Task 1 — Single Layer Perceptron (from scratch)

Objective: Train a perceptron to classify points based on a linear boundary using Python from first principles.

a. Setup Python Environment

b. Define Perceptron Function

c. Generate Synthetic Training Data

d. Train with Perceptron Learning Rule

e. Visualize Decision Boundary

```
from sklearn.datasets import make_blobs

# Generate synthetic dataset
X, y = make_blobs(n_samples=100, n_features=2, centers=2, cluster_std=1.0, random_state=42)

# Print the first 5 rows of X and y
print("First 5 rows of X:")
print(X[:5])
print("\nFirst 5 rows of y:")
print(y[:5])
```

```
First 5 rows of X:
[[-2.98837186  8.82862715]
 [ 5.72293008  3.02697174]
 [-3.05358035  9.12520872]
 [ 5.461939    3.86996267]
 [ 4.86733877  3.28031244]]

First 5 rows of y:
[0 1 0 1 1]
```

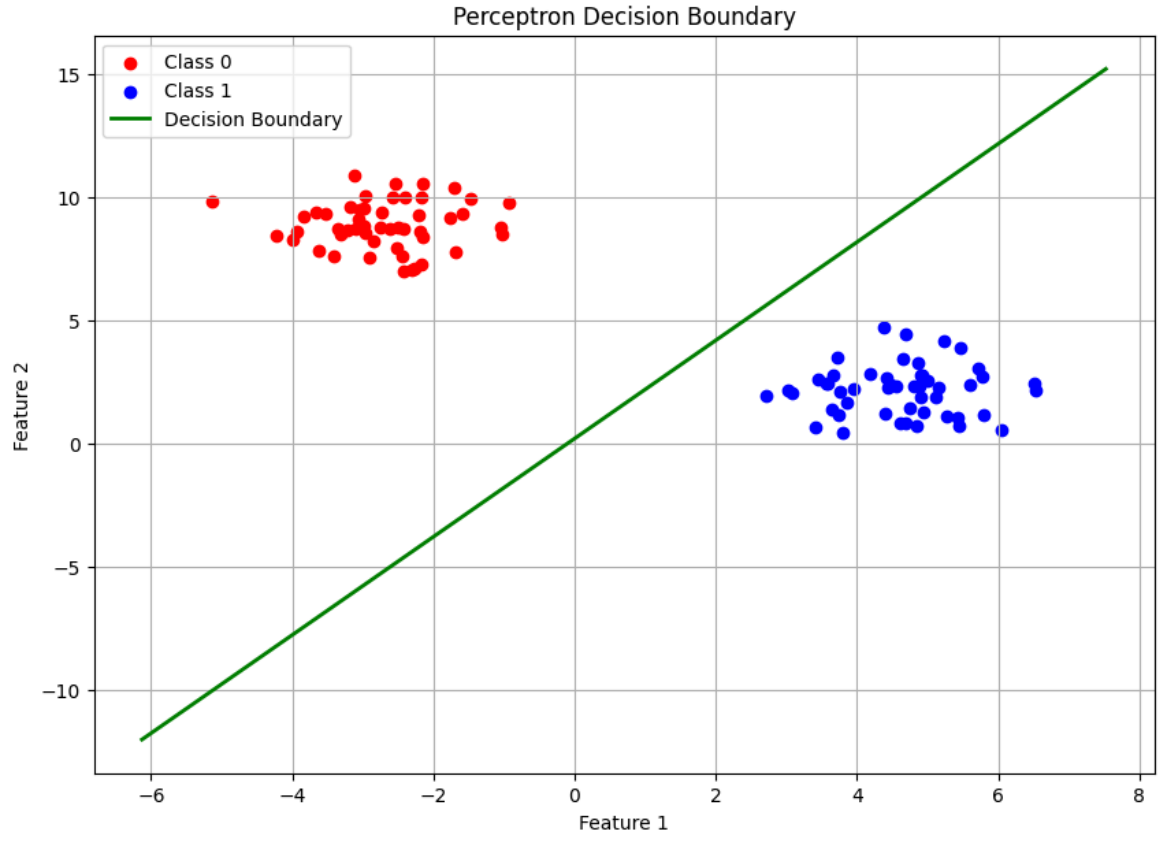
```
import numpy as np

# 1. Define relu function
def relu(x):
  return np.where(x >= 0, 1, -1)
```

```
# 2. Transform labels y from 0/1 to -1/1
y_transformed = np.where(y == 1, 1, -1)

# 3. Initialize weights and bias with small random values
n_features = X.shape[1]
weights = np.random.randn(n_features) * 0.01 # Small random weights
bias = np.random.randn() * 0.01 # Small random bias

# 4. Set learning rate and epochs
learning_rate = 0.01
n_epochs = 100

print(f"Initial Weights: {weights}, Bias: {bias}\n")

# 5. Implement the perceptron training loop
for epoch in range(n_epochs):
  for i in range(X.shape[0]):
    # Calculate net input
    net_input = np.dot(X[i], weights) + bias
    # Get prediction
    prediction = relu(net_input)

    # Update weights and bias if misclassified
    if prediction != y_transformed[i]:
      weights = weights + learning_rate * y_transformed[i] * X[i]
      bias = bias + learning_rate * y_transformed[i]

# 6. Print the final trained weights and bias
print(f"Final Weights: {weights}")
print(f"Final Bias: {bias}")
```

