Q1. Write a Python program to plot a Linear Activation Function that is used in Neural Networks.

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

# Linear activation function: f(x) = x
x = np.linspace(-10, 10, 100)
y = x

plt.plot(x, y, label="Linear Activation")
plt.title("Linear Activation Function")
plt.xlabel("Input")
plt.ylabel("Output")
plt.grid(True)
plt.legend()
plt.show()
```

Q2. Write a Python program to plot a Sigmoid Activation Function that is used in Neural Networks.

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

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

x = np.linspace(-10, 10, 100)
y = sigmoid(x)

plt.plot(x, y, label="Sigmoid", color='blue')
plt.title("Sigmoid Activation Function")
plt.xlabel("Input")
plt.ylabel("Output")
plt.grid(True)
plt.legend()
plt.show()
```

Q3. Write a Python program to plot a Tanh Activation Function that is used in Neural Networks.

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

# Tanh activation function
def tanh(x):
   return np.tanh(x)

x = np.linspace(-10, 10, 100)
y = tanh(x)

plt.plot(x, y, label="Tanh", color='purple')
plt.title("Tanh Activation Function")
plt.xlabel("Input")
plt.ylabel("Output")
plt.grid(True)
plt.legend()
plt.show()
```

Q4. Write a Python program to plot a ReLU Activation Function that is used in Neural Networks.

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

# ReLU activation function
def relu(x):
   return np.maximum(0, x)

x = np.linspace(-10, 10, 100)
y = relu(x)

plt.plot(x, y, label="ReLU", color='green')
plt.title("ReLU Activation Function")
plt.xlabel("Input")
plt.ylabel("Output")
plt.grid(True)
plt.legend()
plt.show()
```

Q5. Write a Python program to plot a Step Activation Function that is used in Neural Networks.

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

# Step activation function
def step_function(x):
    return np.where(x >= 0, 1, 0)

x = np.linspace(-10, 10, 100)
y = step_function(x)

plt.plot(x, y, label="Step", color='red')
plt.title("Step Activation Function")
plt.xlabel("Input")
plt.ylabel("Output")
plt.grid(True)
plt.legend()
plt.show()
```

Q6. Generate AND Function using McCulloch–Pitts Neural Net by a Python Program.

```
# McCulloch-Pitts neuron for AND gate
def mcp_and(x1, x2):
    weights = [1, 1]
    threshold = 2
    net_input = x1*weights[0] + x2*weights[1]
    return 1 if net_input >= threshold else 0

# Test all combinations
print("AND Gate using McCulloch-Pitts Neuron:")
for x1 in [0, 1]:
    for x2 in [0, 1]:
        print(f"AND({x1}, {x2}) = {mcp_and(x1, x2)}")
```

Q7. Generate OR Function using McCulloch–Pitts Neural Net by a Python Program.

```
# McCulloch-Pitts neuron for OR gate
def mcp_or(x1, x2):
    weights = [1, 1]
    threshold = 1
    net_input = x1*weights[0] + x2*weights[1]
    return 1 if net_input >= threshold else 0
```

```
# Test all combinations
print("OR Gate using McCulloch-Pitts Neuron:")
for x1 in [0, 1]:
    for x2 in [0, 1]:
        print(f"OR({x1}, {x2}) = {mcp_or(x1, x2)}")
```

Q8. Generate NOT Function using McCulloch–Pitts Neural Net by a Python Program.

```
# McCulloch-Pitts neuron for NOT gate
def mcp_not(x):
    weight = -1
    threshold = 0
    net_input = x * weight
    return 1 if net_input >= threshold else 0

# Test both values
print("NOT Gate using McCulloch-Pitts Neuron:")
for x in [0, 1]:
    print(f"NOT({x}) = {mcp_not(x)}")
```

Q9. Generate AND-NOT Function using McCulloch–Pitts Neural Net by a Python Program.

```
# McCulloch-Pitts neuron for AND-NOT gate (A AND NOT B)
def mcp_andnot(x1, x2):
    weights = [1, -1]
    threshold = 1
    net_input = x1*weights[0] + x2*weights[1]
    return 1 if net_input >= threshold else 0

# Test all combinations
print("AND-NOT Gate using McCulloch-Pitts Neuron:")
for x1 in [0, 1]:
    for x2 in [0, 1]:
        print(f"ANDNOT({x1}, {x2}) = {mcp_andnot(x1, x2)}")
```

Q10. Generate NOR Function using McCulloch–Pitts Neural Net by a Python Program.

