COMPILER CONSTRUCTION

LAB MANUAL

AMITY SCHOOL OF ENGINEERING & TECHNOLOGY

AUUP, NOIDA

B.TECH – COMPUTER SCIENCE AND ENGINEERING

SEMESTER – 6

COURSE CODE – CSE304

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

AMITY UNIVERSITY, NOIDA, UTTARPRADESH

Name: Ojaswin Aggarwal

Enrollment No: A2305222103

Section: 6CSE2X

INDEX

Experiment 1

Objective:

To implement various activation functions commonly used in neural networks using Python.

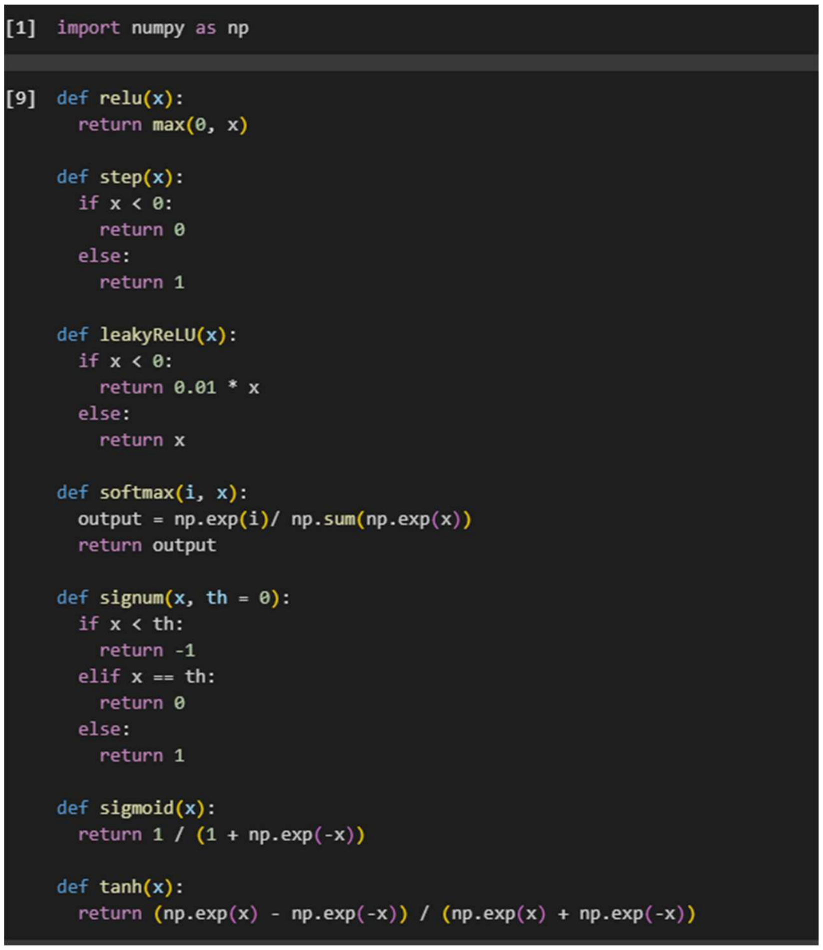
Background:

Activation functions introduce non-linearity in neural networks, allowing them to capture complex patterns. Some commonly used activation functions include:

- **Binary Step:** Returns 0 or 1 based on a threshold, making it useful for binary classification.
- **Linear:** Outputs a scaled version of the input, but lacks non-linearity.
- **Sigmoid:** Maps values between 0 and 1, commonly used for probability estimation.
- **Tanh:** Normalizes values between -1 and 1, offering a centered output.
- **ReLU (Rectified Linear Unit):** Sets negative inputs to zero, improving gradient flow.
- **Leaky ReLU:** Allows small negative values to prevent neuron inactivation.
- **Softmax:** Converts logits into probabilities for multi-class classification.

Each activation function plays a significant role in optimizing neural networks.

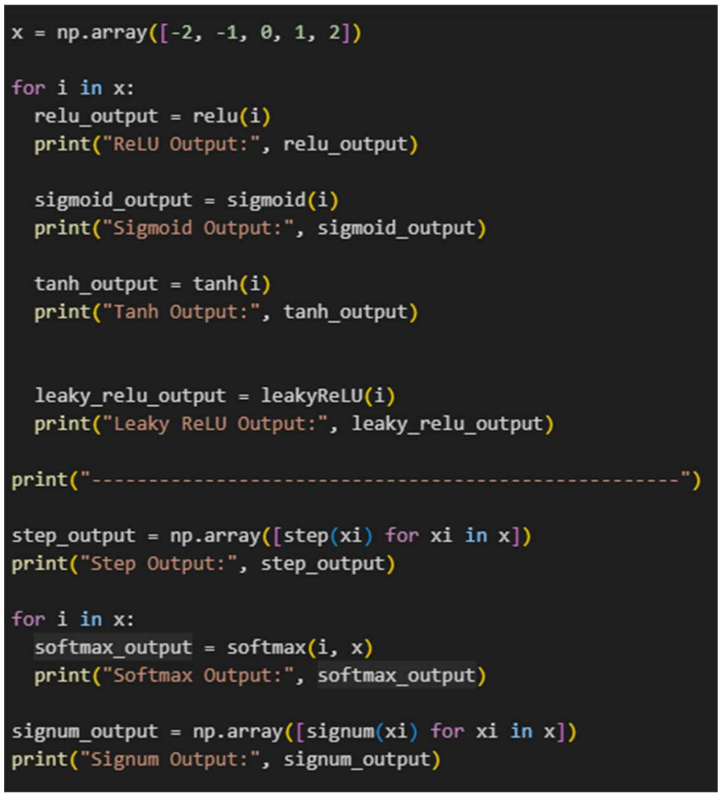
Code -



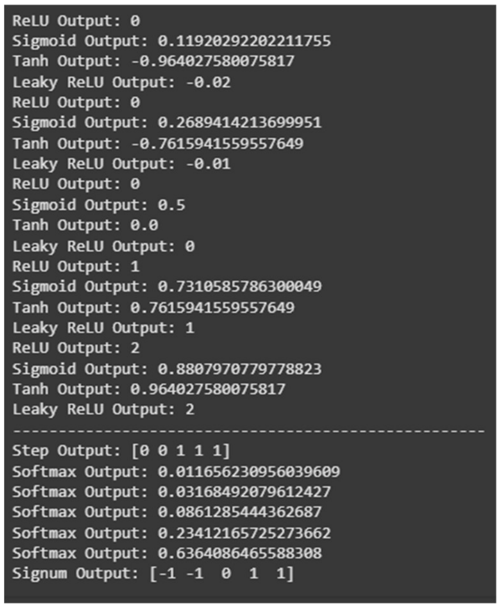
```
[1]  import numpy as np
```

```
[9]  def relu(x):
       return max(0, x)

     def step(x):
       if x < 0:
         return 0
       else:
         return 1

     def leakyReLU(x):
       if x < 0:
         return 0.01 * x
       else:
         return x

     def softmax(i, x):
       output = np.exp(i)/ np.sum(np.exp(x))
       return output

     def signum(x, th = 0):
       if x < th:
         return -1
       elif x == th:
         return 0
       else:
         return 1

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

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

Output –



```
x = np.array([-2, -1, 0, 1, 2])

for i in x:
  relu_output = relu(i)
  print("ReLU Output:", relu_output)

  sigmoid_output = sigmoid(i)
  print("Sigmoid Output:", sigmoid_output)

  tanh_output = tanh(i)
  print("Tanh Output:", tanh_output)


  leaky_relu_output = leakyReLU(i)
  print("Leaky ReLU Output:", leaky_relu_output)

print("---------------------------------------------------")

step_output = np.array([step(xi) for xi in x])
print("Step Output:", step_output)

for i in x:
  softmax_output = softmax(i, x)
  print("Softmax Output:", softmax_output)

signum_output = np.array([signum(xi) for xi in x])
print("Signum Output:", signum_output)
```



```
ReLU Output: 0
Sigmoid Output: 0.11920292202211755
Tanh Output: -0.964027580075817
Leaky ReLU Output: -0.02
ReLU Output: 0
Sigmoid Output: 0.2689414213699951
Tanh Output: -0.7615941559557649
Leaky ReLU Output: -0.01
ReLU Output: 0
Sigmoid Output: 0.5
Tanh Output: 0.0
Leaky ReLU Output: 0
ReLU Output: 1
Sigmoid Output: 0.7310585786300049
Tanh Output: 0.7615941559557649
Leaky ReLU Output: 1
ReLU Output: 2
Sigmoid Output: 0.8807970779778823
Tanh Output: 0.964027580075817
Leaky ReLU Output: 2
---------------------------------------------------
Step Output: [0 0 1 1 1]
Softmax Output: 0.011656230956039609
Softmax Output: 0.031684920796124276
Softmax Output: 0.0861285444362687
Softmax Output: 0.23412165725273662
Softmax Output: 0.6364086465588308
Signum Output: [-1 -1  0  1  1]
```

Experiment 2

Objective:

a) To classify given data points using a perceptron with a step activation function.
b) To compute the output using the same inputs, weights, and bias but with a sigmoid activation function.

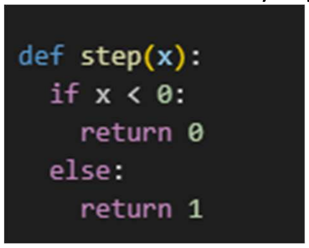
Background:

A perceptron is a fundamental unit in neural networks used for binary classification tasks. It consists of:

- **Inputs:** Feature values.
- **Weights:** Associated parameters that influence decision-making.
- **Bias:** A constant that shifts the activation threshold.
- **Activation Function:** Determines the final output.
-

(a) Step Activation Function:

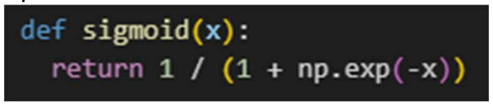
- Also called a threshold function.
- Outputs **1** if the weighted sum exceeds a threshold, otherwise **0**.
- Works well for linearly separable problems but struggles with complex data.



```python
def step(x):
  if x < 0:
    return 0
  else:
    return 1
```

(b) Sigmoid Activation Function:

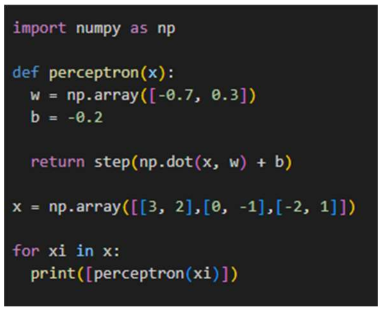
- Outputs values between **0 and 1**, making it useful for probability-based classification.
- Unlike the step function, it is **continuous and differentiable**, allowing for gradient-based optimization.



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

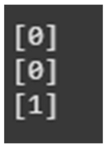
Code –

a) **Step Activation function**



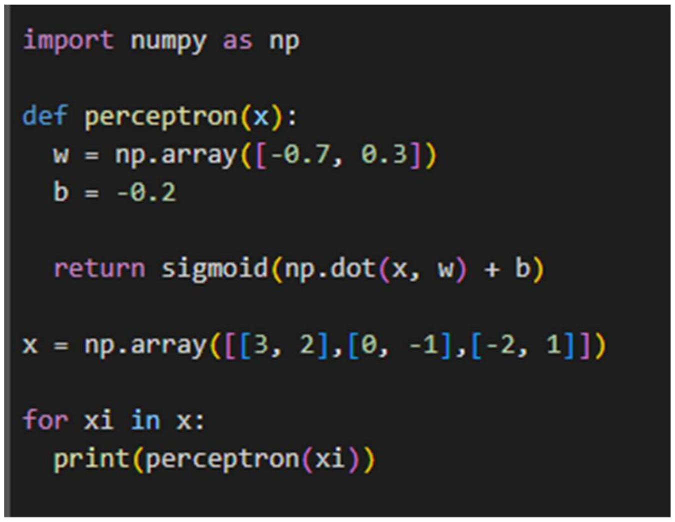
```python
import numpy as np

def perceptron(x):
  w = np.array([-0.7, 0.3])
  b = -0.2

  return step(np.dot(x, w) + b)

x = np.array([[3, 2],[0, -1],[-2, 1]])

for xi in x:
  print([perceptron(xi)])
```

Output –



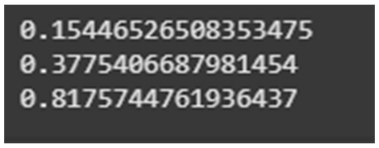
```
[0]
[0]
[1]
```

b) Sigmoid Activation function