```
Initial Weights: [0.0114735  0.00857534], Bias: 0.010082427439460409

Final Weights: [ 0.09858652 -0.04944122]
Final Bias: 0.010082427439460409
```

```
import matplotlib.pyplot as plt
```

```
# Plotting the data points
plt.figure(figsize=(10, 7))
plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='red', label='Class 0')
plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='blue', label='Class 1')

# Define the range for the decision boundary line
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x_values = np.linspace(x_min, x_max, 100)

# Calculate the corresponding y-values for the decision boundary
# The decision boundary is defined by weights[0]*x + weights[1]*y + bias = 0
# So, y = (-weights[0]*x - bias) / weights[1]

# Handle the case where weights[1] might be zero to avoid division by zero
if weights[1] != 0:
    y_values = (-weights[0] * x_values - bias) / weights[1]
    plt.plot(x_values, y_values, color='green', linestyle='-', linewidth=2, label='Decision Boundary')
elif weights[0] != 0: # If weights[1] is zero, the boundary is a vertical line
    plt.axvline(x=-bias / weights[0], color='green', linestyle='-', linewidth=2, label='Decision Boundary')
else: # Both weights are zero, which means no meaningful boundary (or bias is zero, perfectly separated)
    print("Weights are too small or zero, cannot plot a meaningful decision boundary.")

# Add labels, title, and legend
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Perceptron Decision Boundary')
plt.legend()
plt.grid(True)
plt.show()
```



Task 2 — Multi-Layer Perceptron (MLP)

Objective: Build an MLP classifier using NumPy (from scratch) to learn non-linear decision boundaries. (For context: an MLP adds "hidden layers" and trains using backpropagation.

a. Define Network Architecture

For simplicity:

- Input layer: 2 neurons
- Hidden layer: 4 neurons (ReLU activation)
- Output layer: 1 neuron (sigmoid)