```
# McCulloch-Pitts neuron for NOR gate
def mcp_nor(x1, x2):
    weights = [-1, -1]
    threshold = -0.5
    net_input = x1*weights[0] + x2*weights[1]
    return 1 if net_input >= threshold else 0
```

```
# Test all combinations
print("NOR Gate using McCulloch-Pitts Neuron:")
for x1 in [0, 1]:
    for x2 in [0, 1]:
        print(f"NOR({x1}, {x2}) = {mcp_nor(x1, x2)}")
```

Q.11 Generate ORNOT function using McCulloch-Pitts neural net by a python program

```
# ORNOT: A OR (NOT B)
def mcp_ornot(a, b):
    # weights [1, -1], threshold 0 gives A OR (NOT B)
    w1, w2 = 1, -1
    threshold = 0.0
    net = a*w1 + b*w2
    return 1 if net >= threshold else 0

print("A B -> ORNOT")
for a in [0,1]:
    for b in [0,1]:
        print(f"{a} {b} -> {mcp_ornot(a,b)}")
```

Q.12 Generate NAND function using McCulloch-Pitts neural net by a python program

```
# NAND gate using McCulloch-Pitts neuron
def mcp_nand(a, b):
    # choose weights -1, -1 and threshold -1.5 to get NAND
    w1, w2 = -1, -1
    threshold = -1.5
    net = a*w1 + b*w2
    return 1 if net >= threshold else 0

print("A B -> NAND")
for a in [0,1]:
    for b in [0,1]:
        print(f"{a} {b} -> {mcp_nand(a,b)}")
```

Q.13 Write a Python Program using Perceptron Neural Network to recognize even and odd numbers. Given numbers are in ASCII from 0 to 9.

```
import numpy as np

# Prepare ASCII 7-bit binary inputs for '0'..'9'
def ascii_bits(ch, bits=7):
    val = ord(ch)
    arr = [(val >> i) & 1 for i in range(bits-1, -1, -1)]
    return np.array(arr, dtype=float)

X = np.array([ascii_bits(chr(ord('0')+i)) for i in range(10)])  # shape (10,7)
y = np.array([1 if i % 2 == 0 else 0 for i in range(10)])       # even -> 1, odd -> 0

# Perceptron training (binary labels 0/1)
np.random.seed(1)
w = np.random.randn(X.shape[1]) * 0.1
b = 0.0
lr = 0.1
epochs = 100

for epoch in range(epochs):
    errors = 0
    for xi, target in zip(X, y):
        activation = np.dot(w, xi) + b
        out = 1 if activation >= 0 else 0
        err = target - out
        if err != 0:
            w += lr * err * xi
            b += lr * err
            errors += 1
    if errors == 0:
        break

print("Trained weights:", w, "bias:", b)
print("Results (char, ascii bits, target, pred):")
for i, xi in enumerate(X):
    pred = 1 if np.dot(w, xi) + b >= 0 else 0
    print(chr(ord('0')+i), xi.astype(int).tolist(), y[i], pred)
```

Q.14 Write a python program to Implement a single-layer perceptron to classify binary patterns.

```
import numpy as np

# All 2-bit binary patterns
X = np.array([[0,0],[0,1],[1,0],[1,1]], dtype=float)
# Label: 1 if first bit is 1, else 0
y = X[:,0].astype(int)

# Perceptron training
w = np.zeros(X.shape[1])
b = 0.0
lr = 0.2
for epoch in range(20):
    for xi, target in zip(X,y):
        out = 1 if (np.dot(w, xi) + b) >= 0 else 0
        err = target - out
        if err != 0:
            w += lr * err * xi
            b += lr * err