```python
import numpy as np

def perceptron(x):
  w = np.array([-0.7, 0.3])
  b = -0.2

  return sigmoid(np.dot(x, w) + b)

x = np.array([[3, 2],[0, -1],[-2, 1]])

for xi in x:
  print(perceptron(xi))
```

Output –



```
0.15446526508353475
0.3775406687981454
0.8175744761936437
```

Experiment 3

Objective:

To understand vectorization using NumPy by comparing non-vectorized and vectorized implementations of forward propagation in a neural network.

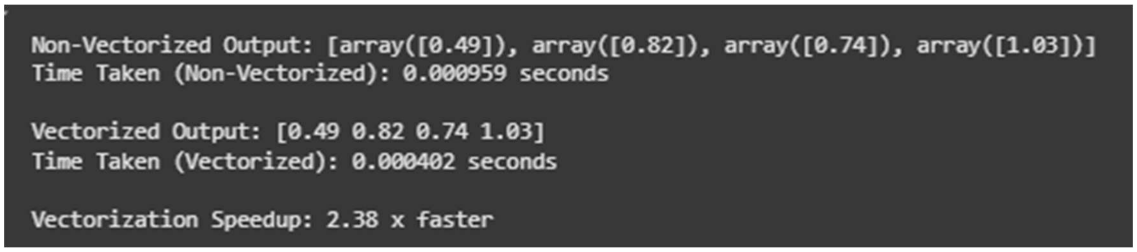
Background:

Vectorization is a technique that optimizes numerical computations by applying operations to entire arrays instead of iterating through individual elements. This significantly enhances performance by reducing execution time and memory overhead.

NumPy, a widely used library for numerical computing, supports vectorized operations that improve efficiency in machine learning and deep learning tasks.
Key Concepts:

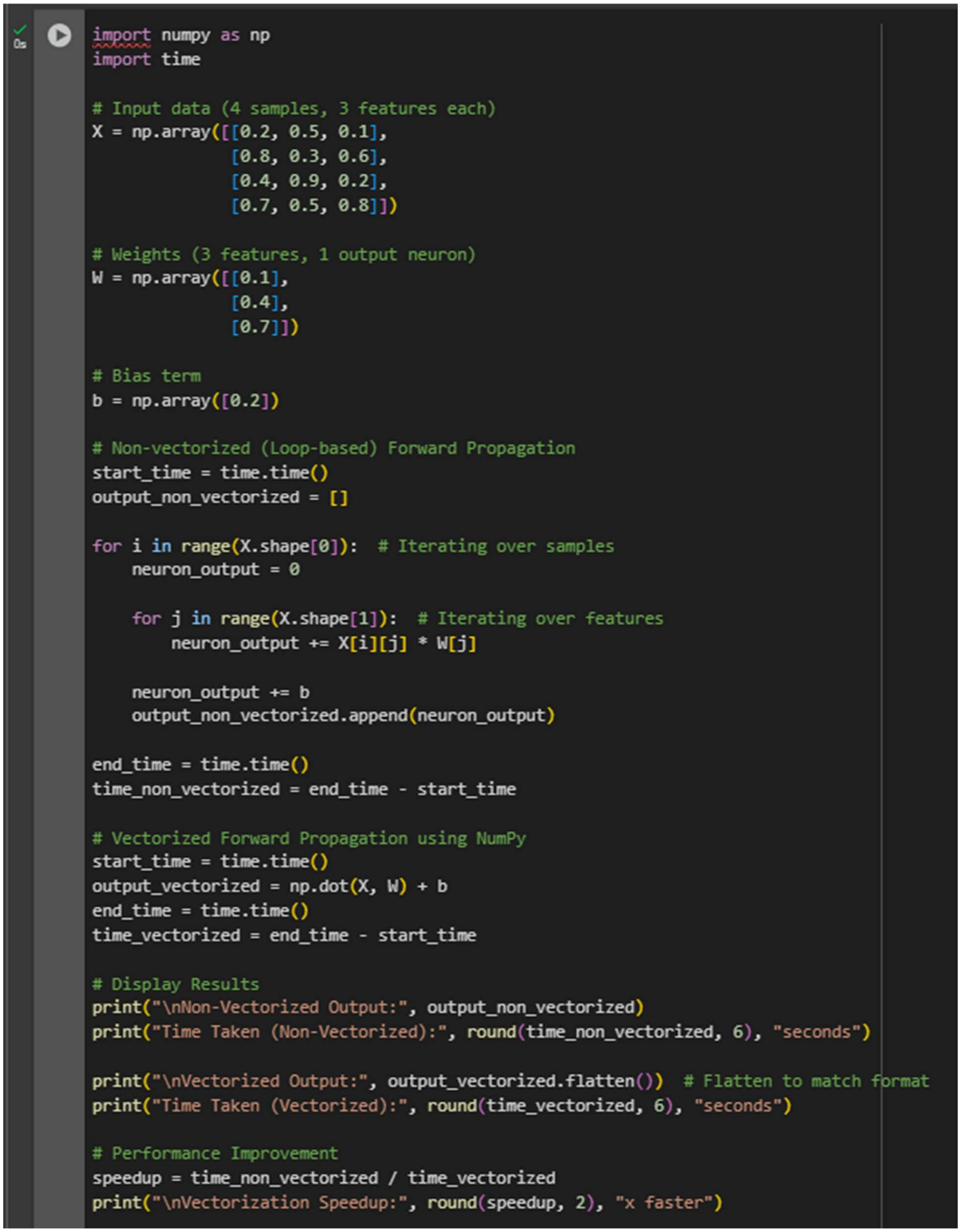
- Non-vectorized approach: Uses explicit loops for element-wise computations, which can be slow for large datasets.
- Vectorized approach: Utilizes NumPy’s dot product (np.dot) to perform computations in a single step, significantly reducing execution time.
- Use in ML/DL: Speeding up matrix operations is critical in deep learning, where large-scale computations are performed during training and inference.

Output –



```
Non-Vectorized Output: [array([0.49]), array([0.82]), array([0.74]), array([1.03])]
Time Taken (Non-Vectorized): 0.000959 seconds

Vectorized Output: [0.49 0.82 0.74 1.03]
Time Taken (Vectorized): 0.000402 seconds

Vectorization Speedup: 2.38 x faster
```

Code –



```
import numpy as np
import time

# Input data (4 samples, 3 features each)
X = np.array([[0.2, 0.5, 0.1],
              [0.8, 0.3, 0.6],
              [0.4, 0.9, 0.2],
              [0.7, 0.5, 0.8]])

# Weights (3 features, 1 output neuron)
W = np.array([[0.1],
              [0.4],
              [0.7]])

# Bias term
b = np.array([0.2])

# Non-vectorized (Loop-based) Forward Propagation
start_time = time.time()
output_non_vectorized = []

for i in range(X.shape[0]):  # Iterating over samples
    neuron_output = 0

    for j in range(X.shape[1]):  # Iterating over features
        neuron_output += X[i][j] * W[j]

    neuron_output += b
    output_non_vectorized.append(neuron_output)

end_time = time.time()
time_non_vectorized = end_time - start_time

# Vectorized Forward Propagation using NumPy
start_time = time.time()
output_vectorized = np.dot(X, W) + b
end_time = time.time()
time_vectorized = end_time - start_time

# Display Results
print("\nNon-Vectorized Output:", output_non_vectorized)
print("Time Taken (Non-Vectorized):", round(time_non_vectorized, 6), "seconds")

print("\nVectorized Output:", output_vectorized.flatten())  # Flatten to match format
print("Time Taken (Vectorized):", round(time_vectorized, 6), "seconds")

# Performance Improvement
speedup = time_non_vectorized / time_vectorized
print("\nVectorization Speedup:", round(speedup, 2), "x faster")
```

Experiment 4

Aim:

To implement AND, OR, NOR, NAND, XOR, and XNOR logic gates using an Artificial Neural Network (ANN).

Theory:

Logic gates such as AND, OR, NOR, NAND, XOR, and XNOR can be implemented using perceptrons in an ANN. A perceptron computes the weighted sum of inputs and applies an activation function (unit step function), given by:

y=f(W·X+b)y = f(W \cdot X + b)y=f(W·X+b)

where:

W represents the weight vector,

X denotes the input vector, and

b is the bias term.

Implementation of Logic Gates:

AND & OR: Implemented using single-layer perceptrons with appropriate weights and bias values.

NAND & NOR: Derived by applying a NOT operation to the output of AND and OR perceptrons, respectively.

XOR & XNOR: Require a multi-layer perceptron (MLP) since they are not linearly separable. This is achieved by combining AND, OR, and NOT operations.

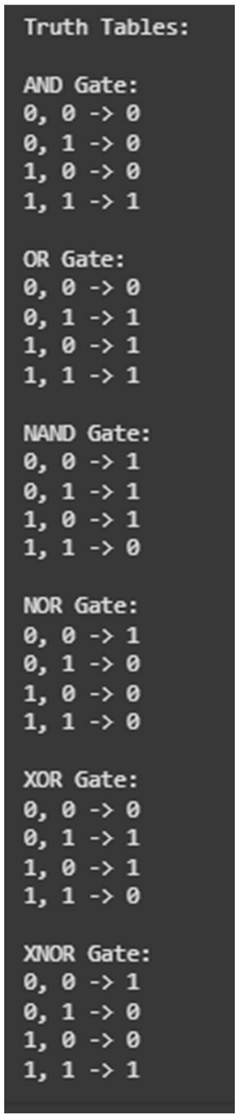
This experiment demonstrates how single-layer perceptrons can implement basic logic functions, while multi-layer networks are necessary for more complex operations like XOR and XNOR.

Code –

```
import numpy as np

# Activation function (Unit Step Function)
def activation(x):
    return 1 if x >= 0 else 0

# Perceptron Model
def perceptron_logic_gate(inputs, weights, bias):
    weighted_sum = np.dot(inputs, weights) + bias
    return activation(weighted_sum)

# Training weights and bias for basic gates
logic_gates = {
    "AND": (np.array([1, 1]), -1.5),
    "OR": (np.array([1, 1]), -0.5),
    "NAND": (np.array([-1, -1]), 1.5),
    "NOR": (np.array([-1, -1]), 0.5),
}
```

```
def xor_gate(x1, x2):
    """ XOR using AND, OR, and NAND """
    and_out = perceptron_logic_gate([x1, x2], logic_gates["AND"][0], logic_gates["AND"][1])
    or_out = perceptron_logic_gate([x1, x2], logic_gates["OR"][0], logic_gates["OR"][1])
    nand_out = perceptron_logic_gate([x1, x2], logic_gates["NAND"][0], logic_gates["NAND"][1])
    return perceptron_logic_gate([nand_out, or_out], logic_gates["AND"][0], logic_gates["AND"][1])

def xnor_gate(x1, x2):
    """ XNOR is NOT XOR """
    return 1 - xor_gate(x1, x2)

# Testing all logic gates
inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]
print("Truth Tables:")
for gate, (weights, bias) in logic_gates.items():
    print(f"\n{gate} Gate:")
    for x1, x2 in inputs:
        output = perceptron_logic_gate([x1, x2], weights, bias)
        print(f"{x1}, {x2} -> {output}")

print("\nXOR Gate:")
for x1, x2 in inputs:
    print(f"{x1}, {x2} -> {xor_gate(x1, x2)}")

print("\nXNOR Gate:")
for x1, x2 in inputs:
    print(f"{x1}, {x2} -> {xnor_gate(x1, x2)}")
```

Output –



