Practical No:- 01

Aim: I. Introduction to TensorFlow:

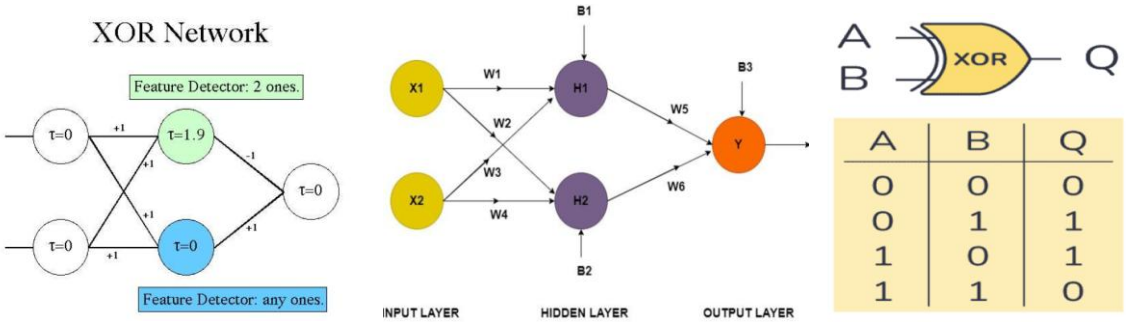
a. Create tensors with different shapes and data types.
b. Perform basic operations like addition, subtraction, multiplication, and division on tensors.
c. Reshape, slice, and index tensors to extract specific elements or sections.
d. Performing matrix multiplication and finding eigenvectors and eigenvalues using TensorFlow.

II. Program to solve the XOR problem using deep forward network.

Description:

A **deep forward network**, also known as a **feed-forward neural network**, is a type of artificial neural network in which information flows in only one direction, from the input layer to the output layer, without forming any cycles or feedback loops. The network is called *deep* because it contains one or more hidden layers between the input and output layers. Each layer is made up of neurons that are connected to neurons in the next layer through weighted connections. The purpose of a feed-forward network is to learn a mapping between input data and output data by adjusting these weights during training.

In this experiment, a deep forward network is used to solve the **XOR (Exclusive OR)** problem. The XOR problem is a logical function where the output is 1 if the two input values are different and 0 if they are the same. XOR is a **non-linearly separable problem**, meaning it cannot be solved using a single-layer perceptron or a straight line decision boundary. Therefore, a hidden layer is required to introduce non-linearity into the model.



A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

The diagram shows a deep forward neural network used to solve the XOR problem. The input layer takes two inputs. A hidden layer with tanh activation introduces non-linearity, which is essential because XOR is not linearly separable. The output layer uses sigmoid activation to produce binary output. During training, weights are updated using gradient descent to minimize binary cross-entropy loss.

The network architecture consists of an **input layer** with two neurons corresponding to the two XOR inputs, a **hidden layer** with two neurons using the *tanh* activation function, and an **output layer** with one neuron using the *sigmoid* activation function. The tanh activation function helps the network learn non-linear relationships, while the sigmoid activation function converts the output into a value between 0 and 1, making it suitable for binary classification. During training, the network uses a loss function called

binary cross-entropy to measure prediction error and an optimizer such as **stochastic gradient descent (SGD)** to update the weights and minimize this error.

After training, the feed-forward network is able to approximate the XOR function by correctly classifying most input combinations. This experiment demonstrates that deep forward networks can successfully solve non-linear problems by using hidden layers and non-linear activation functions. It also highlights the importance of network architecture, activation functions, and optimization techniques in training neural networks effectively.

Code: # I. Introduction to tensorflow.

```
import tensorflow as tf

# a) Create tensors with different shapes and data types

t1 = tf.constant([1, 2, 3], dtype=tf.int32)                  # 1D integer tensor

t2 = tf.constant([[1.0, 2.0], [3.0, 4.0]], dtype=tf.float32)    # 2D float tensor

t3 = tf.constant([[True, False], [False, True]], dtype=tf.bool) # 2D boolean tensor

print("t1:", t1, "shape:", t1.shape, "dtype:", t1.dtype)

print("t2:", t2, "shape:", t2.shape, "dtype:", t2.dtype)

print("t3:", t3, "shape:", t3.shape, "dtype:", t3.dtype)

# b) Basic operations: +, -, *, /

a = tf.constant([1.0, 2.0, 3.0])

b = tf.constant([4.0, 5.0, 6.0])

add_result = tf.add(a, b)

sub_result = tf.subtract(a, b)

mul_result = tf.multiply(a, b)

div_result = tf.divide(a, b)

print("\nAddition:", add_result)

print("Subtraction:", sub_result)

print("Multiplication:", mul_result)

print("Division:", div_result)

# c) Reshape, slice, and index tensors

x = tf.constant([[1, 2, 3],
```

```
            [4, 5, 6],
            [7, 8, 9]])
reshaped_x = tf.reshape(x, (1, 9))   # 3x3 -> 1x9
row_0 = x[0, :]                  # first row
col_1 = x[:, 1]                  # second column
element = x[1, 2]                # element at row 1, col 2 (0-based index)
print("\nOriginal x:\n", x)
print("Reshaped x (1x9):", reshaped_x)
print("First row:", row_0)
print("Second column:", col_1)
print("Element at (1,2):", element)
# d) Matrix multiplication and eigenvalues/eigenvectors
A = tf.constant([[1.0, 2.0],
            [3.0, 4.0]])
B = tf.constant([[5.0, 6.0],
            [7.0, 8.0]])
matmul_result = tf.matmul(A, B)
print("\nMatrix A:\n", A)
print("Matrix B:\n", B)
print("A x B:\n", matmul_result)
# Eigenvalues and eigenvectors of A
A_complex = tf.cast(A, tf.complex64)   # eig works on complex type
eigenvalues, eigenvectors = tf.linalg.eig(A_complex)
print("\nEigenvalues of A:", eigenvalues)
print("Eigenvectors of A:\n", eigenvectors)
```

