error(y5, t)
    delta3, delta4, delta5 = calculate_local_gradients(y3, y4, y5, w35, w45, error)
    w13, w14, w23, w24, w35, w45 = update_weights(x1, x2, delta3, delta4, delta5, eta, w13, w14, w23, w24, w35, w45, y3, y4, alpha)
    return error, w13, w14, w23, w24, w35, w45
```

```
# Training loop
for epoch in range(10):
    error, w13, w14, w23, w24, w35, w45 = train_one_epoch(x1, x2, w13, w14, w23, w24, w35, w45, t, learning_rate, alpha)
    errors.append(error)
```

```
# Plotting
plt.plot(range(1, 11), errors, marker='o')
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.grid()
plt.show()
```



```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data"
names = ['age', 'workclass', 'fnlwgt', 'education', 'education-num', 'marital-status',
         'occupation', 'relationship', 'race', 'sex', 'capital-gain', 'capital-loss',
         'hours-per-week', 'native-country', 'income']

df = pd.read_csv(url, header=None, names=names, na_values=' ?')

df.dropna(inplace=True)

label_encoders = {}
categorical_cols = df.select_dtypes(include=['object']).columns.tolist()
for col in categorical_cols:
    label_encoders[col] = LabelEncoder()
    df[col] = label_encoders[col].fit_transform(df[col])

X = df.drop('income', axis=1)
y = df['income']

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

print("X_train.shape:", X_train.shape)
print("X_test.shape:", X_test.shape)
print("y_train.shape:", y_train.shape)
print("y_test.shape:", y_test.shape)

X_train.shape: (24129, 14)
X_test.shape: (6033, 14)
y_train.shape: (24129,)
y_test.shape: (6033,)

import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

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

def sigmoid_derivative(x):
    return x * (1 - x)

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1):
```



```

self.input_size = input_size
self.hidden_size = hidden_size
self.output_size = output_size
self.learning_rate = learning_rate

self.weights1 = np.random.randn(self.input_size, self.hidden_size)
self.bias1 = np.zeros((1, self.hidden_size))
self.weights2 = np.random.randn(self.hidden_size, self.output_size)
self.bias2 = np.zeros((1, self.output_size))

self.training_loss = []

def forward(self, X):

    self.hidden_input = np.dot(X, self.weights1) + self.bias1
    self.hidden_output = sigmoid(self.hidden_input)
    self.output = sigmoid(np.dot(self.hidden_output, self.weights2) + self.bias2)
    return self.output

def backward(self, X, y, output):
    error = y - output
    output_delta = error * sigmoid_derivative(output)

    error_hidden = output_delta.dot(self.weights2.T)
    hidden_delta = error_hidden * sigmoid_derivative(self.hidden_output)

    self.weights2 += self.learning_rate * self.hidden_output.T.dot(output_delta)
    self.bias2 += self.learning_rate * np.sum(output_delta, axis=0, keepdims=True)
    self.weights1 += self.learning_rate * X.T.dot(hidden_delta)
    self.bias1 += self.learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)

def train(self, X, y, epochs=1000):
    for epoch in range(epochs):
        output = self.forward(X)
        self.backward(X, y, output)

        loss = np.mean((y - output)**2)
        self.training_loss.append(loss)

def predict(self, X):
    return np.round(self.forward(X))

input_size = X_train.shape[1]
hidden_size = 5

output_size = 1

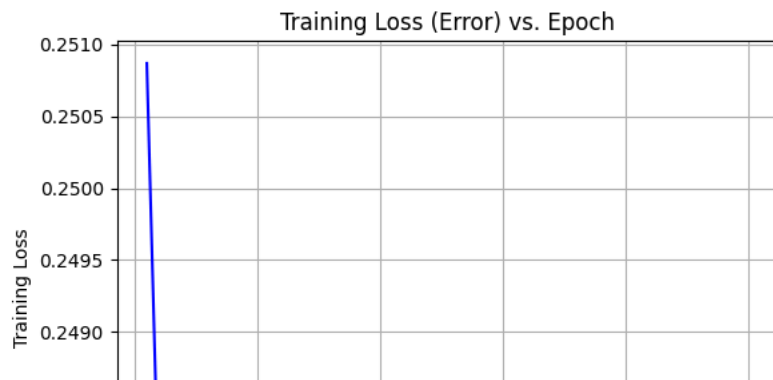
model = NeuralNetwork(input_size, hidden_size, output_size, learning_rate=0.1)
model.train(X_train, y_train.values.reshape(-1, 1), epochs=50)