b. Define Activation Functions

c. Initialize Weights & Biases

d. Train with Backpropagation

e. Evaluate the Model

```
import numpy as np

# 1. Define the number of input, hidden, and output neurons
n_input = X.shape[1] # Number of features in X
n_hidden = 4
n_output = 1 # For binary classification

# 2. Initialize weights and biases for the hidden layer
np.random.seed(42) # for reproducibility
weights_input_hidden = np.random.randn(n_input, n_hidden) * 0.01
bias_hidden = np.random.randn(1, n_hidden) * 0.01

# 3. Initialize weights and biases for the output layer
weights_hidden_output = np.random.randn(n_hidden, n_output) * 0.01
bias_output = np.random.randn(1, n_output) * 0.01

# 4. Print the shapes of all initialized parameters
print(f"Shape of Input (X): {X.shape}")
print(f"Number of Input Neurons: {n_input}")
print(f"Number of Hidden Neurons: {n_hidden}")
print(f"Number of Output Neurons: {n_output}")
print(f"\nShape of weights_input_hidden: {weights_input_hidden.shape}")
print(f"Shape of bias_hidden: {bias_hidden.shape}")
print(f"Shape of weights_hidden_output: {weights_hidden_output.shape}")
print(f"Shape of bias_output: {bias_output.shape}")
```

```
Shape of Input (X): (100, 2)
Number of Input Neurons: 2
Number of Hidden Neurons: 4
Number of Output Neurons: 1

Shape of weights_input_hidden: (2, 4)
Shape of bias_hidden: (1, 4)
Shape of weights_hidden_output: (4, 1)
Shape of bias_output: (1, 1)
```

```
import numpy as np

# Redefine activation functions with fix for sigmoid input clipping
def relu(x):
  return np.maximum(0, x)

def relu_derivative(x):
  return (x > 0).astype(float)

def sigmoid(x):
  # Clip input to prevent overflow in exp when x is very negative or very positive
  x_clipped = np.clip(x, -500, 500) # Values beyond this range can cause exp to overflow
  return 1 / (1 + np.exp(-x_clipped))

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

# Redefine forward_propagation and backward_propagation
def forward_propagation(X, weights_input_hidden, bias_hidden, weights_hidden_output, bias_output):
  Z1 = np.dot(X, weights_input_hidden) + bias_hidden
  A1 = relu(Z1)
  Z2 = np.dot(A1, weights_hidden_output) + bias_output
  A2 = sigmoid(Z2)
  return Z1, A1, Z2, A2

def backward_propagation(X, y_binary, Z1, A1, Z2, A2, weights_hidden_output):
  m = X.shape[0]
  # Error at output layer, using y_binary (0/1) for cross-entropy compatible gradient
  dZ2 = A2 - y_binary.reshape(-1, 1)

  # Gradients for output layer parameters
  dW2 = (1/m) * np.dot(A1.T, dZ2)
  db2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)

  # Error at hidden layer
  dA1 = np.dot(dZ2, weights_hidden_output.T)
  dZ1 = dA1 * relu_derivative(Z1)

  # Gradients for hidden layer parameters
  dW1 = (1/m) * np.dot(X.T, dZ1)
```

```
  dW1 = (1/m) * np.dot(X.T, dZ1)
  db1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)

  return dW1, db1, dW2, db2

# Define learning rate and epochs
learning_rate = 0.001
n_epochs = 1000

print("Starting MLP training...")

# 4. Implement the training loop
for epoch in range(n_epochs):
  # Convert y_transformed (-1/1) to y_binary (0/1) for binary cross-entropy loss and dZ2
  y_binary = (y_transformed + 1) / 2

  # Forward propagation
  Z1, A1, Z2, A2 = forward_propagation(X, weights_input_hidden, bias_hidden, weights_hidden_output, bias_output)

  # Backward propagation (pass y_binary)
  dW1, db1, dW2, db2 = backward_propagation(X, y_binary, Z1, A1, Z2, A2, weights_hidden_output)

  # Update weights and biases
  weights_input_hidden = weights_input_hidden - learning_rate * dW1
  bias_hidden = bias_hidden - learning_rate * db1
  weights_hidden_output = weights_hidden_output - learning_rate * dW2
  bias_output = bias_output - learning_rate * db2

  # (Optional) Print loss every few epochs to monitor progress
  if epoch % 100 == 0:
    # Clip A2 values to prevent log(0) or log(1-1) errors in loss calculation
    A2_clipped = np.clip(A2, 1e-10, 1 - 1e-10)
    # Use y_binary for loss calculation
    loss = -np.mean(y_binary.reshape(-1, 1) * np.log(A2_clipped) + (1 - y_binary.reshape(-1, 1)) * np.log(1 - A2_clipped))
    print(f"Epoch {epoch}, Loss: {loss:.4f}")

print("MLP training complete.")
print(f"\nFinal weights_input_hidden:\n{weights_input_hidden}")
print(f"Final bias_hidden:\n{bias_hidden}")