```
Truth Tables:

AND Gate:
0, 0 -> 0
0, 1 -> 0
1, 0 -> 0
1, 1 -> 1

OR Gate:
0, 0 -> 0
0, 1 -> 1
1, 0 -> 1
1, 1 -> 1

NAND Gate:
0, 0 -> 1
0, 1 -> 1
1, 0 -> 1
1, 1 -> 0

NOR Gate:
0, 0 -> 1
0, 1 -> 0
1, 0 -> 0
1, 1 -> 0

XOR Gate:
0, 0 -> 0
0, 1 -> 1
1, 0 -> 1
1, 1 -> 0

XNOR Gate:
0, 0 -> 1
0, 1 -> 0
1, 0 -> 0
1, 1 -> 1
```

Experiment 5

Aim:
To implement a linear regression model with one variable for housing price prediction.

Theory:
Linear regression is a statistical method used to model the relationship between a dependent variable (housing price) and an independent variable (house size in square feet). The model assumes that this relationship can be approximated by a straight line. The goal is to determine the optimal parameters:

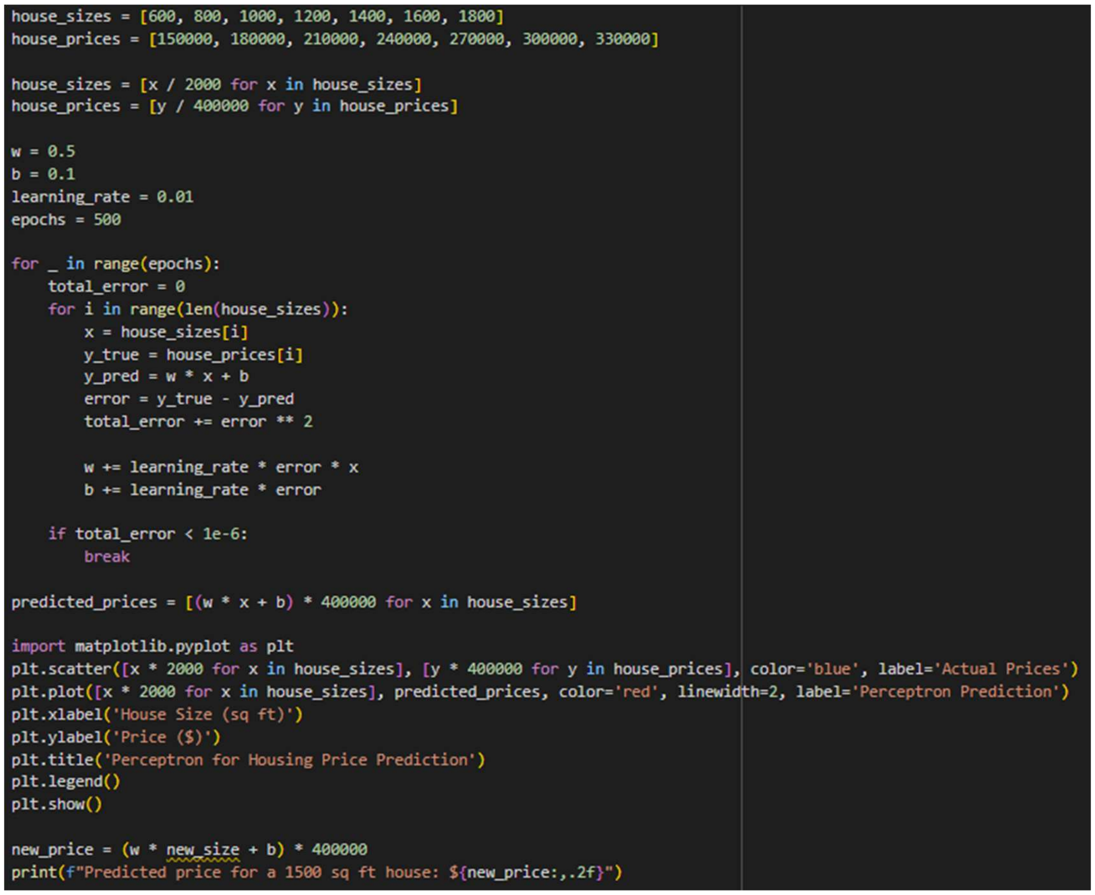
- Weight (w): Represents the slope, indicating how price changes with size.
- Bias (b): Represents the intercept, the predicted price when the house size is zero.

The linear regression equation is given by:

$$\text{Price}=(w\cdot \text{size})+b$$

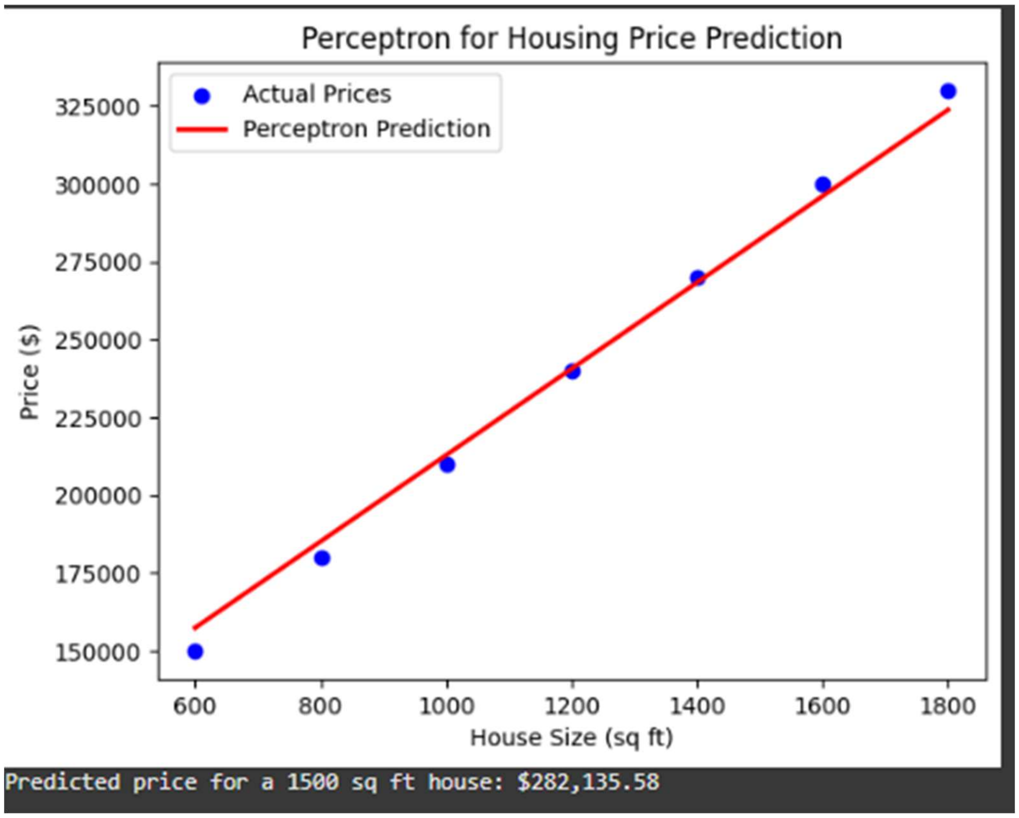
This experiment demonstrates how linear regression can be used for housing price prediction by fitting a line to data points and minimizing errors.

Code –



```python
house_sizes = [600, 800, 1000, 1200, 1400, 1600, 1800]
house_prices = [150000, 180000, 210000, 240000, 270000, 300000, 330000]

house_sizes = [x / 2000 for x in house_sizes]
house_prices = [y / 400000 for y in house_prices]

w = 0.5
b = 0.1
learning_rate = 0.01
epochs = 500

for _ in range(epochs):
    total_error = 0
    for i in range(len(house_sizes)):
        x = house_sizes[i]
        y_true = house_prices[i]
        y_pred = w * x + b
        error = y_true - y_pred
        total_error += error ** 2

        w += learning_rate * error * x
        b += learning_rate * error

    if total_error < 1e-6:
        break

predicted_prices = [(w * x + b) * 400000 for x in house_sizes]

import matplotlib.pyplot as plt
plt.scatter([x * 2000 for x in house_sizes], [y * 400000 for y in house_prices], color='blue', label='Actual Prices')
plt.plot([x * 2000 for x in house_sizes], predicted_prices, color='red', linewidth=2, label='Perceptron Prediction')
plt.xlabel('House Size (sq ft)')
plt.ylabel('Price ($)')
plt.title('Perceptron for Housing Price Prediction')
plt.legend()
plt.show()

new_price = (w * new_size + b) * 400000
print(f"Predicted price for a 1500 sq ft house: ${new_price:,.2f}")
```

Output –



```
Predicted price for a 1500 sq ft house: $282,135.58
```

Experiment 5

Aim:

To implement the linear regression model with one variable for housing price prediction.

Theory:

Linear regression is a statistical method used to model the relationship between a dependent variable (housing price) and an independent variable (house size in square feet). The model assumes this relationship can be approximated by a straight line. The objective is to determine the optimal parameters—weight (slope) and bias (intercept)—that best fit the given data.

The price of a house is predicted using the equation:

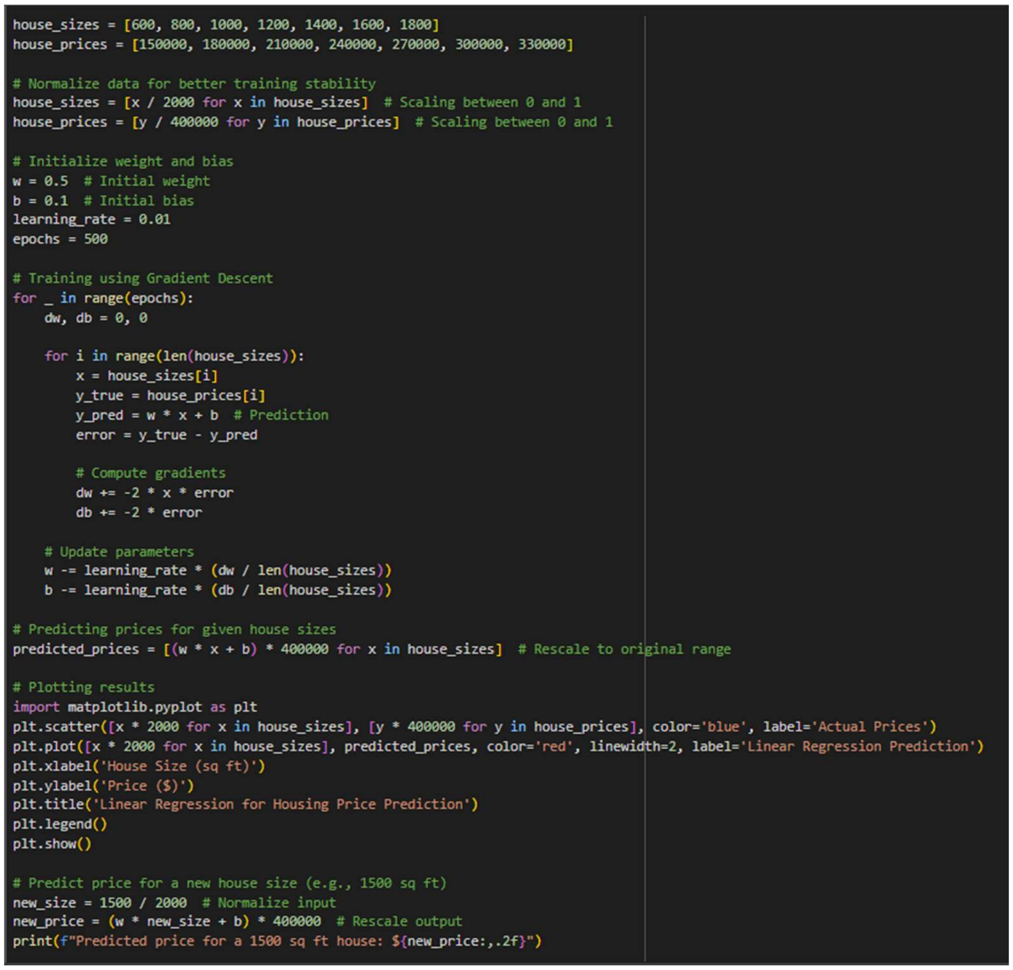
$$\text{Price}=(w\times \text{size})+b$$

where:

- w (weight/slope): Represents the rate at which the price changes as the house size increases.
- b (bias/intercept): Represents the predicted price when the house size is zero.

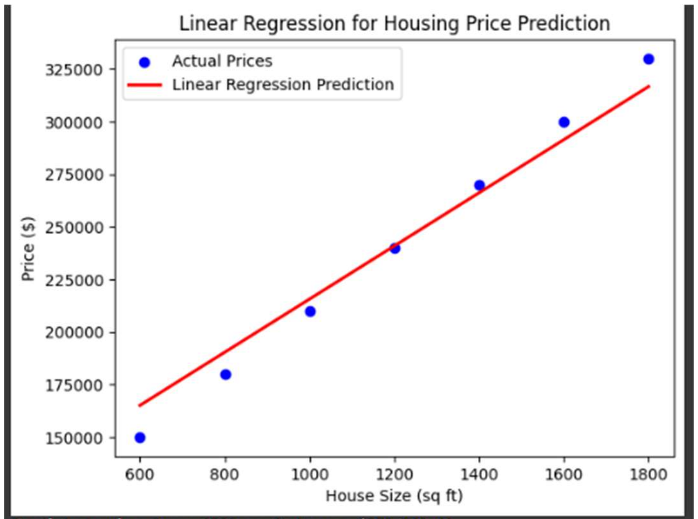
This model is widely used in predictive analytics and real estate valuation, allowing for data-driven price estimations based on property size.

Code –