y_pred = model.predict(X_test)

test_accuracy = accuracy_score(y_test.values.reshape(-1, 1), y_pred)
print("Test Accuracy:", test_accuracy)
print()
plt.plot(range(1, len(model.training_loss) + 1), model.training_loss, color='blue')
plt.title('Training Loss (Error) vs. Epoch')
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.grid(True)
plt.show()

```

↔ Test Accuracy: 0.7463948284435604



9

## ✓ 9. Detecting credit card fraud with neural network

```
import numpy as np
import pandas as pd
import os
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from keras.callbacks import ReduceLROnPlateau, EarlyStopping
from sklearn.metrics import average_precision_score, confusion_matrix
import matplotlib.pyplot as plt
```

```
data = pd.read_csv('creditcard.csv')
data.head()
```

```
↺
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V2
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.11047
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.10128
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.90941
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.19032
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.13745

5 rows × 31 columns

```
data.drop("Time",axis=1,inplace=True)
```

```
data.isnull().any().describe()
```

```
↺
```

	0
count	30
unique	1
top	False
freq	30

dtype: object

## ✓ Training Model

```
limit = int(0.9*len(data))
train = data.loc[:limit]
val_test = data.loc[limit:]
val_test.reset_index(drop=True, inplace=True)
val_test_limit = int(0.5*len(val_test))
val = val_test.loc[:val_test_limit]
test = val_test.loc[val_test_limit:]
```

### Balancing Data

```
train_positive = train[train["Class"] == 1]
train_positive = pd.concat([train_positive] * int(len(train) / len(train_positive)), ignore_index=True)
noise = np.random.uniform(0.9, 1.1, train_positive.shape)
train_positive = train_positive.multiply(noise)
train_positive["Class"] = 1
train_extended = pd.concat([train, train_positive], ignore_index=True)
train_shuffled = train_extended.sample(frac=1, random_state=0).reset_index(drop=True)
```


```
X_train = train_shuffled.drop(labels=["Class"], axis=1)
Y_train = train_shuffled["Class"]
X_val = val.drop(labels=["Class"], axis=1)
Y_val = val["Class"]
X_test = test.drop(labels=["Class"], axis=1)
Y_test = test["Class"]
```

```
# Feature Scaling
scaler = StandardScaler()
X_train[X_train.columns] = scaler.fit_transform(X_train)
X_val[X_val.columns] = scaler.transform(X_val)
X_test[X_test.columns] = scaler.transform(X_test)

# Model Architecture
model = Sequential()
model.add(Dense(64, activation="relu", input_dim=X_train.shape[1]))
model.add(Dense(32, activation="relu"))
model.add(Dense(16, activation="relu"))
model.add(Dense(8, activation="relu"))
model.add(Dense(4, activation="relu"))
model.add(Dense(2, activation="relu"))
model.add(Dense(1, activation="sigmoid"))

# Model Compilation
model.compile(optimizer=Adam(learning_rate=1e-4),
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()

# Model Training
history = model.fit(X_train,
                    Y_train,
                    epochs=10,
                    validation_data=(X_val, Y_val),
                    callbacks=[ReduceLROnPlateau(patience=3, verbose=1, min_lr=1e-6),
                             EarlyStopping(patience=5, verbose=1)])
```

 /usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input\_shape`/`input\_dim` arg  
 super().\_\_init\_\_(activity\_regularizer=activity\_regularizer, \*\*kwargs)  
 Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	1,920
dense_1 (Dense)	(None, 32)	2,080
dense_2 (Dense)	(None, 16)	528
dense_3 (Dense)	(None, 8)	136
dense_4 (Dense)	(None, 4)	36
dense_5 (Dense)	(None, 2)	10
dense_6 (Dense)	(None, 1)	3

Total params: 4,713 (18.41 KB)

Trainable params: 4,713 (18.41 KB)

Non-trainable params: 0 (0.00 B)

Epoch 1/10

16015/16015 ————— 44s 2ms/step - accuracy: 0.9018 - loss: 0.3839 - val\_accuracy: 0.9900 - val\_loss: 0.0212 - learning