print("weights:", w, "bias:", b)
for xi, target in zip(X,y):
    print(xi, "->", 1 if np.dot(w, xi)+b >= 0 else 0, " (true:", target, ")")
```

Q.15 Write python program to Visualize decision boundaries of a perceptron for 2D inputs.

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

# Simple linearly separable dataset (2D)
X = np.array([[0,0],[0,1],[1,0],[1,1],[2,0],[2,1]])
y = np.array([0,0,0,1,1,1])  # simple separable labels

# Train perceptron
w = np.zeros(2); b = 0.0; lr = 0.1
for epoch in range(50):
    for xi, target in zip(X,y):
        out = 1 if (np.dot(w, xi) + b) >= 0 else 0
        err = target - out
        if err != 0:
            w += lr * err * xi
            b += lr * err

# Plot
```

```
xx = np.linspace(np.min(X[:,0])-0.5, np.max(X[:,0])+0.5, 200)
# decision boundary: w0*x + w1*y + b = 0 -> y = (-w0*x - b) / w1  (if w1 !=0)
plt.figure()
for label in np.unique(y):
    pts = X[y==label]
    plt.scatter(pts[:,0], pts[:,1], label=f"class {label}")
if abs(w[1]) > 1e-6:
    yy = (-w[0]*xx - b) / w[1]
    plt.plot(xx, yy, label="decision boundary")
plt.legend(); plt.xlabel("x0"); plt.ylabel("x1"); plt.title("Perceptron Decision Boundary")
plt.grid(True); plt.show()
```

Q.16 Write a python program to design a Hopfield Network which stores 4 vectors.

```
# Define 4 bipolar patterns (length 8 for example)
patterns = np.array([
    [1, 1, 1, 1, -1, -1, -1, -1],
    [1, -1, 1, -1, 1, -1, 1, -1],
    [1, 1, -1, -1, 1, 1, -1, -1],
    [1, -1, -1, 1, 1, -1, -1, 1],
], dtype=int)

n = patterns.shape[1]
W = np.zeros((n,n), dtype=float)

# Hebbian learning (no self-weights)
for p in patterns:
    W += np.outer(p, p)
np.fill_diagonal(W, 0)

def recall(x, steps=10):
    s = x.copy()
    for _ in range(steps):
        for i in range(n):
            net = np.dot(W[i], s)
            s[i] = 1 if net >= 0 else -1
    return s

# test recall from noisy version of pattern 0
test = patterns[0].copy()
# flip a couple bits
test[0] *= -1; test[3] *= -1
print("noisy:", test)
print("recalled:", recall(test))
```

```
print("original:", patterns[0])
```

Q.17 Write a python Program for Bidirectional Associative Memory (BAM) with two pairs of vectors.

```
import numpy as np

# Example pairs (bipolar)
A1 = np.array([1, -1, 1])
B1 = np.array([1, 1, -1, -1])

A2 = np.array([-1, 1, -1])
B2 = np.array([-1, -1, 1, 1])

# BAM weight matrix (A->B)
W = np.outer(A1, B1) + np.outer(A2, B2)  # shape (len(A), len(B))

def recall_A_to_B(a, steps=5):
    b = np.sign(a @ W)
    b[b==0] = 1
    return b

def recall_B_to_A(b, steps=5):
    a = np.sign(W @ b)
    a[a==0] = 1
    return a

print("Recall A1->B:", recall_A_to_B(A1))
print("Recall B1->A:", recall_B_to_A(B1))
# try noisy A1
noisyA = A1.copy(); noisyA[0] *= -1
print("Noisy A->B:", recall_A_to_B(noisyA))
```

Q.18 Write a python program to show Back Propagation Network for XOR function with Binary Input and Output.

```
import numpy as np

# XOR dataset
X = np.array([[0,0],[0,1],[1,0],[1,1]], dtype=float)
y = np.array([[0],[1],[1],[0]], dtype=float)

# sigmoid and derivative
```

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

# network sizes
np.random.seed(0)
W1 = np.random.randn(2,2) * 0.5
b1 = np.zeros((1,2))
W2 = np.random.randn(2,1) * 0.5
b2 = np.zeros((1,1))

lr = 0.5
for epoch in range(10000):
    # forward
    z1 = X.dot(W1) + b1
    a1 = sigmoid(z1)
    z2 = a1.dot(W2) + b2
    a2 = sigmoid(z2)
    # loss (MSE)
    loss = np.mean((y - a2)**2)
    # backprop
    d2 = (a2 - y) * dsig(a2)
    dW2 = a1.T.dot(d2)
    db2 = np.sum(d2, axis=0, keepdims=True)
    d1 = d2.dot(W2.T) * dsig(a1)
    dW1 = X.T.dot(d1)
    db1 = np.sum(d1, axis=0, keepdims=True)
    # update
    W2 -= lr * dW2
    b2 -= lr * db2
    W1 -= lr * dW1
    b1 -= lr * db1

print("Predictions after training:")
for xi in X:
    a1 = sigmoid(xi.dot(W1) + b1)
    a2 = sigmoid(a1.dot(W2) + b2)
    print(xi, np.round(a2.ravel()))
```

Q.19 Write a python program to Implement a simple feedforward network with 1 hidden layer.