```python
house_sizes = [600, 800, 1000, 1200, 1400, 1600, 1800]
house_prices = [150000, 180000, 210000, 240000, 270000, 300000, 330000]

# Normalize data for better training stability
house_sizes = [x / 2000 for x in house_sizes]  # Scaling between 0 and 1
house_prices = [y / 400000 for y in house_prices]  # Scaling between 0 and 1

# Initialize weight and bias
w = 0.5  # Initial weight
b = 0.1  # Initial bias
learning_rate = 0.01
epochs = 500

# Training using Gradient Descent
for _ in range(epochs):
    dw, db = 0, 0

    for i in range(len(house_sizes)):
        x = house_sizes[i]
        y_true = house_prices[i]
        y_pred = w * x + b  # Prediction
        error = y_true - y_pred

        # Compute gradients
        dw += -2 * x * error
        db += -2 * error

    # Update parameters
    w -= learning_rate * (dw / len(house_sizes))
    b -= learning_rate * (db / len(house_sizes))

# Predicting prices for given house sizes
predicted_prices = [(w * x + b) * 400000 for x in house_sizes]  # Rescale to original range

# Plotting results
import matplotlib.pyplot as plt
plt.scatter([x * 2000 for x in house_sizes], [y * 400000 for y in house_prices], color='blue', label='Actual Prices')
plt.plot([x * 2000 for x in house_sizes], predicted_prices, color='red', linewidth=2, label='Linear Regression Prediction')
plt.xlabel('House Size (sq ft)')
plt.ylabel('Price ($)')
plt.title('Linear Regression for Housing Price Prediction')
plt.legend()
plt.show()

# Predict price for a new house size (e.g., 1500 sq ft)
new_size = 1500 / 2000  # Normalize input
new_price = (w * new_size + b) * 400000  # Rescale output
print(f"Predicted price for a 1500 sq ft house: ${new_price:,.2f}")
```

Output –



```
Predicted price for a 1500 sq ft house: $278,749.61
```

Ojaswin Aggarwal A2305222103 6CSE2X

Experiment 6

Aim:

To automate the optimization of w and b in the housing price prediction model using Gradient Descent.

Theory:

This experiment focuses on optimizing the parameters w (weight) and b (bias) in a linear regression model using Gradient Descent, an iterative optimization algorithm. The objective is to minimize the cost function (Mean Squared Error), which quantifies the difference between predicted and actual values.

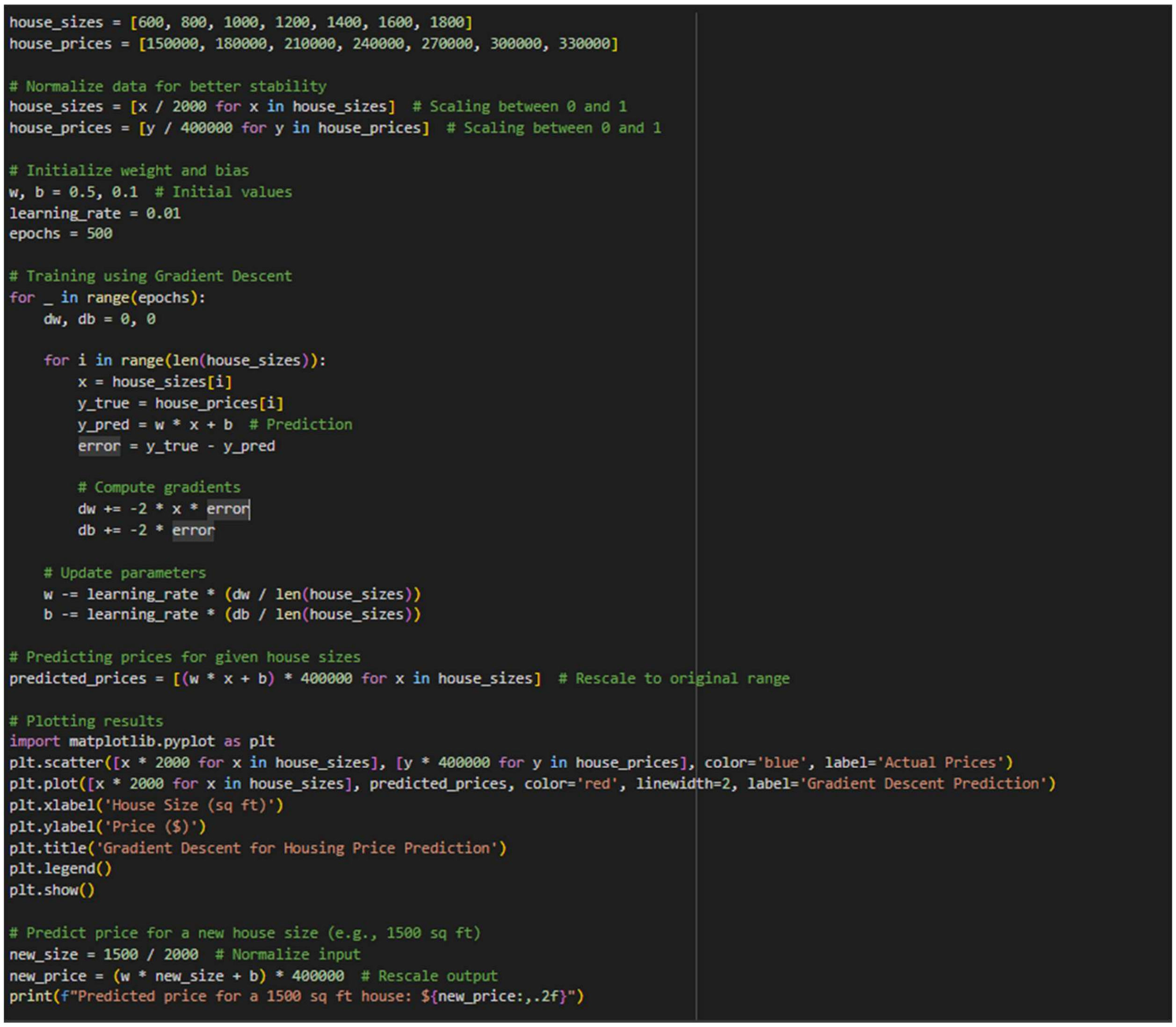
Gradient Descent Algorithm:

The optimization process follows these steps:

1. Initialization: Set initial values for w and b.
2. Compute Gradients: Calculate how much the cost function changes concerning w and b.
3. Update Parameters: Adjust w and b using the gradients and a learning rate (α).
4. Repeat: Continue iterating until the cost function converges (i.e., minimal change in subsequent iterations).

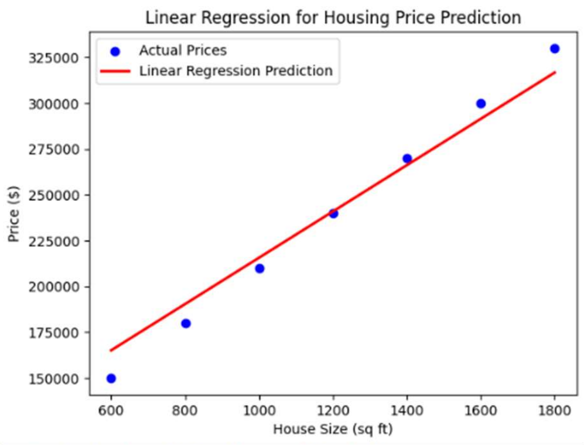
By following this approach, Gradient Descent efficiently finds the optimal values of w and b, ensuring the model best fits the given housing price data.

Code –



```
house_sizes = [600, 800, 1000, 1200, 1400, 1600, 1800]
house_prices = [150000, 180000, 210000, 240000, 270000, 300000, 330000]

# Normalize data for better stability
house_sizes = [x / 2000 for x in house_sizes]  # Scaling between 0 and 1
house_prices = [y / 400000 for y in house_prices]  # Scaling between 0 and 1

# Initialize weight and bias
w, b = 0.5, 0.1  # Initial values
learning_rate = 0.01
epochs = 500

# Training using Gradient Descent
for _ in range(epochs):
    dw, db = 0, 0

    for i in range(len(house_sizes)):
        x = house_sizes[i]
        y_true = house_prices[i]
        y_pred = w * x + b  # Prediction
        error = y_true - y_pred

        # Compute gradients
        dw += -2 * x * error
        db += -2 * error

    # Update parameters
    w -= learning_rate * (dw / len(house_sizes))
    b -= learning_rate * (db / len(house_sizes))

# Predicting prices for given house sizes
predicted_prices = [(w * x + b) * 400000 for x in house_sizes]  # Rescale to original range

# Plotting results
import matplotlib.pyplot as plt
plt.scatter([x * 2000 for x in house_sizes], [y * 400000 for y in house_prices], color='blue', label='Actual Prices')
plt.plot([x * 2000 for x in house_sizes], predicted_prices, color='red', linewidth=2, label='Gradient Descent Prediction')
plt.xlabel('House Size (sq ft)')
plt.ylabel('Price ($)')
plt.title('Gradient Descent for Housing Price Prediction')
plt.legend()
plt.show()

# Predict price for a new house size (e.g., 1500 sq ft)
new_size = 1500 / 2000  # Normalize input
new_price = (w * new_size + b) * 400000  # Rescale output
print(f"Predicted price for a 1500 sq ft house: ${new_price:,.2f}")
```

Output –