Epoch 2/10

16015/16015 ————— 40s 2ms/step - accuracy: 0.9939 - loss: 0.0907 - val\_accuracy: 0.9969 - val\_loss: 0.0106 - learning

Epoch 3/10

16015/16015 ————— 42s 2ms/step - accuracy: 0.9977 - loss: 0.0272 - val\_accuracy: 0.9971 - val\_loss: 0.0129 - learning

Epoch 4/10

16015/16015 ————— 40s 2ms/step - accuracy: 0.9985 - loss: 0.0115 - val\_accuracy: 0.9974 - val\_loss: 0.0118 - learning

Epoch 5/10

16007/16015 ————— 0s 2ms/step - accuracy: 0.9988 - loss: 0.0072

Epoch 5: ReduceLROnPlateau reducing learning rate to 9.999999747378752e-06.

16015/16015 ————— 42s 2ms/step - accuracy: 0.9988 - loss: 0.0072 - val\_accuracy: 0.9980 - val\_loss: 0.0108 - learning

Epoch 6/10

16015/16015 ————— 40s 2ms/step - accuracy: 0.9994 - loss: 0.0047 - val\_accuracy: 0.9987 - val\_loss: 0.0082 - learning

Epoch 7/10

16015/16015 ————— 41s 3ms/step - accuracy: 0.9995 - loss: 0.0042 - val\_accuracy: 0.9983 - val\_loss: 0.0097 - learning

Epoch 8/10

16015/16015 ————— 40s 2ms/step - accuracy: 0.9995 - loss: 0.0040 - val\_accuracy: 0.9984 - val\_loss: 0.0089 - learning

Epoch 9/10

16015/16015 ————— 40s 2ms/step - accuracy: 0.9995 - loss: 0.0039 - val\_accuracy: 0.9988 - val\_loss: 0.0081 - learning

Epoch 10/10

16015/16015 ————— 41s 2ms/step - accuracy: 0.9996 - loss: 0.0036 - val\_accuracy: 0.9985 - val\_loss: 0.0087 - learning

```
num_epochs = len(history.history["loss"])
```

```
fig, axarr = plt.subplots(1, 2, figsize=(24, 8))
```

```
# Loss Plot
```

```
axarr[0].set_xlabel("Number of Epochs")
```

```
axarr[0].set_ylabel("Loss")
```

```
sns.lineplot(x=range(1, num_epochs+1),
```

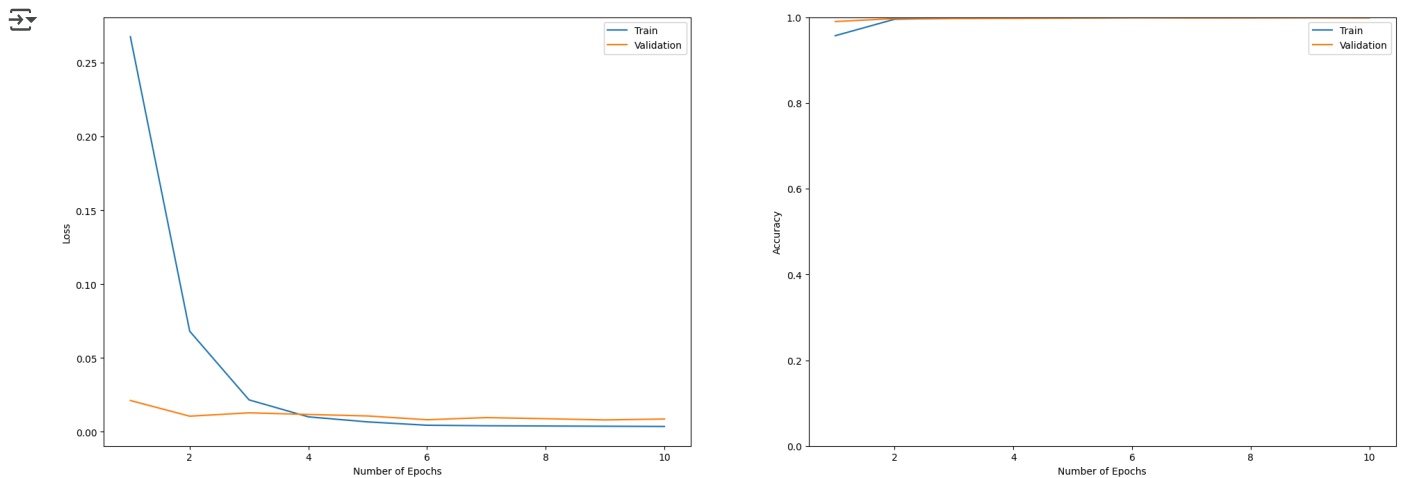
```

        y=history.history["loss"],
        label="Train",
        ax=axarr[0])
sns.lineplot(x=range(1, num_epochs+1),
             y=history.history["val_loss"],
             label="Validation",
             ax=axarr[0])

# Accuracy Plot
axarr[1].set_xlabel("Number of Epochs")
axarr[1].set_ylabel("Accuracy")
axarr[1].set_ylim(0, 1)
sns.lineplot(x=range(1, num_epochs+1),
             y=history.history["accuracy"],
             label="Train",
             ax=axarr[1])
sns.lineplot(x=range(1, num_epochs+1),
             y=history.history["val_accuracy"],
             label="Validation",
             ax=axarr[1])

plt.show()