print(f"Final weights_hidden_output:\n{weights_hidden_output}")
print(f"Final bias_output:\n{bias_output}")
```

```
Starting MLP training...
Epoch 0, Loss: 0.6930
Epoch 100, Loss: 0.6927
Epoch 200, Loss: 0.6923
Epoch 300, Loss: 0.6918
Epoch 400, Loss: 0.6909
Epoch 500, Loss: 0.6896
Epoch 600, Loss: 0.6877
Epoch 700, Loss: 0.6847
Epoch 800, Loss: 0.6802
Epoch 900, Loss: 0.6734
MLP training complete.

Final weights_input_hidden:
[[ 0.01081659 -0.00138264 -0.05584522  0.01333306]
 [ 0.00041411 -0.00234137  0.10343167  0.00940782]]
Final bias_hidden:
[[-0.00342538  0.0054256   0.00042998 -0.00465966]]
Final weights_hidden_output:
[[ 0.00908641]
 [-0.0191328 ]
 [-0.117477  ]
 [ 0.00266418]]
Final bias_output:
[[-0.00427076]]
```

```
from sklearn.preprocessing import StandardScaler

# Initialize StandardScaler
scaler = StandardScaler()

# Fit on X and transform X
X_scaled = scaler.fit_transform(X)

print("Input data X has been scaled (standardized).")
print(f"Mean of X_scaled: {X_scaled.mean(axis=0)}")
print(f"Standard deviation of X_scaled: {X_scaled.std(axis=0)}")

# Re-initialize weights and biases after scaling X, as they were trained on unscaled data
```

```
# and we are essentially restarting the training process with scaled data.
np.random.seed(42) # for reproducibility, ensuring new initialization is consistent
weights_input_hidden = np.random.randn(n_input, n_hidden) * 0.01
bias_hidden = np.random.randn(1, n_hidden) * 0.01
weights_hidden_output = np.random.randn(n_hidden, n_output) * 0.01
bias_output = np.random.randn(1, n_output) * 0.01

# Retrain with scaled data and re-initialized parameters
learning_rate = 0.001 # Keep the reduced learning rate
n_epochs = 1000 # Keep the number of epochs

print("Starting MLP training with scaled data and re-initialized parameters...")

for epoch in range(n_epochs):
  y_binary = (y_transformed + 1) / 2

  Z1, A1, Z2, A2 = forward_propagation(X_scaled, weights_input_hidden, bias_hidden, weights_hidden_output, bias_output)

  dW1, db1, dW2, db2 = backward_propagation(X_scaled, y_binary, Z1, A1, Z2, A2, weights_hidden_output)

  weights_input_hidden = weights_input_hidden - learning_rate * dW1
  bias_hidden = bias_hidden - learning_rate * db1
  weights_hidden_output = weights_hidden_output - learning_rate * dW2
  bias_output = bias_output - learning_rate * db2

  if epoch % 100 == 0:
    A2_clipped = np.clip(A2, 1e-10, 1 - 1e-10)
    loss = -np.mean(y_binary.reshape(-1, 1) * np.log(A2_clipped) + (1 - y_binary.reshape(-1, 1)) * np.log(1 - A2_clippe
    print(f"Epoch {epoch}, Loss: {loss:.4f}")

print("MLP training complete with scaled data.")
print(f"\nFinal weights_input_hidden:\n{weights_input_hidden}")
print(f"Final bias_hidden:\n{bias_hidden}")
print(f"Final weights_hidden_output:\n{weights_hidden_output}")
print(f"Final bias_output:\n{bias_output}")
```

```
Input data X has been scaled (standardized).
Mean of X_scaled: [-1.23234756e-16 -1.06692433e-15]
Standard deviation of X_scaled: [1. 1.]
Starting MLP training with scaled data and re-initialized parameters...
Epoch 0, Loss: 0.6932
Epoch 100, Loss: 0.6931
Epoch 200, Loss: 0.6931
Epoch 300, Loss: 0.6931
Epoch 400, Loss: 0.6931
Epoch 500, Loss: 0.6930
Epoch 600, Loss: 0.6930
Epoch 700, Loss: 0.6930
Epoch 800, Loss: 0.6930
Epoch 900, Loss: 0.6930
MLP training complete with scaled data.

Final weights_input_hidden:
[[ 0.00565131 -0.00816538  0.00225104  0.01430353]
 [-0.00301617  0.00444482  0.02004944  0.008413  ]]
Final bias_hidden:
[[-0.00399222  0.00831374 -0.00029576 -0.00550124]]
Final weights_hidden_output:
[[ 0.00326044]
 [-0.02201548]
 [-0.01979404]
 [-0.00496992]]
Final bias_output:
[[-0.00783364]]
```

```
print("Evaluating MLP model performance...")

# Make predictions using the trained model
_, _, _, A2_final = forward_propagation(X_scaled, weights_input_hidden, bias_hidden, weights_hidden_output, bias_output

# Convert probabilities to binary predictions (0 or 1)
predictions = (A2_final >= 0.5).astype(int)

# Convert y_transformed (-1/1) to y_binary (0/1) for accuracy comparison
y_binary = (y_transformed + 1) / 2

# Calculate accuracy
accuracy = np.mean(predictions == y_binary.reshape(-1, 1)) * 100

print(f"Model Accuracy on training data: {accuracy:.2f}%")
```

```
Evaluating MLP model performance...
Model Accuracy on training data: 50.00%
```