```
Predicted price for a 1500 sq ft house: $278,749.61
```

Experiment 7

Aim:

To implement and update the weights for the given neural network, considering a target output of 0.5 and a learning rate of 1.

Theory:

In this experiment, a Neural Network is implemented to solve a simple binary classification problem, specifically learning the XOR function. The network consists of three layers:

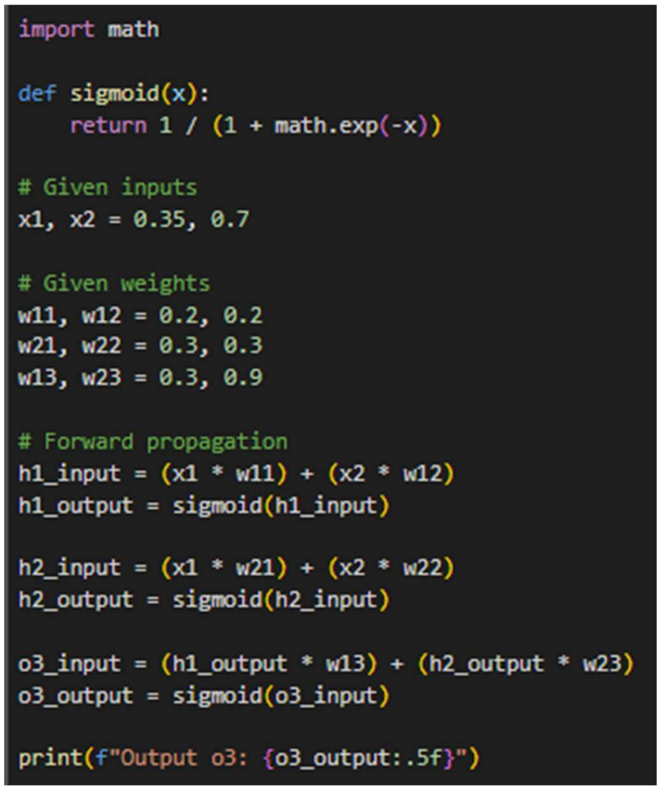
- Input layer
- Hidden layer
- Output layer

Key Components:

1. Feedforward Process:
    - Input data is passed through the network.
    - Each layer's output is computed using weights, biases, and the sigmoid activation function.
2. Backpropagation:
    - The error between the predicted and target output is calculated.
    - The error is propagated backward through the network.
    - Weights and biases are updated using the gradient descent method.
3. Training:
    - The network is trained over multiple epochs, where weights are continually updated.
    - The goal is to minimize the loss function, ensuring the output converges towards 0.5.

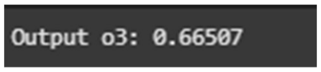
By using gradient descent with a learning rate of 1, the network iteratively adjusts the weights to approximate the XOR function effectively.

Code –



```
import math

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

# Given inputs
x1, x2 = 0.35, 0.7

# Given weights
w11, w12 = 0.2, 0.2
w21, w22 = 0.3, 0.3
w13, w23 = 0.3, 0.9

# Forward propagation
h1_input = (x1 * w11) + (x2 * w12)
h1_output = sigmoid(h1_input)

h2_input = (x1 * w21) + (x2 * w22)
h2_output = sigmoid(h2_input)

o3_input = (h1_output * w13) + (h2_output * w23)
o3_output = sigmoid(o3_input)

print(f"Output o3: {o3_output:.5f}")
```

Output –



```
Output o3: 0.66507
```

Experiment 8

Aim:

To implement Artificial Neural Networks (ANNs) using TensorFlow on:
a. Power Plant Dataset (Regression)
b. Churn Dataset (Classification)

Theory:

This experiment involves implementing ANNs using TensorFlow and Keras to solve two different machine learning tasks:

1. Power Plant Dataset (Regression):
   - The goal is to predict the net electrical energy output of a power plant based on environmental and operational factors such as temperature, pressure, and humidity.
   - A feedforward neural network is used with multiple hidden layers and the ReLU activation function to capture complex patterns in the data.
   - The Mean Squared Error (MSE) loss function is applied, which measures the average squared difference between actual and predicted values.
   - The model is trained using the Adam optimizer, and evaluation metrics such as Mean Squared Logarithmic Error (MSLE) are used to assess performance.

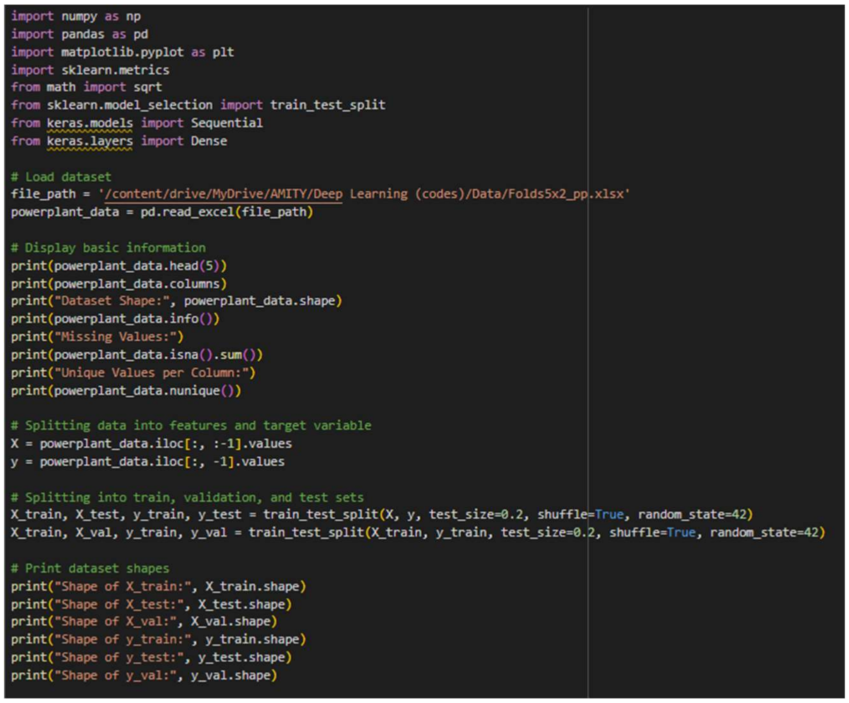
2. Churn Dataset (Classification):
   - The objective is to predict whether a customer will churn (leave) or stay based on features like age, balance, and credit score.
   - A neural network is designed with a sigmoid activation function at the output layer to perform binary classification.
   - The Binary Cross-Entropy loss function is used to measure classification error.
   - Model performance is evaluated using:
   - Accuracy (overall correctness of predictions)
   - Confusion Matrix (to analyze false positives and false negatives)
   - ROC Curve (Receiver Operating Characteristic) to assess classification ability.

Data Preprocessing:
- Both implementations require:
- Normalization of numerical features for better convergence.
- Encoding categorical variables (if any).
- Splitting data into training and testing sets for evaluation.

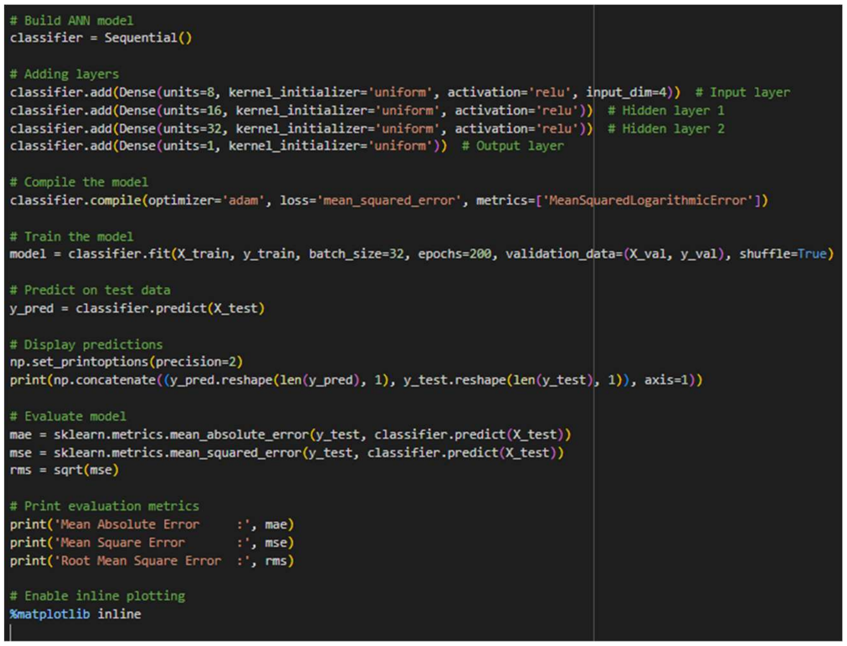
By following this approach, the ANN models can effectively learn patterns from the datasets and make accurate predictions.

Code –

a) Power plant dataset

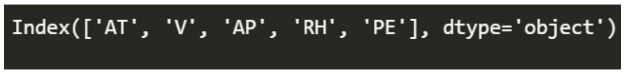


```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn.metrics
from math import sqrt
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense

# Load dataset
file_path = '/content/drive/MyDrive/AMITY/Deep Learning (codes)/Data/Folds5x2_pp.xlsx'
powerplant_data = pd.read_excel(file_path)

# Display basic information
print(powerplant_data.head(5))
print(powerplant_data.columns)
print("Dataset Shape:", powerplant_data.shape)
print(powerplant_data.info())
print("Missing Values:")
print(powerplant_data.isna().sum())
print("Unique Values per Column:")
print(powerplant_data.nunique())

# Splitting data into features and target variable
X = powerplant_data.iloc[:, :-1].values
y = powerplant_data.iloc[:, -1].values

# Splitting into train, validation, and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=True, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, shuffle=True, random_state=42)

# Print dataset shapes
print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of X_val:", X_val.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)
print("Shape of y_val:", y_val.shape)
```

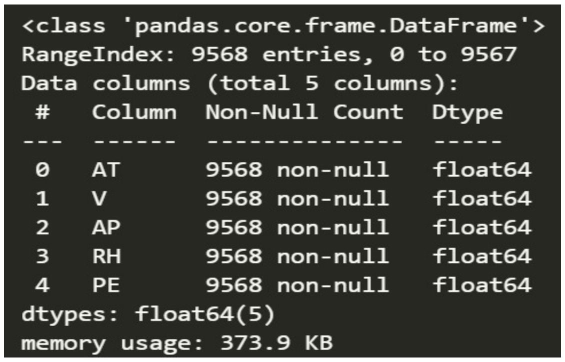


```python
# Build ANN model
classifier = Sequential()

# Adding layers
classifier.add(Dense(units=8, kernel_initializer='uniform', activation='relu', input_dim=4))  # Input layer
classifier.add(Dense(units=16, kernel_initializer='uniform', activation='relu'))  # Hidden layer 1
classifier.add(Dense(units=32, kernel_initializer='uniform', activation='relu'))  # Hidden layer 2
classifier.add(Dense(units=1, kernel_initializer='uniform'))  # Output layer

# Compile the model
classifier.compile(optimizer='adam', loss='mean_squared_error', metrics=['MeanSquaredLogarithmicError'])

# Train the model
model = classifier.fit(X_train, y_train, batch_size=32, epochs=200, validation_data=(X_val, y_val), shuffle=True)

# Predict on test data
y_pred = classifier.predict(X_test)

# Display predictions
np.set_printoptions(precision=2)
print(np.concatenate((y_pred.reshape(len(y_pred), 1), y_test.reshape(len(y_test), 1)), axis=1))

# Evaluate model
mae = sklearn.metrics.mean_absolute_error(y_test, classifier.predict(X_test))
mse = sklearn.metrics.mean_squared_error(y_test, classifier.predict(X_test))
rms = sqrt(mse)

# Print evaluation metrics
print('Mean Absolute Error     :', mae)
print('Mean Square Error       :', mse)
print('Root Mean Square Error  :', rms)

# Enable inline plotting
%matplotlib inline
```

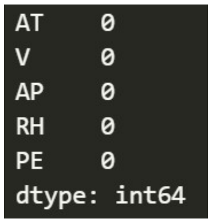
Output –



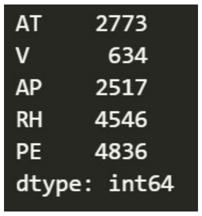
```
Index(['AT', 'V', 'AP', 'RH', 'PE'], dtype='object')
```



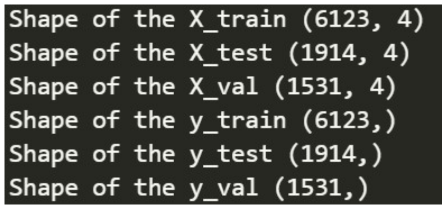
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9568 entries, 0 to 9567
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   AT      9568 non-null   float64
 1   V       9568 non-null   float64
 2   AP      9568 non-null   float64
 3   RH      9568 non-null   float64
 4   PE      9568 non-null   float64
dtypes: float64(5)
memory usage: 373.9 KB
```