```



```

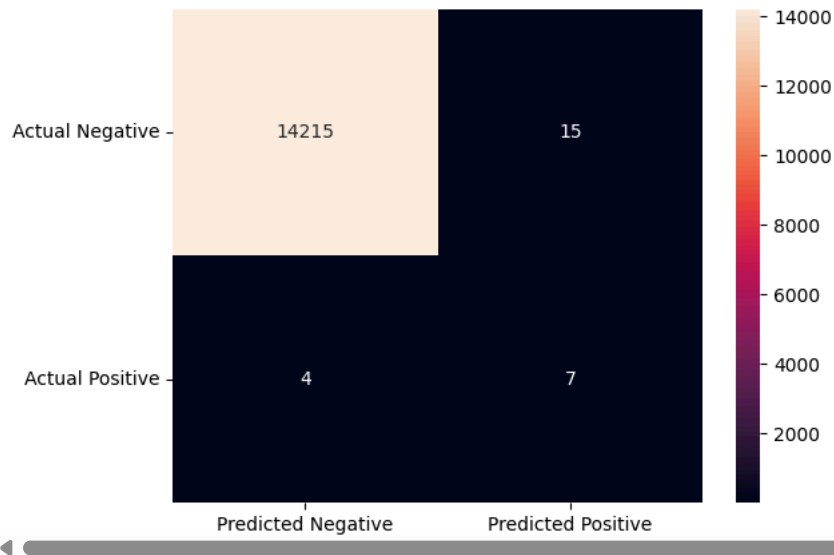
# Evaluate model on test set
test_results = model.evaluate(X_test, Y_test)
print("The model test accuracy is {}".format(test_results[1]))

# Make predictions and calculate average precision
predictions = (model.predict(X_test) > 0.5).astype("int32")
ap_score = average_precision_score(Y_test, predictions)
print("The model test average precision score is {}".format(ap_score))

# Create and plot confusion matrix
confusion = pd.DataFrame(confusion_matrix(Y_test, predictions))
confusion.columns = ["Predicted Negative", "Predicted Positive"]
confusion.index = ["Actual Negative", "Actual Positive"]
sns.heatmap(confusion, annot=True, fmt='d')
plt.yticks(rotation=0)
plt.show()

```

446/446 1s 3ms/step - accuracy: 0.9983 - loss: 0.0167  
The model test accuracy is 0.9986658096313477.  
446/446 1s 1ms/step  
The model test average precision score is 0.20276021799472016.



