Practical:1

AIM: Implement the AND,OR, NAND and NOR using a Perceptron.

```
def perceptron(x1, x2, w1, w2, bias):
    # Calculate weighted sum
    net = (x1 * w1) + (x2 * w2) + bias

    # Apply Step Activation Function
    if net >= 0:
        return 1
    else:
        return 0

# Input combinations
inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]

# AND Gate
print("AND Gate")
print("X1 X2 Output")
for x1, x2 in inputs:
    output = perceptron(x1, x2, 1, 1, -1.5)
    print(x1, x2, " ", output)

# OR Gate
print("\nOR Gate")
print("X1 X2 Output")
```

```
for x1, x2 in inputs:
    output = perceptron(x1, x2, 1, 1, -0.5)
    print(x1, x2, " ", output)

# NAND Gate
print("\nNAND Gate")
print("X1 X2 Output")
for x1, x2 in inputs:
    output = perceptron(x1, x2, -1, -1, 1.5)
    print(x1, x2, " ", output)

# NOR Gate
print("\nNOR Gate")
print("X1 X2 Output")
for x1, x2 in inputs:
    output = perceptron(x1, x2, -1, -1, 0.5)
    print(x1, x2, " ", output)
```

OUTPUT:

```
AND Gate
X1 X2 Output
0 0   0
0 1   0
1 0   0
1 1   1
```

OR Gate

X1 X2 Output

0 0 0

0 1 1

1 0 1

1 1 1

NAND Gate

X1 X2 Output

0 0 1

0 1 1

1 0 1

1 1 0

NOR Gate

X1 X2 Output

0 0 1

0 1 0

1 0 0

1 1 0

Practical:2

Aim: To Implement the logic behind the Recurrent Neural Network.

```
import numpy as np

# Activation function (tanh)
def tanh(x):
    return np.tanh(x)

# Input sequence
inputs = [1, 2, 3, 4]

# Initialize weights
Wx = 0.5    # Weight for input
Wh = 0.8    # Weight for previous hidden state
b = 0.1     # Bias

# Initial hidden state
h = 0

print("Recurrent Neural Network (RNN) Implementation")
print("--------------------------------------------")

# Process each input sequentially
for t, x in enumerate(inputs):
```

```
    h = tanh(Wx * x + Wh * h + b)
    print(f"Time Step {t+1}")
    print(f"Input = {x}")
    print(f"Hidden State = {h:.4f}")
print()
```

Practical:3

Aim: To Implement the logic behind an Artificial Neural Network.

```
import numpy as np

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Input values
X = np.array([0.5, 0.8])

# Weights from Input Layer to Hidden Layer
W_input_hidden = np.array([[0.2, 0.4],
                           [0.3, 0.7]])

# Bias for Hidden Layer
B_hidden = np.array([0.1, 0.2])

# Weights from Hidden Layer to Output Layer
W_hidden_output = np.array([0.6, 0.9])

# Bias for Output Layer
B_output = 0.3

# Forward Propagation
```

```
# Hidden Layer Calculation
hidden_input = np.dot(X, W_input_hidden) + B_hidden
hidden_output = sigmoid(hidden_input)

# Output Layer Calculation
final_input = np.dot(hidden_output, W_hidden_output) + B_output
final_output = sigmoid(final_input)

# Display Results
print("Artificial Neural Network (ANN)")
print("--------------------------------")
print("Input:", X)
print("Hidden Layer Output:", hidden_output)
print("Final Output:", final_output)
```

Practical:4

Aim: Implementing the techniques of Convolutional Neural Network (CNN)

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Create CNN Model
model = Sequential()

# Convolution Layer
model.add(Conv2D(filters=32,
                 kernel_size=(3,3),
                 activation='relu',
                 input_shape=(28,28,1)))

# Max Pooling Layer
model.add(MaxPooling2D(pool_size=(2,2)))

# Second Convolution Layer
model.add(Conv2D(filters=64,
                 kernel_size=(3,3),
                 activation='relu'))