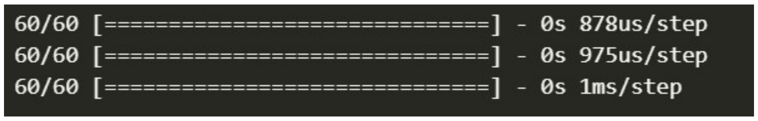
```
AT    0
V     0
AP    0
RH    0
PE    0
dtype: int64
```



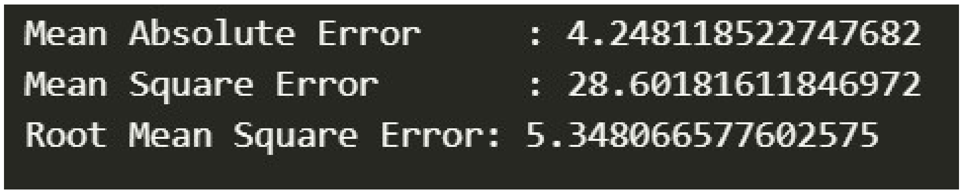
```
AT    2773
V      634
AP    2517
RH    4546
PE    4836
dtype: int64
```



```
Shape of the X_train (6123, 4)
Shape of the X_test (1914, 4)
Shape of the X_val (1531, 4)
Shape of the y_train (6123,)
Shape of the y_test (1914,)
Shape of the y_val (1531,)
```

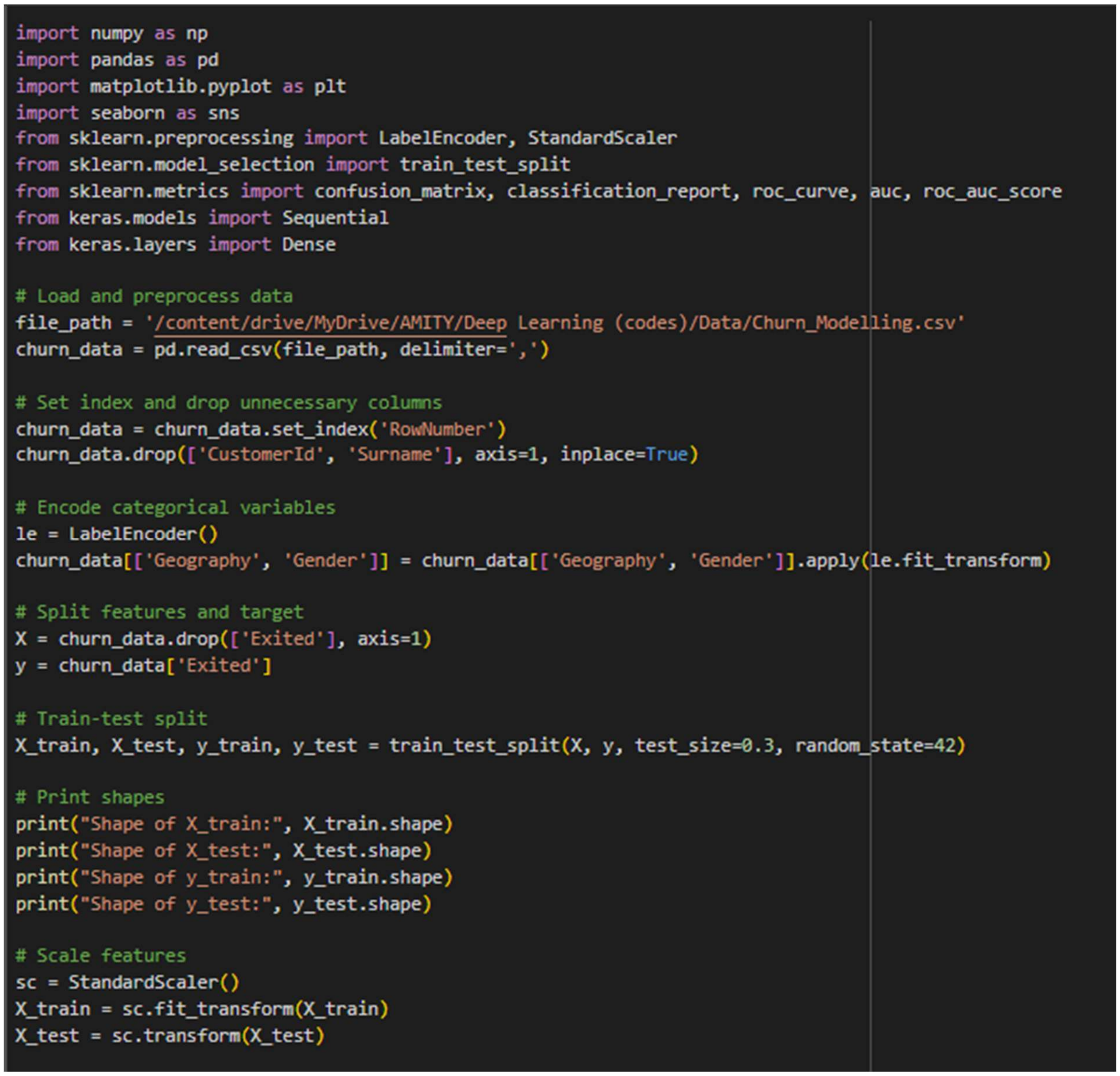


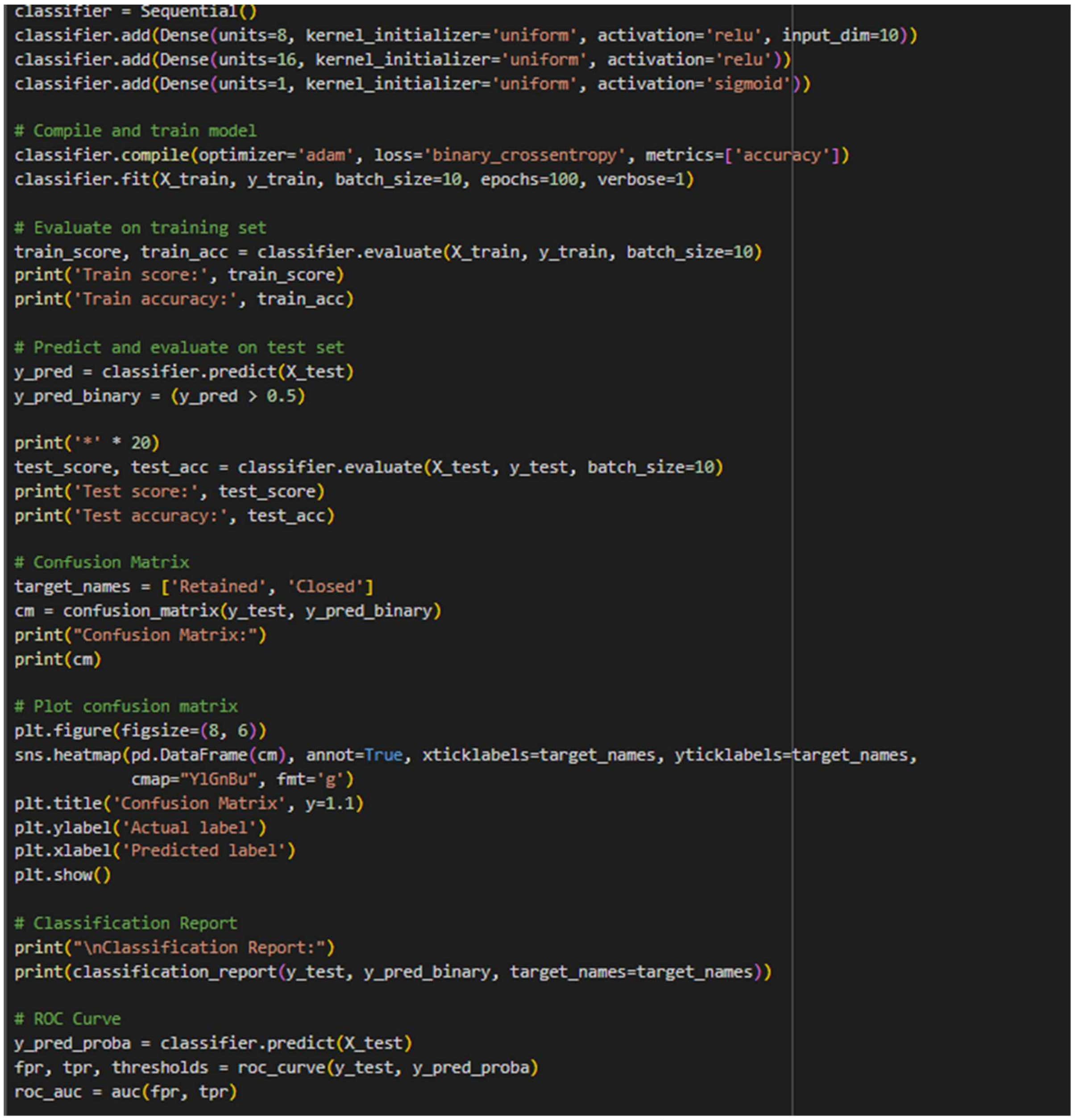
```
60/60 [==============================] - 0s 878us/step
60/60 [==============================] - 0s 975us/step
60/60 [==============================] - 0s 1ms/step
```



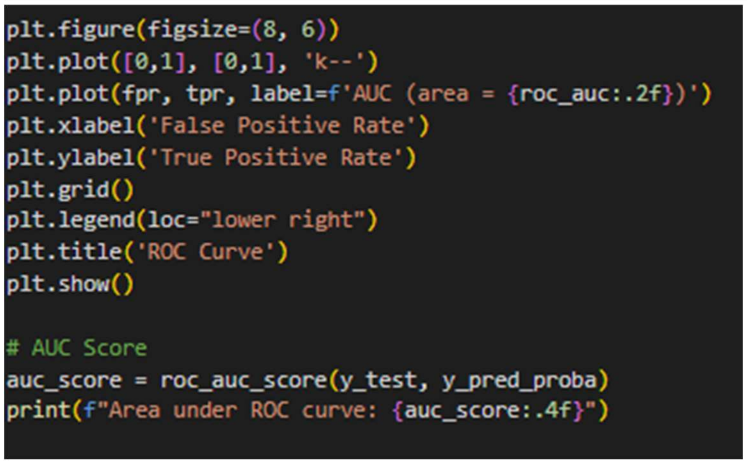
```
Mean Absolute Error     : 4.248118522747682
Mean Square Error       : 28.60181611846972
Root Mean Square Error: 5.348066577602575
```

b) **Churn Dataset**


```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc, roc_auc_score
from keras.models import Sequential
from keras.layers import Dense

# Load and preprocess data
file_path = '/content/drive/MyDrive/AMITY/Deep Learning (codes)/Data/Churn_Modelling.csv'
churn_data = pd.read_csv(file_path, delimiter=',')

# Set index and drop unnecessary columns
churn_data = churn_data.set_index('RowNumber')
churn_data.drop(['CustomerId', 'Surname'], axis=1, inplace=True)

# Encode categorical variables
le = LabelEncoder()
churn_data[['Geography', 'Gender']] = churn_data[['Geography', 'Gender']].apply(le.fit_transform)

# Split features and target
X = churn_data.drop(['Exited'], axis=1)
y = churn_data['Exited']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Print shapes
print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)

# Scale features
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

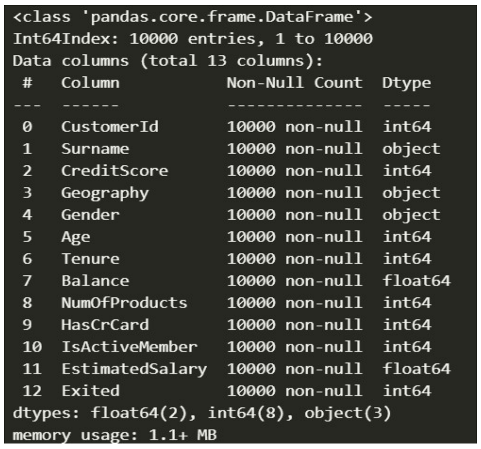