10

```

import numdifftools as nd

def fun(x):
    return 2*x[0]*x[1] - x[0]**2 - 2*x[1]**2 + 2*x[0]

def fun2(x):
    return 2*x[0]*x[1] + x[1]**2 + 6*x[0] + 2*x[1]

# Calculate gradient at point [2, -2]
grad2 = nd.Gradient(fun)([2, -2])
print(grad2) # Output: [-6. 12.]

import matplotlib.pyplot as plt

def gradient_descent(epoch, input):
    eta = 0.1 # Learning rate
    history = [[input[0], input[1]]] # Store optimization path

    for i in range(epoch):
        grad = nd.Gradient(fun)(input)
        input[0] = input[0] - eta * grad[0]
        input[1] = input[1] - eta * grad[1]
        history.append([input[0], input[1]])

    return history

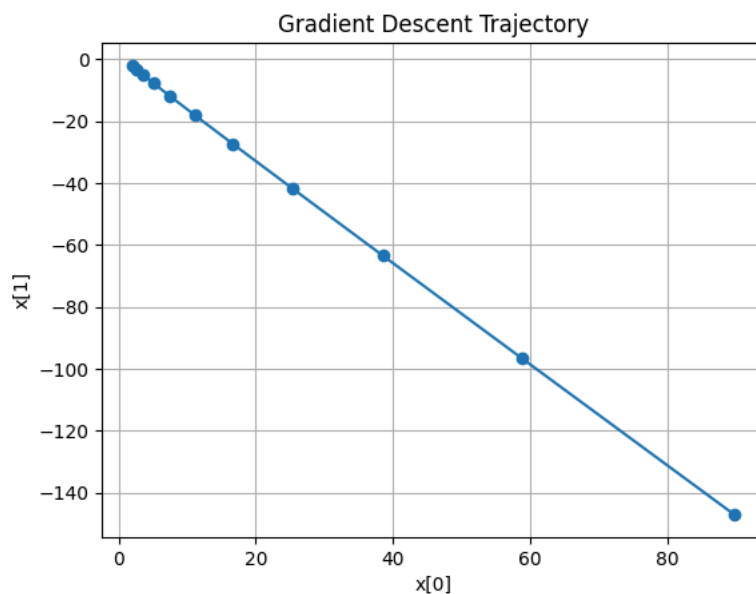
# Run gradient descent for 10 epochs starting at [2, -2]
history = gradient_descent(10, [2, -2])

# Plot optimization trajectory
x_history = [point[0] for point in history]
y_history = [point[1] for point in history]

plt.plot(x_history, y_history, 'o-')
plt.xlabel("x[0]")
plt.ylabel("x[1]")
plt.title("Gradient Descent Trajectory")
plt.grid(True)
plt.show()

```

[-6. 12.]



```

def gradient_descent(epoch, input):
    eta = 0.1 # Learning rate
    history = [[input[0], input[1]]] # Store optimization path

    for i in range(epoch):
        grad = nd.Gradient(fun)(input)
        input[0] = input[0] - eta * grad[0] # Changed + to - for gradient descent
        input[1] = input[1] - eta * grad[1] # Changed + to - for gradient descent
        history.append([input[0], input[1]])

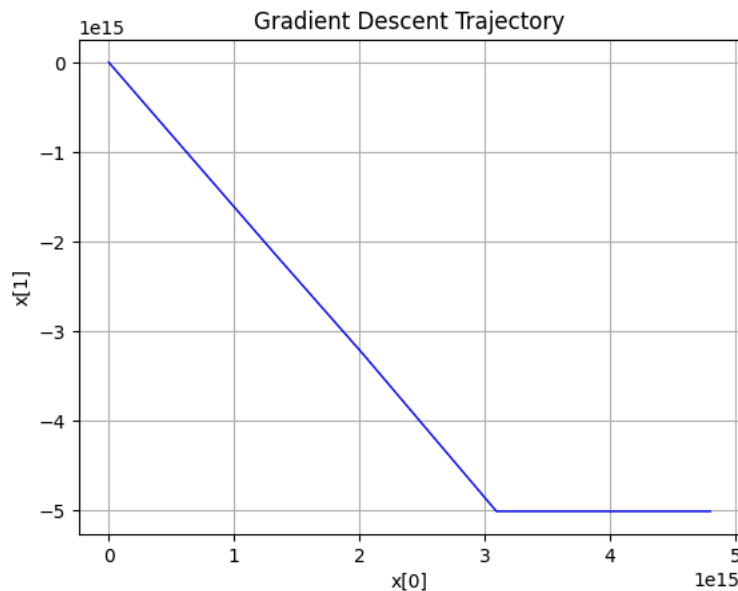
    return history

# Run gradient descent for 500 epochs starting at [2, -2]
history = gradient_descent(500, [2, -2])