# Second Max Pooling Layer
```

```
model.add(MaxPooling2D(pool_size=(2,2)))

# Flatten Layer
model.add(Flatten())

# Fully Connected Layer
model.add(Dense(128, activation='relu'))

# Output Layer (10 Classes)
model.add(Dense(10, activation='softmax'))

# Display Model Summary
model.summary()
```

Practical:5

Aim: Implement the XOR and XNOR using a Multilayer perceptron.

```
from sklearn.neural_network import MLPClassifier


# Input combinations
X = [[0,0],
    [0,1],
    [1,0],
    [1,1]]


# XOR Output
y_xor = [0,1,1,0]


# XNOR Output
y_xnor = [1,0,0,1]


# Create MLP Classifier
mlp_xor = MLPClassifier(hidden_layer_sizes=(2,),
                        activation='logistic',
                        solver='lbfgs',
                        max_iter=1000,
                        random_state=1)


# Train XOR Model
mlp_xor.fit(X, y_xor)
```

```
print("XOR Gate")
print("X1 X2 Output")
for x in X:
    print(x[0], x[1], " ", mlp_xor.predict([x])[0])

# Create another MLP for XNOR
mlp_xnor = MLPClassifier(hidden_layer_sizes=(2,),
                         activation='logistic',
                         solver='lbfgs',
                         max_iter=1000,
                         random_state=1)

# Train XNOR Model
mlp_xnor.fit(X, y_xnor)

print("\nXNOR Gate")
print("X1 X2 Output")
for x in X:
    print(x[0], x[1], " ", mlp_xnor.predict([x])[0])
```

Practical:6

Aim: Implementing Particle Swarm Optimization (PSO)

```
import random
# Objective Function
def objective_function(x):
    return x ** 2

# PSO Parameters
num_particles = 5
max_iterations = 20
w = 0.5      # Inertia Weight
c1 = 1.5     # Cognitive Constant
c2 = 1.5     # Social Constant

# Initialize particles
particles = []
velocities = []
pbest = []
pbest_value = []

for i in range(num_particles):
    position = random.uniform(-10, 10)
    velocity = random.uniform(-1, 1)
```

```
    particles.append(position)
    velocities.append(velocity)
    pbest.append(position)
    pbest_value.append(objective_function(position))

# Global Best
gbest = pbest[pbest_value.index(min(pbest_value))]

# PSO Algorithm
for iteration in range(max_iterations):

    for i in range(num_particles):

        r1 = random.random()
        r2 = random.random()

        # Update Velocity
        velocities[i] = (w * velocities[i] +
                         c1 * r1 * (pbest[i] - particles[i]) +
                         c2 * r2 * (gbest - particles[i]))

        # Update Position
        particles[i] = particles[i] + velocities[i]

        # Update Personal Best
```

```
        fitness = objective_function(particles[i])

        if fitness < pbest_value[i]:
            pbest[i] = particles[i]
            pbest_value[i] = fitness

    # Update Global Best
    gbest = pbest[pbest_value.index(min(pbest_value))]

    print("Iteration", iteration + 1,
          "Best Position =", round(gbest, 4),
          "Best Value =", round(objective_function(gbest), 4))

print("\nOptimal Solution:", round(gbest, 4))
print("Minimum Function Value:", round(objective_function(gbest), 4))
```

Practical:7

Aim: Implementing Backpropagation Algorithm

```
import numpy as np

# Input data (XOR problem)
X = np.array([[0,0],
              [0,1],
              [1,0],
              [1,1]])

# Expected output
Y = np.array([[0],
              [1],
              [1],
              [0]])

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid
def sigmoid_derivative(x):
    return x * (1 - x)
```

```
# Initialize weights randomly
np.random.seed(1)
input_neurons = 2
hidden_neurons = 2
output_neurons = 1

weights_input_hidden = np.random.uniform(size=(input_neurons, hidden_neurons))
weights_hidden_output = np.random.uniform(size=(hidden_neurons, output_neurons))