```
import numpy as np
```

```
def relu(x): return np.maximum(0, x)

class SimpleFFN:
    def __init__(self, in_dim, hid_dim, out_dim):
        np.random.seed(1)
        self.W1 = np.random.randn(in_dim, hid_dim) * 0.1
        self.b1 = np.zeros(hid_dim)
        self.W2 = np.random.randn(hid_dim, out_dim) * 0.1
        self.b2 = np.zeros(out_dim)
    def forward(self, x):
        h = relu(x.dot(self.W1) + self.b1)
        out = h.dot(self.W2) + self.b2
        return out, h

# example usage
net = SimpleFFN(3, 5, 2)
x = np.array([[0.1, 0.2, 0.3]])
y, h = net.forward(x)
print("output:", y)
```

Q.20 Implement Artificial Neural Network training process in Python by using Forward Propagation and Back Propagation.

```
import numpy as np

# Generate toy data: input 2D -> target scalar (sum)
X = np.random.randn(200,2)
y = (X[:,0] + X[:,1]).reshape(-1,1)

# simple 2-4-1 network with sigmoid hidden and linear out
def sigmoid(x): return 1/(1+np.exp(-x))
def dsig(x): return x*(1-x)

np.random.seed(2)
W1 = np.random.randn(2,4)*0.1; b1 = np.zeros((1,4))
W2 = np.random.randn(4,1)*0.1; b2 = np.zeros((1,1))
lr = 0.1
for epoch in range(500):
    # forward
    z1 = X.dot(W1) + b1
    a1 = sigmoid(z1)
    z2 = a1.dot(W2) + b2
    a2 = z2  # linear output
    # loss
```

```
    loss = np.mean((a2 - y)**2)
    # backprop
    d2 = 2*(a2 - y)/X.shape[0]  # dLoss/dz2
    dW2 = a1.T.dot(d2)
    db2 = np.sum(d2, axis=0, keepdims=True)
    d1 = d2.dot(W2.T) * dsig(a1)
    dW1 = X.T.dot(d1)
    db1 = np.sum(d1, axis=0, keepdims=True)
    # update
    W2 -= lr * dW2; b2 -= lr * db2
    W1 -= lr * dW1; b1 -= lr * db1

print("Final loss:", loss)
```

Q.21 Write a python program which will Plot training loss and accuracy curves.

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

# simple binary classification dataset (linearly separable)
X = np.vstack([np.random.randn(100,2) + [2,2], np.random.randn(100,2) + [-2,-2]])
y = np.array([1]*100 + [0]*100).reshape(-1,1)

# small logistic regression model
def sigmoid(x): return 1/(1+np.exp(-x))
W = np.zeros((2,1)); b = 0.0
lr = 0.1
losses = []
accs = []
for epoch in range(200):
    z = X.dot(W) + b
    a = sigmoid(z)
    loss = np.mean(-(y*np.log(a+1e-9) + (1-y)*np.log(1-a+1e-9)))
    losses.append(loss)
    preds = (a >= 0.5).astype(int)
    accs.append(np.mean(preds == y))
    dz = a - y
    dW = X.T.dot(dz) / X.shape[0]
    db = np.sum(dz) / X.shape[0]
    W -= lr * dW; b -= lr * db

plt.figure(figsize=(10,4))
plt.subplot(1,2,1); plt.plot(losses); plt.title("Loss"); plt.xlabel("Epoch")
plt.subplot(1,2,2); plt.plot(accs); plt.title("Accuracy"); plt.xlabel("Epoch")
```

```
plt.tight_layout(); plt.show()
```