```

```
# Plot optimization trajectory
x_history = [point[0] for point in history]
y_history = [point[1] for point in history]

plt.plot(x_history, y_history, 'b-', linewidth=1)
plt.xlabel("x[0]")
plt.ylabel("x[1]")
plt.title("Gradient Descent Trajectory") # Corrected title
plt.grid(True)
plt.show()
```



```
def gradient_descent(epoch, input):
    eta = 0.1 # Learning rate
    history = [[input[0], input[1]]] # Store optimization path

    for i in range(epoch):
        grad = nd.Gradient(fun2)(input) # Using fun2 for gradient calculation
        input[0] = input[0] - eta * grad[0] # Gradient descent update
        input[1] = input[1] - eta * grad[1] # Gradient descent update
        history.append([input[0], input[1]])

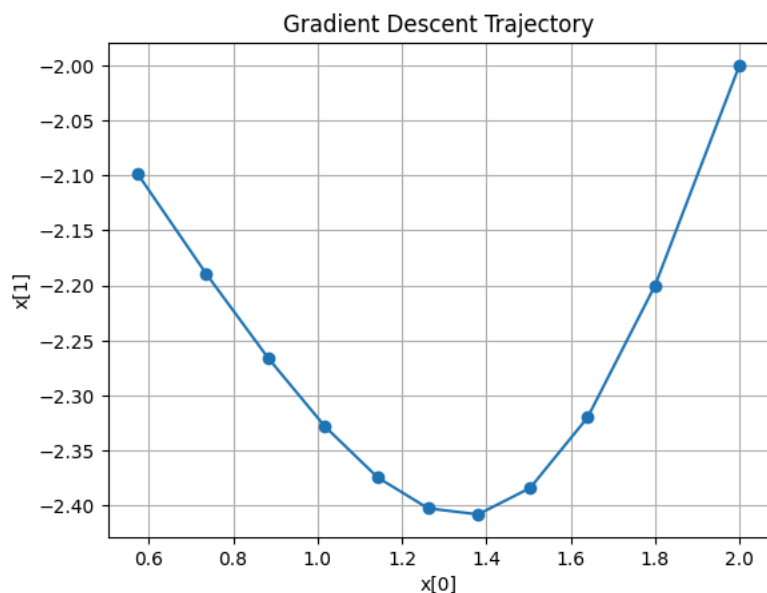
    return history

# Run gradient descent for 10 epochs starting at [2, -2]
history = gradient_descent(10, [2, -2])

# Extract x and y coordinates from history
x_history = [point[0] for point in history]
y_history = [point[1] for point in history]

# Plot optimization trajectory
plt.plot(x_history, y_history, 'o-') # Added marker and line style
plt.xlabel("x[0]")
plt.ylabel("x[1]")
plt.title("Gradient Descent Trajectory")
plt.grid(True) # Added grid for better visualization
plt.show()
```





```
def gradient_descent(epoch, input):
    eta = 0.1 # Learning rate
    history = [[input[0], input[1]] # Store optimization path

    for i in range(epoch):
        grad = nd.Gradient(fun2)(input)
        input[0] = input[0] - eta * grad[0] # Changed + to - for proper gradient descent
        input[1] = input[1] - eta * grad[1] # Changed + to - for proper gradient descent
        history.append([input[0], input[1]])

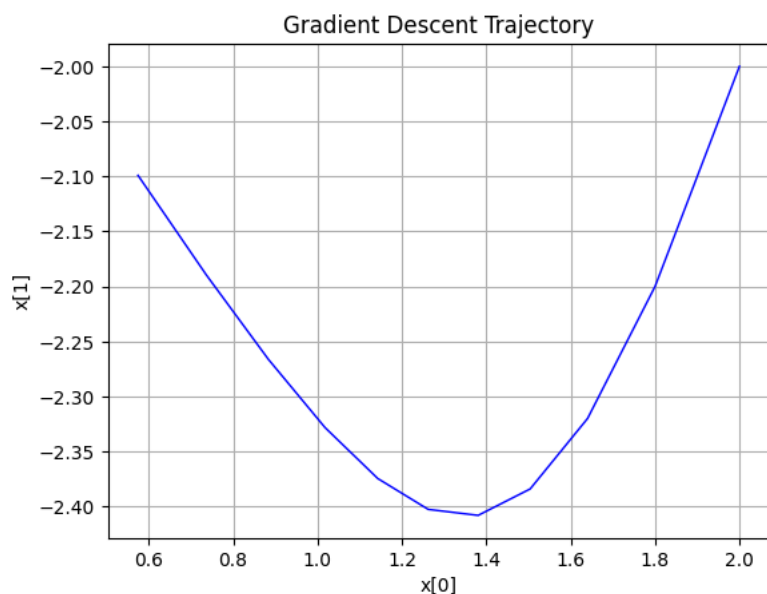
    return history

# Run gradient descent for 10 epochs starting at [2, -2]
history = gradient_descent(10, [2, -2])

# Extract x and y coordinates from history
x_history = [point[0] for point in history]
y_history = [point[1] for point in history]

# Plot optimization trajectory
plt.plot(x_history, y_history, 'b-', linewidth=1)
plt.xlabel("x[0]")
plt.ylabel("x[1]")
plt.title("Gradient Descent Trajectory") # Corrected title
plt.grid(True)
plt.show()

# Print final position
print("Final position:", history[-1])
```



