Ex.No: 2 Solve XOR problem using Multi-Layer Perceptron

```
from sklearn.neural_network import MLPClassifier

# Data
X = [[0,0],[0,1],[1,0],[1,1]]
y = [0,1,1,0]

# Define MLP
mlp = MLPClassifier(hidden_layer_sizes=(2,), max_iter=10000,
learning_rate_init=0.1, random_state=42)

# Train
mlp.fit(X, y)

# Predict
print("Predictions:", mlp.predict(X))
print("Probabilities:\n", mlp.predict_proba(X))
```

Output:

Predictions: [1 1 1 1]

Probabilities:

[[0.49131601 0.50868399]

[0.49131601 0.50868399]

[0.49131601 0.50868399]

[0.49131601 0.50868399]]

Ex.No: 3 Implement Stochastic Gradient Descent Algorithm

```
import numpy as np

# Example dataset: y = 2x + 3
X = np.array([1, 2, 3, 4, 5], dtype=float)
y = np.array([5, 7, 9, 11, 13], dtype=float)

# Initialize parameters
w = np.random.randn()  # weight
b = np.random.randn()  # bias

# Learning rate
lr = 0.01

# Number of epochs
epochs = 100

# SGD training
for epoch in range(epochs):
    for i in range(len(X)):
```

```
        xi = X[i]
        yi = y[i]

        # Prediction
        y_pred = w * xi + b

        # Error
        error = yi - y_pred

        # Gradients
        dw = -2 * xi * error
        db = -2 * error

        # Update parameters
        w -= lr * dw
        b -= lr * db

    if epoch % 10 == 0:
        loss = np.mean((y - (w * X + b))**2)
        print(f"Epoch {epoch}, Loss: {loss:.4f}, w: {w:.3f}, b: {b:.3f}")

print("\nFinal model: y =", round(w, 2), "x +", round(b, 2))
```

Output:

Epoch 0, Loss: 8.0138, w: 1.102, b: 3.163

Epoch 10, Loss: 0.0229, w: 1.920, b: 3.341

Epoch 20, Loss: 0.0159, w: 1.933, b: 3.285

Epoch 30, Loss: 0.0111, w: 1.944, b: 3.238

Epoch 40, Loss: 0.0077, w: 1.953, b: 3.198

Epoch 50, Loss: 0.0054, w: 1.961, b: 3.166

Epoch 60, Loss: 0.0038, w: 1.967, b: 3.138

Epoch 70, Loss: 0.0026, w: 1.973, b: 3.115

Epoch 80, Loss: 0.0018, w: 1.977, b: 3.096

Epoch 90, Loss: 0.0013, w: 1.981, b: 3.080

Final model: y = 1.98 x + 3.07

Ex.No: 4 Implement Gradient Descent with AdaGrad

```
import numpy as np

# Example dataset: y = 2x + 3
X = np.array([1, 2, 3, 4, 5], dtype=float)
y = np.array([5, 7, 9, 11, 13], dtype=float)

# Initialize parameters
w = np.random.randn()
b = np.random.randn()

# Learning rate
lr = 0.1
epochs = 100

# For AdaGrad: accumulators (squared gradients)
eps = 1e-8  # to avoid division by zero
Gw, Gb = 0, 0

# Training loop
for epoch in range(epochs):
```

```
    # Predictions
    y_pred = w * X + b

    # Gradients (MSE loss)
    dw = -2 * np.sum(X * (y - y_pred)) / len(X)
    db = -2 * np.sum(y - y_pred) / len(X)

    # Accumulate squared gradients
    Gw += dw**2
    Gb += db**2

    # Update parameters using AdaGrad rule
    w -= (lr / (np.sqrt(Gw) + eps)) * dw
    b -= (lr / (np.sqrt(Gb) + eps)) * db

    if epoch % 10 == 0:
        loss = np.mean((y - y_pred)**2)
        print(f"Epoch {epoch}, Loss: {loss:.4f}, w: {w:.3f}, b: {b:.3f}")

print("\nFinal model: y =", round(w, 2), "x +", round(b, 2))
```

OUTPUT:

Epoch 0, Loss: 7.1346, w: 2.032, b: 0.634

Epoch 10, Loss: 1.3459, w: 2.345, b: 0.968

Epoch 20, Loss: 0.7498, w: 2.439, b: 1.099

Epoch 30, Loss: 0.6178, w: 2.472, b: 1.177

Epoch 40, Loss: 0.5708, w: 2.480, b: 1.233

Epoch 50, Loss: 0.5413, w: 2.478, b: 1.280

Epoch 60, Loss: 0.5163, w: 2.471, b: 1.322

Epoch 70, Loss: 0.4931, w: 2.462, b: 1.361

Epoch 80, Loss: 0.4712, w: 2.452, b: 1.398

Epoch 90, Loss: 0.4505, w: 2.442, b: 1.434

Final model: y = 2.43 x + 1.47

Ex.No: 1 Implement a simple feed-forward neural network

```
import numpy as np


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


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


def calculate_accuracy(predictions, targets):
    predicted_labels = np.round(predictions)
    correct = np.sum(predicted_labels == targets)
    return correct / len(targets)


X = np.array([[0,0],
              [0,1],
              [1,0],
              [1,1]])


y = np.array([[0],
              [1],
```

```
        [1],
        [0]])

input_layer = 2
hidden_layer = 2
output_layer = 1

np.random.seed(42)
weights_input_hidden = np.random.uniform(-1, 1, (input_layer,
hidden_layer))
bias_hidden = np.random.uniform(-1, 1, hidden_layer)
weights_hidden_output = np.random.uniform(-1, 1, (hidden_layer,
output_layer))
bias_output = np.random.uniform(-1, 1, output_layer)

learning_rate = 0.1
epochs = 10000

for epoch in range(epochs):
    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
    output = sigmoid(final_input)

    error = y - output
    d_output = error * sigmoid_derivative(output)

    error_hidden = np.dot(d_output, weights_hidden_output.T)
    d_hidden = error_hidden * sigmoid_derivative(hidden_output)

    weights_hidden_output += np.dot(hidden_output.T, d_output) *
learning_rate
    bias_output += np.sum(d_output, axis=0) * learning_rate

    weights_input_hidden += np.dot(X.T, d_hidden) * learning_rate
    bias_hidden += np.sum(d_hidden, axis=0) * learning_rate

    if (epoch+1) % 1000 == 0:
        acc = calculate_accuracy(output, y)
        print(f"Epoch {epoch+1} - Accuracy: {acc*100:.2f}%")

print("\nFinal Output:")
print(np.round(output))
```

OUTPUT:

```
Epoch 1000 - Accuracy: 50.00%
Epoch 2000 - Accuracy: 50.00%
Epoch 3000 - Accuracy: 75.00%
Epoch 4000 - Accuracy: 100.00%
Epoch 5000 - Accuracy: 100.00%
Epoch 6000 - Accuracy: 100.00%
Epoch 7000 - Accuracy: 100.00%
Epoch 8000 - Accuracy: 100.00%
Epoch 9000 - Accuracy: 100.00%
Epoch 10000 - Accuracy: 100.00%

Final Output:
[[0.]
 [1.]
 [1.]
 [0.]]
```