Task 3 — Perform following tasks:

1. Compare single layer vs MLP on linearly separable vs non-linear data (e.g., XOR).
2. Explore how hidden layer size impacts learning.
3. Plot loss over training epochs.

```
from sklearn.datasets import make_blobs
import numpy as np

# Re-define relu function from Task 1, as it's used by perceptron
def relu(x):
  return np.where(x >= 0, 1, -1)

def train_perceptron(X, y_binary, learning_rate, n_epochs):
    n_features = X.shape[1]
    weights = np.random.randn(n_features) * 0.01 # Initialize weights
    bias = np.random.randn() * 0.01 # Initialize bias

    # Transform y_binary (0/1) to -1/1 for perceptron update rule
    y_transformed_for_update = np.where(y_binary == 1, 1, -1)

    for epoch in range(n_epochs):
        for i in range(X.shape[0]):
            # Calculate net input
            net_input = np.dot(X[i], weights) + bias
            # Get prediction using the perceptron's activation (relu here)
            prediction = relu(net_input)

            # Update weights and bias if misclassified
            if prediction != y_transformed_for_update[i]:
                weights = weights + learning_rate * y_transformed_for_update[i] * X[i]
                bias = bias + learning_rate * y_transformed_for_update[i]
    return weights, bias

def perceptron_predict(X, weights, bias):
  linear_output = np.dot(X, weights) + bias
  # The perceptron output is 1 if >= 0, -1 otherwise. Convert to 0/1 for comparison.
  return np.where(linear_output >= 0, 1, 0)

# Linear dataset
X_lin, y_lin_orig = make_blobs(n_samples=200, centers=2, random_state=42)
y_lin = y_lin_orig.reshape(-1,1) # Keep original for print, flatten for training

# XOR dataset
X_xor = np.array([[0,0],[0,1],[1,0],[1,1]])
y_xor = np.array([[0],[1],[1],[0]]) # Keep original for print, flatten for training

# Perceptron for Linear data
lr_perc = 0.1
n_epochs_perc = 20
w_lin, b_lin = train_perceptron(X_lin, y_lin.flatten(), lr_perc, n_epochs_perc)

# Perceptron for XOR data
w_xor, b_xor = train_perceptron(X_xor, y_xor.flatten(), lr_perc, n_epochs_perc)

# Perceptron predictions
pred_lin_perc = perceptron_predict(X_lin, w_lin, b_lin)
pred_xor_perc = perceptron_predict(X_xor, w_xor, b_xor)

print("Perceptron - Linear Predictions vs True")
print(np.c_[pred_lin_perc[:10], y_lin.flatten()[:10]])

print("\nPerceptron - XOR Predictions vs True")
print(np.c_[pred_xor_perc, y_xor.flatten()])
```

```
Perceptron - Linear Predictions vs True
[[1 1]
 [1 1]
 [1 1]
 [0 0]
 [1 1]
 [1 1]
 [1 1]
 [1 1]
```