Final position: [np.float64(0.5746027519999966), np.float64(-2.0990116863999977)]

```

import numdifftools as nd
import numpy as np
import matplotlib.pyplot as plt

def fun(x):
    return x[0]*x[1] + 4*x[1] - 3*x[0]**2 - x[1]**2

def newton_method(fun, x0, max_iter=100, tol=1e-6):
    x = np.array(x0, dtype=float)
    x_history = [x.copy()]

    for i in range(max_iter):
        grad = nd.Gradient(fun)(x)
        hess = nd.Hessian(fun)(x)

        try:
            hess_inv = np.linalg.inv(hess)
        except np.linalg.LinAlgError:
            print(f"Hessian is singular at iteration {i}")
            return x, x_history

        x_new = x - np.dot(hess_inv, grad)
        x_history.append(x_new.copy())

        if np.linalg.norm(x_new - x) < tol:
            return x_new, x_history

    x = x_new

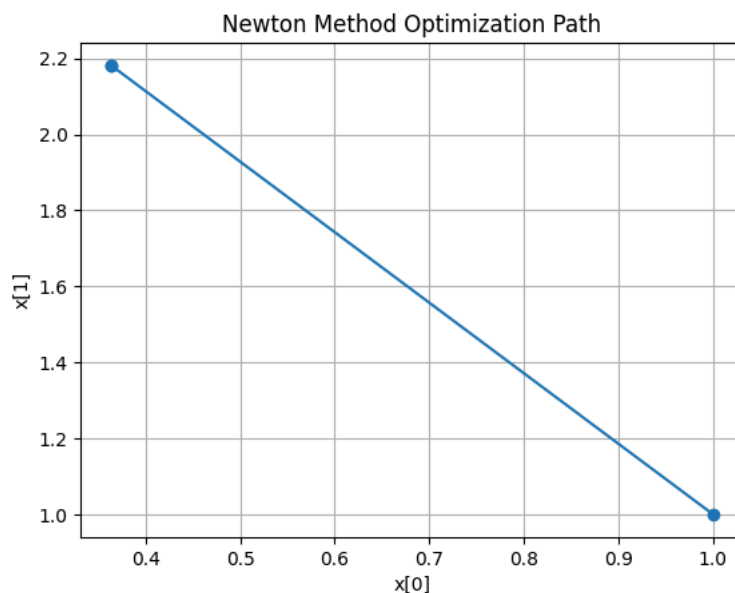
    print("Maximum iterations reached without convergence")
    return x, x_history

# Initial guess
x0 = [1, 1]
optimal_x, x_history = newton_method(fun, x0)
print(f"Optimal point: {optimal_x}")

# Plot optimization path if desired
x_history = np.array(x_history)
plt.plot(x_history[:, 0], x_history[:, 1], 'o-')
plt.xlabel('x[0]')
plt.ylabel('x[1]')
plt.title('Newton Method Optimization Path')
plt.grid(True)
plt.show()

```

↗ Optimal point: [0.36363636 2.18181818]



## Conclusion

The Newton's method efficiently optimizes the function  $xy + 4y - 3x^2 - y^2$  by iteratively updating the variables using the inverse Hessian and gradient. The visualization aids in understanding the convergence trajectory from the initial guess to the optimal point.