Output:

```
========== RESTART: C:/Users/Smita Manoj Rajwadia/AppDat
t1: tf.Tensor([1 2 3], shape=(3,), dtype=int32) shape: (3,) dtype: <dtype: 'int32'>
t2: tf.Tensor(
[[1. 2.]
 [3. 4.]], shape=(2, 2), dtype=float32) shape: (2, 2) dtype: <dtype: 'float32'>
t3: tf.Tensor(
[[ True False]
 [False  True]], shape=(2, 2), dtype=bool) shape: (2, 2) dtype: <dtype: 'bool'>

Addition: tf.Tensor([5. 7. 9.], shape=(3,), dtype=float32)
Subtraction: tf.Tensor([-3. -3. -3.], shape=(3,), dtype=float32)
Multiplication: tf.Tensor([ 4. 10. 18.], shape=(3,), dtype=float32)
Division: tf.Tensor([0.25 0.4  0.5 ], shape=(3,), dtype=float32)

Original x:
 tf.Tensor(
[[1 2 3]
 [4 5 6]
 [7 8 9]], shape=(3, 3), dtype=int32)
Reshaped x (1x9): tf.Tensor([[1 2 3 4 5 6 7 8 9]], shape=(1, 9), dtype=int32)
First row: tf.Tensor([1 2 3], shape=(3,), dtype=int32)
Second column: tf.Tensor([2 5 8], shape=(3,), dtype=int32)
Element at (1,2): tf.Tensor(6, shape=(), dtype=int32)

Matrix A:
 tf.Tensor(
[[1. 2.]
 [3. 4.]], shape=(2, 2), dtype=float32)
Matrix B:
 tf.Tensor(
[[5. 6.]
 [7. 8.]], shape=(2, 2), dtype=float32)
A x B:
 tf.Tensor(
[[19. 22.]
 [43. 50.]], shape=(2, 2), dtype=float32)

Eigenvalues of A: tf.Tensor([-0.37228122+0.j  5.372281  +0.j], shape=(2,), dtype=complex64)
Eigenvectors of A:
 tf.Tensor(
[[-0.8245648 +0.j -0.41597357+0.j]
 [ 0.56576747+0.j -0.90937674+0.j]], shape=(2, 2), dtype=complex64)
```

Code: # II. Program to solve the XOR problem using deep forward network.

```
import tensorflow as tf

import numpy as np

# XOR Dataset

X = np.array([

    [0, 0],

    [0, 1],

    [1, 0],

    [1, 1]

], dtype=np.float32)


y = np.array([

    [0],
```

```
    [1],
    [1],
    [0]
], dtype=np.float32)
# Deep Forward (Feed-Forward) Neural Network
model = tf.keras.Sequential([
    tf.keras.Input(shape=(2,)),               # Input layer
    tf.keras.layers.Dense(4, activation='tanh'),# Hidden layer (tanh kept)
    tf.keras.layers.Dense(1, activation='sigmoid')  # Output layer
])
# Compile the model
model.compile(
    optimizer='adam',                  # Faster & stable optimizer
    loss='binary_crossentropy',        # For binary classification
    metrics=['accuracy']
)
# Train the model
model.fit(X, y, epochs=3000, verbose=0)
# Test the model
predictions = model.predict(X)
print("Inputs:\n", X)
print("True outputs:\n", y)
print("Predicted outputs (probabilities):\n", predictions)
print("Predicted outputs (rounded):\n", np.round(predictions))
```

Output:

```
_________________________________
[1m1/1[0m [32m━━━━━━━━━━━━━━━━[0m [0m[1m0s[0m 39ms/step
Inputs:
 [[0. 0.]
 [0. 1.]
 [1. 0.]
 [1. 1.]]
True outputs:
 [[0.]
 [1.]
 [1.]
 [0.]]
Predicted outputs (probabilities):
 [[0.03028351]
 [0.97806555]
 [0.9718435 ]
 [0.02020486]]
Predicted outputs (rounded):
 [[0.]
 [1.]
 [1.]
 [0.]]
```

Learnings:

From this experiment, we learned that a **feed-forward (deep) neural network** can solve complex **non-linear problems** like the XOR function, which cannot be solved using a single-layer perceptron. We understood the importance of **hidden layers** and **non-linear activation functions** in learning complex patterns. The experiment also helped us understand how a neural network is **trained using a loss function and optimizer** to minimize error. Overall, this practical gave hands-on understanding of how deep forward networks learn input–output relationships using TensorFlow.