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

Perceptron - XOR Predictions vs True
[[1 0]
 [0 1]
 [0 1]
 [0 0]]
```

```
# MLP Training and Prediction Functions
def train_mlp(X, y, hidden_size=8, lr=0.05, epochs=20000, print_loss=True):
    np.random.seed(42)

    input_size = X.shape[1]
    output_size = 1
    # He initialization for ReLU/tanh (common practice)
    W1 = np.random.randn(input_size, hidden_size) * np.sqrt(2. / input_size)
    b1 = np.zeros((1, hidden_size))
    W2 = np.random.randn(hidden_size, output_size) * np.sqrt(2. / hidden_size)
    b2 = np.zeros((1, output_size))

    losses = []
    for epoch in range(epochs):
        # Forward
        Z1 = np.dot(X, W1) + b1
        A1 = np.tanh(Z1)  # Using tanh for hidden layer as in the original (truncated) code

        Z2 = np.dot(A1, W2) + b2
        A2 = 1 / (1 + np.exp(-np.clip(Z2, -500, 500)))    # Sigmoid with clipping for numerical stability

        # Binary Cross-Entropy Loss
        loss = -np.mean(y*np.log(A2+1e-8) + (1-y)*np.log(1-A2+1e-8))
        losses.append(loss)

        # Backprop
        dZ2 = A2 - y # Derivative of BCE loss with sigmoid
        dW2 = np.dot(A1.T, dZ2) / len(X)
        db2 = np.sum(dZ2, axis=0, keepdims=True) / len(X)

        dA1 = np.dot(dZ2, W2.T)
        dZ1 = dA1 * (1 - np.tanh(Z1)**2)   # tanh derivative
        dW1 = np.dot(X.T, dZ1) / len(X)
        db1 = np.sum(dZ1, axis=0, keepdims=True) / len(X)

        # Update
        W2 -= lr * dW2
        b2 -= lr * db2
        W1 -= lr * dW1
        b1 -= lr * db1

        if print_loss and epoch % 2000 == 0:
            print(f"Epoch {epoch}, Loss: {loss:.4f}")
    return W1, b1, W2, b2, losses

def predict_mlp(X, W1, b1, W2, b2):
    Z1 = np.dot(X, W1) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = 1 / (1 + np.exp(-np.clip(Z2, -500, 500))) # Sigmoid with clipping
    return np.where(A2 >= 0.5, 1, 0)