# Biases
bias_hidden = np.random.uniform(size=(1, hidden_neurons))
bias_output = np.random.uniform(size=(1, output_neurons))

learning_rate = 0.5
epochs = 10000

# Training using Backpropagation
for epoch in range(epochs):

    # Forward Propagation
    hidden_input = np.dot(X, weights_input_hidden) + bias_hidden
    hidden_output = sigmoid(hidden_input)

    final_input = np.dot(hidden_output, weights_hidden_output) + bias_output
    predicted_output = sigmoid(final_input)
```

```
    # Calculate Error
    error = Y - predicted_output

    # Backpropagation
    d_output = error * sigmoid_derivative(predicted_output)

    hidden_error = d_output.dot(weights_hidden_output.T)
    d_hidden = hidden_error * sigmoid_derivative(hidden_output)

    # Update Weights and Biases
    weights_hidden_output += hidden_output.T.dot(d_output) * learning_rate
    weights_input_hidden += X.T.dot(d_hidden) * learning_rate

    bias_output += np.sum(d_output, axis=0, keepdims=True) * learning_rate
    bias_hidden += np.sum(d_hidden, axis=0, keepdims=True) * learning_rate

# Display Results
print("Predicted Output after Training:")
print(np.round(predicted_output, 3))
```

Practical:8

Aim: Implementing Naive Bayes Classification

```
from sklearn.naive_bayes import GaussianNB

# Training data
# Features: [Height (cm), Weight (kg)]
X = [
    [150, 50],
    [160, 60],
    [170, 65],
    [180, 80],
    [155, 55],
    [175, 70]
]

# Target labels
# 0 = Female, 1 = Male
y = [0, 0, 1, 1, 0, 1]

# Create Gaussian Naive Bayes model
model = GaussianNB()

# Train the model
model.fit(X, y)
```

```
# Test data
test_data = [[165, 58]]

# Predict class
prediction = model.predict(test_data)

# Display result
print("Test Data:", test_data)

if prediction[0] == 0:
    print("Predicted Class: Female")
else:
    print("Predicted Class: Male")
```

Practical:9

Aim: Implementing a Generative Adversarial Network (GAN)

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# ----------------------------
# Generator Model
# ----------------------------
generator = Sequential([
    Dense(128, activation='relu', input_dim=100),
    Dense(256, activation='relu'),
    Dense(784, activation='sigmoid')   # Output: 28x28 image (flattened)
])

# ----------------------------
# Discriminator Model
# ----------------------------
discriminator = Sequential([
    Dense(256, activation='relu', input_dim=784),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

```
# Compile Discriminator
discriminator.compile(
    optimizer=Adam(0.0002),
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# ----------------------------
# Build GAN
# ----------------------------
discriminator.trainable = False

gan = Sequential([
    generator,
    discriminator
])

# Compile GAN
gan.compile(
    optimizer=Adam(0.0002),
    loss='binary_crossentropy'
)

# Display Model Summary
print("Generator Model:")
```

```
generator.summary()

print("\nDiscriminator Model:")
discriminator.summary()

print("\nGAN Model:")
gan.summary()
```

Practical:10

Aim: Implementing Principal Component Analysis (PCA)

```
from sklearn.decomposition import PCA
import numpy as np
# Sample Dataset
X = np.array([
    [2.5, 2.4],
    [0.5, 0.7],
    [2.2, 2.9],
    [1.9, 2.2],
    [3.1, 3.0],
    [2.3, 2.7],
    [2.0, 1.6],
    [1.0, 1.1],
    [1.5, 1.6],
    [1.1, 0.9]
])

# Create PCA object (Reduce to 1 Principal Component)
pca = PCA(n_components=1)

# Fit and Transform the data
X_pca = pca.fit_transform(X)

# Display Results
```

```
print("Original Data:")
print(X)

print("\nReduced Data using PCA:")
print(X_pca)

print("\nExplained Variance Ratio:")
print(pca.explained_variance_ratio_)
```