```python
classifier = Sequential()
classifier.add(Dense(units=8, kernel_initializer='uniform', activation='relu', input_dim=10))
classifier.add(Dense(units=16, kernel_initializer='uniform', activation='relu'))
classifier.add(Dense(units=1, kernel_initializer='uniform', activation='sigmoid'))

# Compile and train model
classifier.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
classifier.fit(X_train, y_train, batch_size=10, epochs=100, verbose=1)

# Evaluate on training set
train_score, train_acc = classifier.evaluate(X_train, y_train, batch_size=10)
print('Train score:', train_score)
print('Train accuracy:', train_acc)

# Predict and evaluate on test set
y_pred = classifier.predict(X_test)
y_pred_binary = (y_pred > 0.5)

print('*' * 20)
test_score, test_acc = classifier.evaluate(X_test, y_test, batch_size=10)
print('Test score:', test_score)
print('Test accuracy:', test_acc)

# Confusion Matrix
target_names = ['Retained', 'Closed']
cm = confusion_matrix(y_test, y_pred_binary)
print("Confusion Matrix:")
print(cm)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(pd.DataFrame(cm), annot=True, xticklabels=target_names, yticklabels=target_names,
            cmap="YlGnBu", fmt='g')
plt.title('Confusion Matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
plt.show()

# Classification Report
print("\nClassification Report:")
print(classification_report(y_test, y_pred_binary, target_names=target_names))

# ROC Curve
y_pred_proba = classifier.predict(X_test)
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)
```

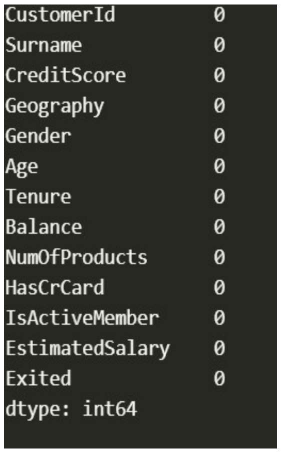


```python
plt.figure(figsize=(8, 6))
plt.plot([0,1], [0,1], 'k--')
plt.plot(fpr, tpr, label=f'AUC (area = {roc_auc:.2f})')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.grid()
plt.legend(loc="lower right")
plt.title('ROC Curve')
plt.show()

# AUC Score
auc_score = roc_auc_score(y_test, y_pred_proba)
print(f"Area under ROC curve: {auc_score:.4f}")
```

Output –

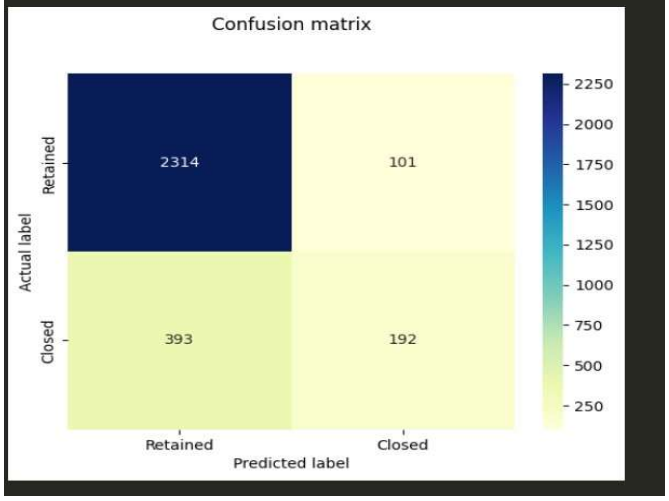


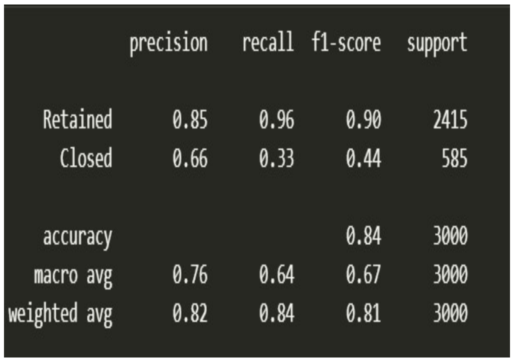
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10000 entries, 1 to 10000
Data columns (total 13 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   CustomerId       10000 non-null  int64  
 1   Surname          10000 non-null  object 
 2   CreditScore      10000 non-null  int64  
 3   Geography        10000 non-null  object 
 4   Gender           10000 non-null  object 
 5   Age              10000 non-null  int64  
 6   Tenure           10000 non-null  int64  
 7   Balance          10000 non-null  float64
 8   NumOfProducts    10000 non-null  int64  
 9   HasCrCard        10000 non-null  int64  
 10  IsActiveMember   10000 non-null  int64  
 11  EstimatedSalary  10000 non-null  float64
 12  Exited           10000 non-null  int64  
dtypes: float64(2), int64(8), object(3)
memory usage: 1.1+ MB
```



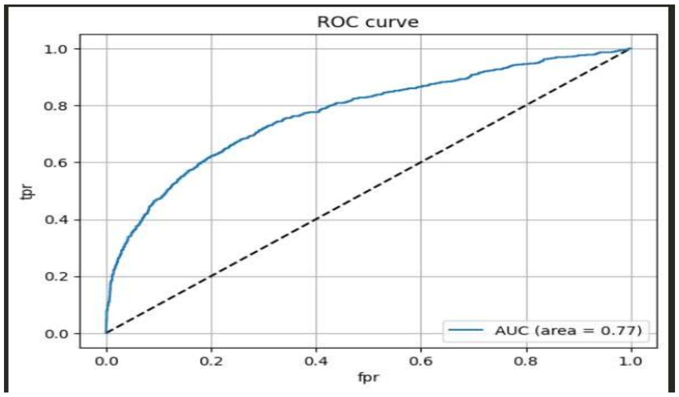
```
CustomerId         0
Surname            0
CreditScore        0
Geography          0
Gender             0
Age                0
Tenure             0
Balance            0
NumOfProducts      0
HasCrCard          0
IsActiveMember     0
EstimatedSalary    0
Exited             0
dtype: int64
```

```
Text(0.5, 23.52222222222222, 'Predicted label')
```





```
              precision    recall  f1-score   support

    Retained       0.85      0.96      0.90      2415
      Closed       0.66      0.33      0.44       585

    accuracy                           0.84      3000
   macro avg       0.76      0.64      0.67      3000
weighted avg       0.82      0.84      0.81      3000
```



Experiment 9

Aim:

To implement Convolutional Neural Networks (CNNs) using TensorFlow for:
a. 3×3 image and 2×2 filter (manual calculation and Python code).
b. 32×32 grayscale image with multiple convolutional and pooling layers.

Theory: In this experiment, we implement a CNN using TensorFlow for two different tasks:

1. 3×3 Image and 2×2 Filter (Manual Calculation & Code):

- A 3×3 input image and a 2×2 filter are given.
- Convolution operation is performed by sliding the filter over the image with stride = 1 and no padding.
- The output feature map is computed as follows:
  - Perform element-wise multiplication between the filter and the corresponding image region.
  - Sum the results for each position.
  - Since the valid output size is determined using the formula:
    - Output size = (input size – filter size)/stride +1
  - the resulting feature map is 2×2.

A Python/TensorFlow implementation is provided to compute this operation programmatically.

2. CNN with 32×32 Grayscale Image (Using TensorFlow):
A 32×32 grayscale image is passed through a series of CNN layers:

Layer-wise Operations & Output Dimensions:

1. Convolutional Layer

   6 filters, each of size 5×5, stride = 1, no padding.

   Output size: (32−5)/1+1=28(32 - 5)/1 + 1 = 28(32−5)/1+1=28

   → Output: 28×28×6
2. Max Pooling Layer

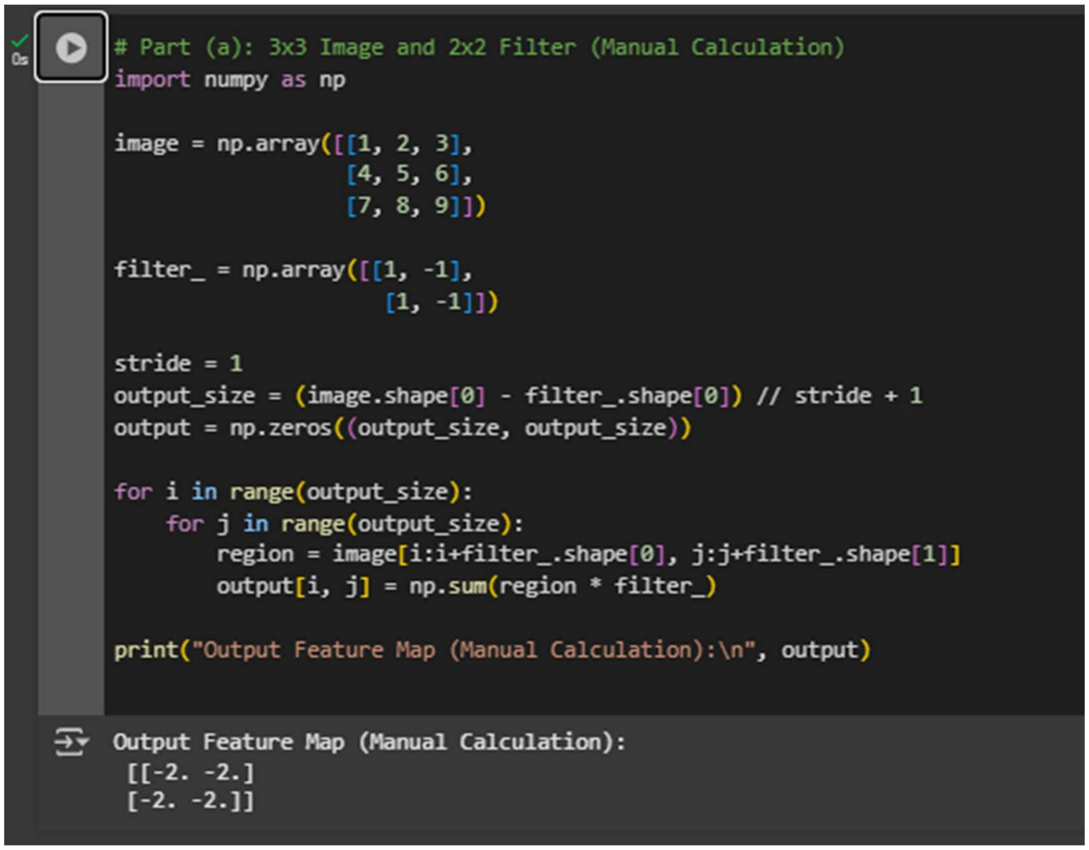
   Pool size = 2×2, stride = 2.
   Output size: (28−2)/2+1=14(28 - 2)/2 + 1 = 14(28−2)/2+1=14
   → Output: 14×14×6
3. Convolutional Layer

   16 filters, each of size 3×3, stride = 1, no padding.
   Output size: (14−3)/1+1=12(14 - 3)/1 + 1 = 12(14−3)/1+1=12
   → Output: 12×12×16
4. Max Pooling Layer

   Pool size = 2×2, stride = 2.
   Output size: (12−2)/2+1=6(12 - 2)/2 + 1 = 6(12−2)/2+1=6
   → Final Output: 6×6×16

A TensorFlow/Python implementation is provided to verify these calculations and print the output dimensions at each step.