# Train MLP on XOR dataset
print("MLP Training on XOR (Non-Linear) Dataset")
W1_xor, b1_xor, W2_xor, b2_xor, losses_xor = train_mlp(X_xor, y_xor, hidden_size=8, epochs=20000, print_loss=True)
pred_xor_mlp = predict_mlp(X_xor, W1_xor, b1_xor, W2_xor, b2_xor)
print("\nMLP - XOR Predictions vs True")
print(np.c_[pred_xor_mlp, y_xor])

# Train MLP on Linear dataset
print("MLP Training on Linear Dataset")
W1_lin, b1_lin, W2_lin, b2_lin, losses_lin = train_mlp(X_lin, y_lin, hidden_size=8, epochs=20000, print_loss=True)
pred_lin_mlp = predict_mlp(X_lin, W1_lin, b1_lin, W2_lin, b2_lin)
print("\nMLP - Linear Predictions (first 10) vs True")
print(np.c_[pred_lin_mlp[:10], y_lin[:10]])

# Compare Accuracies
from sklearn.metrics import accuracy_score
print("\nCOMPARISON: Perceptron vs MLP")
print(f"\nLinear Dataset (Linearly Separable):")
print(f"  Perceptron Accuracy: {accuracy_score(y_lin.flatten(), pred_lin_perc) * 100:.2f}%")
```

```
print(f"  MLP Accuracy: {accuracy_score(y_lin.flatten(), pred_lin_mlp.flatten()) * 100:.2f}%")

print(f"\nXOR Dataset (Non-Linearly Separable):")
print(f"  Perceptron Accuracy: {accuracy_score(y_xor.flatten(), pred_xor_perc.flatten()) * 100:.2f}%")
print(f"  MLP Accuracy: {accuracy_score(y_xor.flatten(), pred_xor_mlp.flatten()) * 100:.2f}%")
```

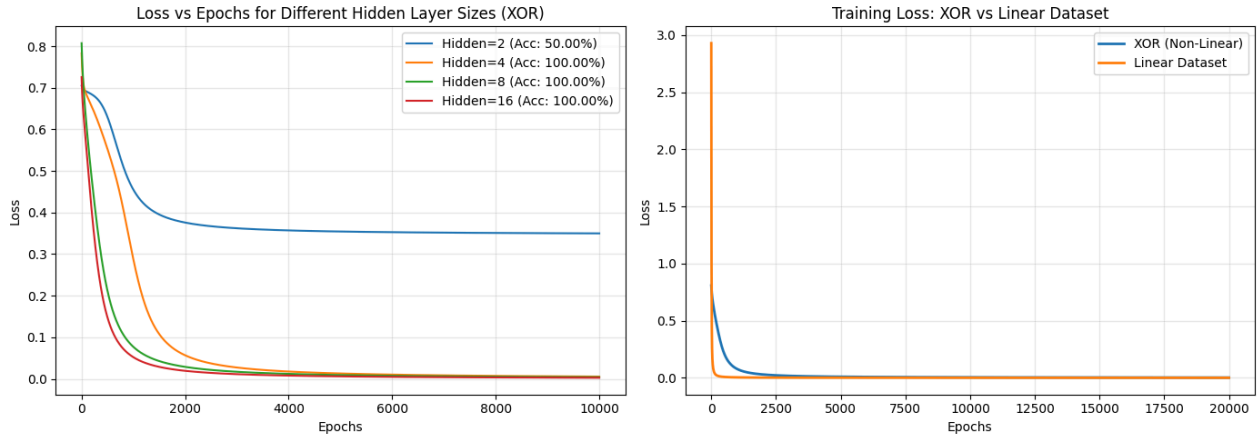
```
MLP Training on XOR (Non-Linear) Dataset
Epoch 0, Loss: 0.8074
Epoch 2000, Loss: 0.0288
Epoch 4000, Loss: 0.0119
Epoch 6000, Loss: 0.0074
Epoch 8000, Loss: 0.0053
Epoch 10000, Loss: 0.0041
Epoch 12000, Loss: 0.0034
Epoch 14000, Loss: 0.0029
Epoch 16000, Loss: 0.0025
Epoch 18000, Loss: 0.0022