Q.22 Write a python program to illustrate ART neural network.

```
import numpy as np

class ART1:
    def __init__(self, num_features, rho=0.8):
        self.rho = rho
        self.weights = []  # list of prototypes (binary)
    def train(self, X):
        for x in X:
            matched = False
            for i, w in enumerate(self.weights):
                # match score = |x & w| / |x|
                intersection = np.sum(x & w)
                match = intersection / (np.sum(x) + 1e-9)
                if match >= self.rho:
                    # update prototype: w <- x & w (or some learning); here we take union-ish
                    self.weights[i] = (w & x)  # conservative update
                    matched = True
                    break
            if not matched:
                self.weights.append(x.copy())
    def predict(self, x):
        for i, w in enumerate(self.weights):
            if np.sum(x & w) / (np.sum(x)+1e-9) >= self.rho:
                return i
        return None

# Example binary inputs
X = np.array([
    [1,1,0,0,1],
    [1,1,0,0,1],
    [0,0,1,1,0],
    [0,0,1,1,0],
], dtype=int)

art = ART1(num_features=5, rho=0.8)
art.train(X)
print("Prototypes:", art.weights)
for x in X:
    print(x, "-> cluster", art.predict(x))
```

Q.23 Write python code for Handwritten digit recognition with one-hot encoding.

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.neural_network import MLPClassifier
import numpy as np

digits = load_digits()
X = digits.data  # 8x8 images flattened
y = digits.target.reshape(-1,1)

# one-hot encode labels
enc = OneHotEncoder(sparse=False)
y_onehot = enc.fit_transform(y)

# train/test split and scaling
X_train, X_test, y_train, y_test = train_test_split(X, y_onehot, test_size=0.3,
random_state=0)
scaler = StandardScaler().fit(X_train)
X_train_s = scaler.transform(X_train)
X_test_s = scaler.transform(X_test)

# Use sklearn's MLP (handles softmax internally when using multi_class)
clf = MLPClassifier(hidden_layer_sizes=(64,), max_iter=200, random_state=1)
# sklearn expects integer labels for training; derive them
y_train_labels = np.argmax(y_train, axis=1)
clf.fit(X_train_s, y_train_labels)
acc = clf.score(X_test_s, np.argmax(y_test, axis=1))
print("Test accuracy:", acc)
```

Q.24 Write python program to find $f(x) = x^2$ by using simple feed forward neural network.

```
import numpy as np

# Data
X = np.linspace(-1,1,200).reshape(-1,1)
y = (X**2)

# simple 1-hidden-layer network
def relu(x): return np.maximum(0,x)
def d_relu(x): return (x>0).astype(float)
```

```
np.random.seed(0)
W1 = np.random.randn(1,10)*0.5; b1 = np.zeros((1,10))
W2 = np.random.randn(10,1)*0.5; b2 = np.zeros((1,1))
lr = 0.01
for epoch in range(3000):
   z1 = X.dot(W1) + b1
   a1 = relu(z1)
   out = a1.dot(W2) + b2
   loss = np.mean((out - y)**2)
   # backprop
   d_out = 2*(out - y)/X.shape[0]
   dW2 = a1.T.dot(d_out)
   db2 = np.sum(d_out, axis=0, keepdims=True)
   d1 = d_out.dot(W2.T) * d_relu(z1)
   dW1 = X.T.dot(d1)
   db1 = np.sum(d1, axis=0, keepdims=True)
   W2 -= lr * dW2; b2 -= lr * db2
   W1 -= lr * dW1; b1 -= lr * db1

# test
xs = np.array([[-0.8], [0.0], [0.5]])
for xi in xs:
   h = relu(xi.dot(W1) + b1)
   pred = h.dot(W2) + b2
   print(xi[0], "pred:", pred[0,0], "true:", xi[0]**2)
```

Q.25 Write a Python program to visualize the weights and biases of a trained feedforward neural network.

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

# Example trained weights (from the previous Q24 network if available).
# For demonstration, create random example weights of sensible shapes:
W1 = np.random.randn(1,10)
b1 = np.random.randn(10)
W2 = np.random.randn(10,1)
b2 = np.random.randn(1)
```

```
plt.figure(figsize=(8,4))
plt.subplot(1,3,1)
plt.imshow(W1, aspect='auto'); plt.colorbar(); plt.title("W1 (input->hidden)")
plt.subplot(1,3,2)
plt.imshow(W2, aspect='auto'); plt.colorbar(); plt.title("W2 (hidden->out)")
plt.subplot(1,3,3)
plt.bar(range(len(b1)), b1); plt.title("b1 (hidden biases)")
plt.tight_layout(); plt.show()
```