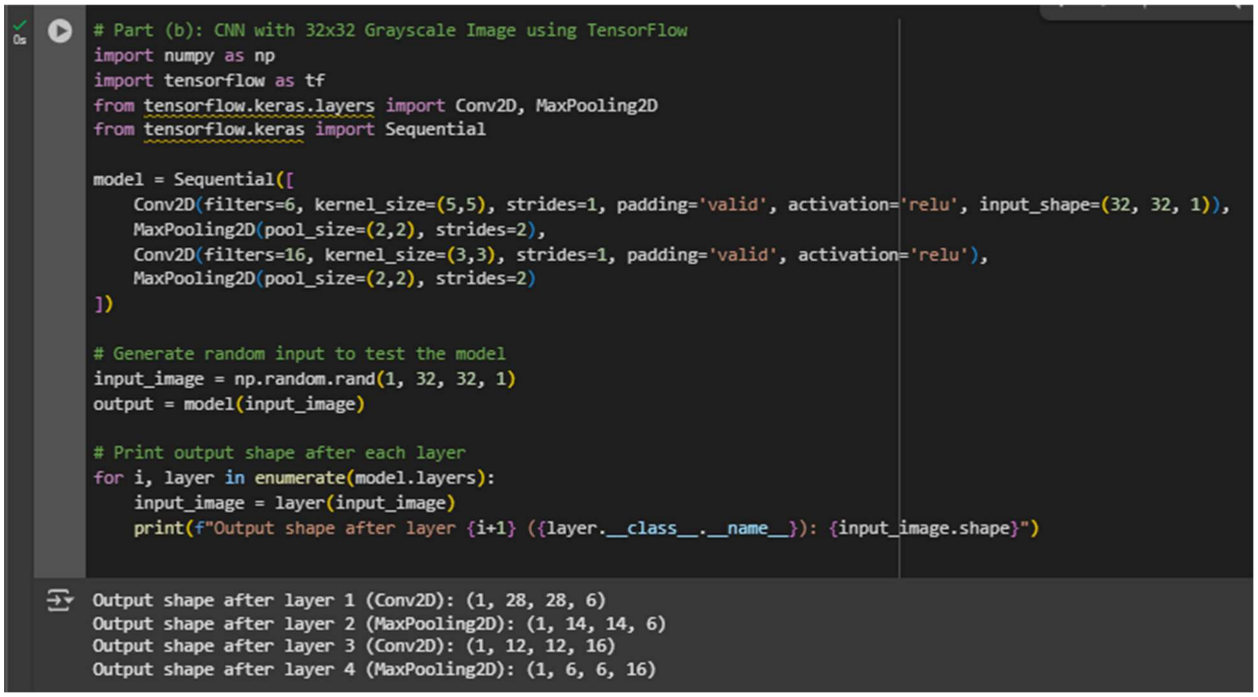
Code –



```
# Part (a): 3x3 Image and 2x2 Filter (Manual Calculation)
import numpy as np

image = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

filter_ = np.array([[1, -1],
                    [1, -1]])

stride = 1
output_size = (image.shape[0] - filter_.shape[0]) // stride + 1
output = np.zeros((output_size, output_size))

for i in range(output_size):
    for j in range(output_size):
        region = image[i:i+filter_.shape[0], j:j+filter_.shape[1]]
        output[i, j] = np.sum(region * filter_)

print("Output Feature Map (Manual Calculation):\n", output)
```

```
Output Feature Map (Manual Calculation):
 [[-2. -2.]
 [-2. -2.]]
```

a)



```
# Part (b): CNN with 32x32 Grayscale Image using TensorFlow
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras import Sequential

model = Sequential([
    Conv2D(filters=6, kernel_size=(5,5), strides=1, padding='valid', activation='relu', input_shape=(32, 32, 1)),
    MaxPooling2D(pool_size=(2,2), strides=2),
    Conv2D(filters=16, kernel_size=(3,3), strides=1, padding='valid', activation='relu'),
    MaxPooling2D(pool_size=(2,2), strides=2)
])

# Generate random input to test the model
input_image = np.random.rand(1, 32, 32, 1)
output = model(input_image)

# Print output shape after each layer
for i, layer in enumerate(model.layers):
    input_image = layer(input_image)
    print(f"Output shape after layer {i+1} ({layer.__class__.__name__}): {input_image.shape}")
```

```
Output shape after layer 1 (Conv2D): (1, 28, 28, 6)
Output shape after layer 2 (MaxPooling2D): (1, 14, 14, 6)
Output shape after layer 3 (Conv2D): (1, 12, 12, 16)
Output shape after layer 4 (MaxPooling2D): (1, 6, 6, 16)
```

b)

Experiment 10

Aim:

To implement a Recurrent Neural Network (RNN) using TensorFlow with the following parameters:

- **Input sequence:** x=[1,2,3]x = [1, 2, 3]
- **Input-hidden layer weights:** Wh=0.5W_h = 0.5, Vh=0.3V_h = 0.3
- **Output layer weights:** Wy=0.7W_y = 0.7
- **Bias terms:** bh=0.1b_h = 0.1, by=0.2b_y = 0.2
- **Initial hidden state:** h0=0h_0 = 0
- **Activation function:** tanh \tanh

Theory:

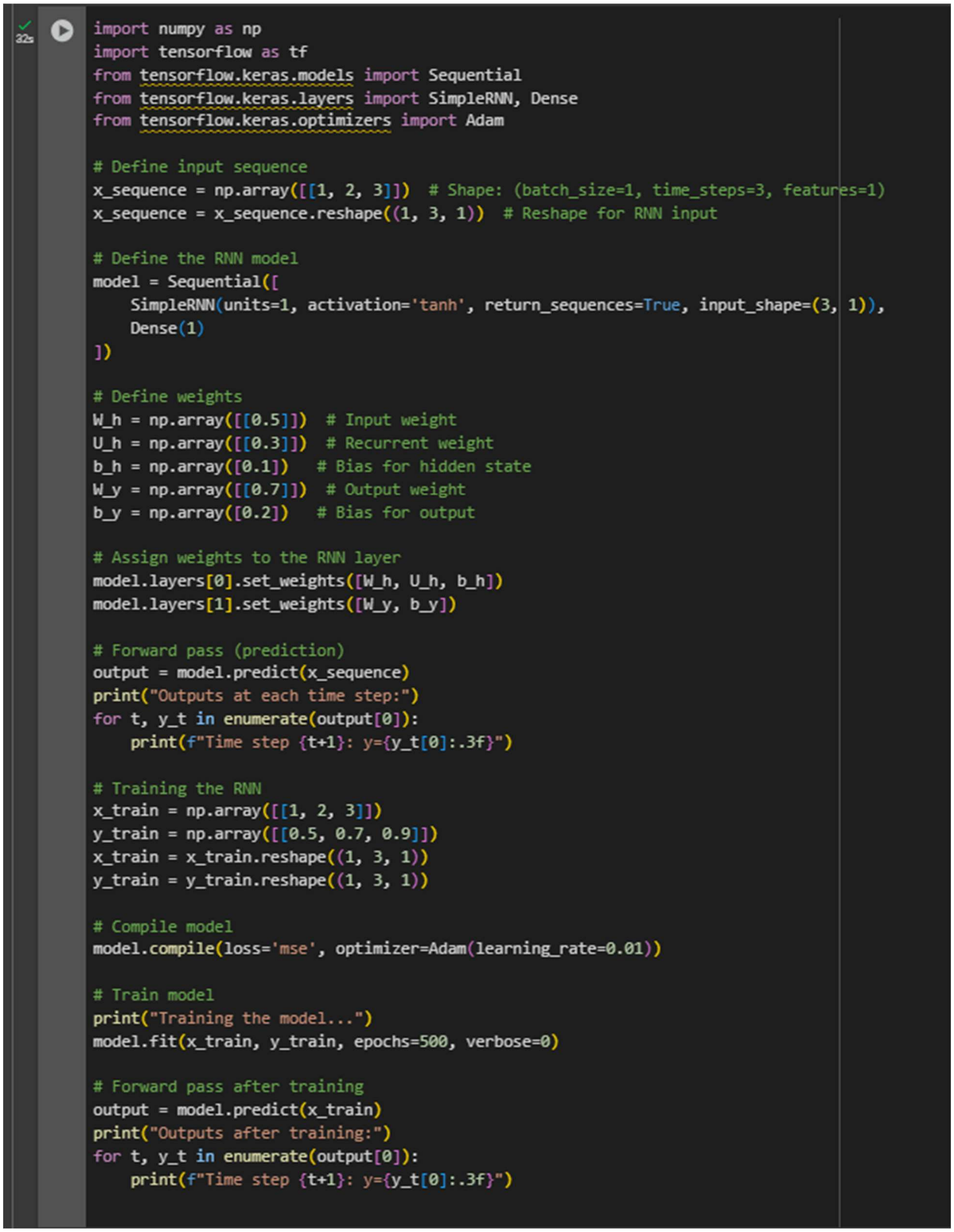
This experiment demonstrates the implementation of an RNN using TensorFlow in two parts:

a) **Forward Pass Without Training:**
- A simple RNN processes the input sequence using predefined weights and biases.
- The hidden state and output at each time step are computed using the tanh \tanh activation function.

b) **Training the Model:**
- The model is trained using the input sequence xtrain=[1,2,3]x_{\text{train}} = [1, 2, 3] with target outputs ytrain=[0.5,0.7,0.9]y_{\text{train}} = [0.5, 0.7, 0.9].
- Training is performed using backpropagation with the Mean Squared Error (MSE) loss function, optimized by the Adam optimizer.
- The model undergoes **500 epochs**, learning to adjust its weights for better predictions.

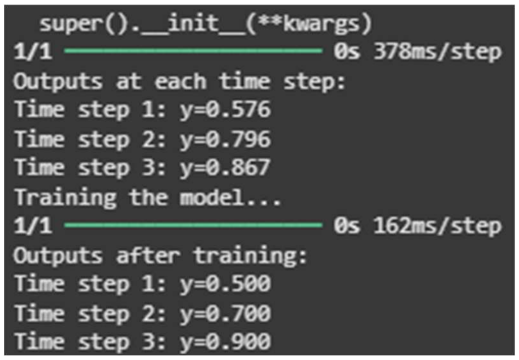
This experiment provides a foundational understanding of RNNs, covering both fixed-weight inference and the learning process through training.

Code –



```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.optimizers import Adam

# Define input sequence
x_sequence = np.array([[1, 2, 3]])  # Shape: (batch_size=1, time_steps=3, features=1)
x_sequence = x_sequence.reshape((1, 3, 1))  # Reshape for RNN input

# Define the RNN model
model = Sequential([
    SimpleRNN(units=1, activation='tanh', return_sequences=True, input_shape=(3, 1)),
    Dense(1)
])

# Define weights
W_h = np.array([[0.5]])  # Input weight
U_h = np.array([[0.3]])  # Recurrent weight
b_h = np.array([0.1])   # Bias for hidden state
W_y = np.array([[0.7]])  # Output weight
b_y = np.array([0.2])   # Bias for output

# Assign weights to the RNN layer
model.layers[0].set_weights([W_h, U_h, b_h])
model.layers[1].set_weights([W_y, b_y])

# Forward pass (prediction)
output = model.predict(x_sequence)
print("Outputs at each time step:")
for t, y_t in enumerate(output[0]):
    print(f"Time step {t+1}: y={y_t[0]:.3f}")

# Training the RNN
x_train = np.array([[1, 2, 3]])
y_train = np.array([[0.5, 0.7, 0.9]])
x_train = x_train.reshape((1, 3, 1))
y_train = y_train.reshape((1, 3, 1))

# Compile model
model.compile(loss='mse', optimizer=Adam(learning_rate=0.01))

# Train model
print("Training the model...")
model.fit(x_train, y_train, epochs=500, verbose=0)

# Forward pass after training
output = model.predict(x_train)
print("Outputs after training:")
for t, y_t in enumerate(output[0]):
    print(f"Time step {t+1}: y={y_t[0]:.3f}")
```

Output –



```
  super().__init__(**kwargs)
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 378ms/step
Outputs at each time step:
Time step 1: y=0.576
Time step 2: y=0.796
Time step 3: y=0.867
Training the model...
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 162ms/step
Outputs after training:
Time step 1: y=0.500
Time step 2: y=0.700
Time step 3: y=0.900
```