MLP - XOR Predictions vs True
[[0 0]
 [1 1]
 [1 1]
 [0 0]]
MLP Training on Linear Dataset
Epoch 0, Loss: 2.9291
Epoch 2000, Loss: 0.0017
Epoch 4000, Loss: 0.0009
Epoch 6000, Loss: 0.0006
Epoch 8000, Loss: 0.0004
Epoch 10000, Loss: 0.0003
Epoch 12000, Loss: 0.0003
Epoch 14000, Loss: 0.0002
Epoch 16000, Loss: 0.0002
Epoch 18000, Loss: 0.0002

MLP - Linear Predictions (first 10) vs True
[[1 1]
 [1 1]
 [1 1]
 [0 0]
 [1 1]
 [1 1]
 [1 1]
 [1 1]
 [0 0]
 [1 1]]

COMPARISON: Perceptron vs MLP

Linear Dataset (Linearly Separable):
  Perceptron Accuracy: 100.00%
  MLP Accuracy: 100.00%

XOR Dataset (Non-Linearly Separable):
  Perceptron Accuracy: 25.00%
  MLP Accuracy: 100.00%
```

```
# Task 3.2: Explore impact of hidden layer size on learning
hidden_sizes = [2, 4, 8, 16]
results = []

for hs in hidden_sizes:
    W1, b1, W2, b2, losses = train_mlp(X_xor, y_xor, hidden_size=hs, epochs=10000, print_loss=False)
    predictions = predict_mlp(X_xor, W1, b1, W2, b2)
    acc = accuracy_score(y_xor.flatten(), predictions.flatten())
    final_loss = losses[-1]
    results.append((hs, acc, final_loss, losses))
    print(f"Hidden Size: {hs:2d} | Accuracy: {acc:.2%} | Final Loss: {final_loss:.4f}")

print("\nConclusion: MLPs with hidden sizes >= 4 can solve XOR problem effectively.")
```

```
Hidden Size:  2 | Accuracy: 50.00% | Final Loss: 0.3499
Hidden Size:  4 | Accuracy: 100.00% | Final Loss: 0.0055
Hidden Size:  8 | Accuracy: 100.00% | Final Loss: 0.0041
Hidden Size: 16 | Accuracy: 100.00% | Final Loss: 0.0026

Conclusion: MLPs with hidden sizes >= 4 can solve XOR problem effectively.
```

```
# Task 3.3: Plot loss over training epochs

# Plot loss curves for different hidden layer sizes
plt.figure(figsize=(14, 5))
```

```
# Plot 1: Loss for different hidden sizes on XOR
plt.subplot(1, 2, 1)
for hs, acc, final_loss, losses in results:
    plt.plot(losses, label=f'Hidden={hs} (Acc: {acc:.2%})')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss vs Epochs for Different Hidden Layer Sizes (XOR)')
plt.legend()
plt.grid(True, alpha=0.3)

# Plot 2: Compare XOR vs Linear dataset loss
plt.subplot(1, 2, 2)
plt.plot(losses_xor, label='XOR (Non-Linear)', linewidth=2)
plt.plot(losses_lin, label='Linear Dataset', linewidth=2)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss: XOR vs Linear Dataset')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\nKey Observations:")
print("1. Perceptron works well on linearly separable data but fails on XOR")
print("2. MLP can solve both linear and non-linear problems")
print("3. Hidden layer size >= 4 is sufficient for XOR")
print("4. Linear problems converge faster than non-linear problems")
```




```
Key Observations:
1. Perceptron works well on linearly separable data but fails on XOR
2. MLP can solve both linear and non-linear problems
3. Hidden layer size >= 4 is sufficient for XOR
4. Linear problems converge faster than non-linear problems
```

```
Start coding or generate with AI.
```