Exercise 4

In this exercise, you will implement forward and backward pass of a simple neural network. You are expected to write all the functions using vectorized numpy operations only.

The following cell has code to load the train and test data. You will be working with the MNIST dataset. The images have been flattened and normalised to be between 0 and 1 for you already.

In [2]:
```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

# Load MNIST dataset
def load_mnist():
    mnist = fetch_openml('mnist_784', version=1)
    X, y = mnist.data / 255.0, mnist.target.astype(int)
    return X, y.to_numpy()  # Convert y to a NumPy array

# One-hot encode labels
def one_hot_encode(y, num_classes):
    encoder = OneHotEncoder(sparse_output=False, categories=[range(num_cl
    return encoder.fit_transform(y.reshape(-1, 1))

# Split dataset
def prepare_data(test_size=0.2):
    X, y = load_mnist()
    y_encoded = one_hot_encode(y, num_classes=10)
    return train_test_split(X, y_encoded, test_size=test_size, random_sta

X_train, X_test, y_train, y_test = prepare_data()
X_train, X_test = X_train.to_numpy(), X_test.to_numpy()
print(f"Training Data Shape: {X_train.shape}, Test Data Shape: {X_test.sh
```

```
Training Data Shape: (56000, 784), Test Data Shape: (14000, 784)
```

You need to implement a two-layer neural network (one hidden layer) using NumPy. Fill in all the required cells below. Only use numpy functions.

- Implement the forward pass. Use ReLU activation for the hidden layer and softmax for the final output. Be sure to use the bias as well. (0.5 point)
- Implement the backward pass. This should return the gradients of the loss w.r.t the weights and biases of the network. The return signature of the backward pass is provided as a comment in the function. (1.5 points)
- For your loss function, use the cross-entropy loss. (0.5 point)
- The `predict` function should run the forward pass and return the predicted class. (0.5 point)
- The `train` function should run the forward pass, compute the loss and and the gradients, and update the parameters using gradient descent with the given

learning rate. It should repeat this for the given number of epochs. You are given some code to evaluate the performance of your network during training. You can uncomment it and match your variable names. (1 point)

In [3]:
```
class TwoLayerNN:
    def __init__(self, input_size, hidden_size, output_size):
        """
        Initialize weights and biases.
        """
        self.W1 = np.random.randn(input_size, hidden_size) * 0.01
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size) * 0.01
        self.b2 = np.zeros((1, output_size))

    def relu(self, Z):
        """
        ReLU activation function.
        """
        return np.maximum(0, Z)

    def relu_derivative(self, Z):
        """
        Derivative of ReLU activation.
        """
        return (Z > 0).astype(float)

    def softmax(self, Z):
        """
        Softmax activation function.
        """
        expZ = np.exp(Z - np.max(Z, axis=1, keepdims=True))
        return expZ / np.sum(expZ, axis=1, keepdims=True)

    def forward(self, X):
        """
        Forward pass.
        """
        # Compute Z1 and A1
        self.Z1 = X.dot(self.W1) + self.b1
        self.A1 = self.relu(self.Z1)

        # Compute Z2 and A2 (softmax output)
        self.Z2 = self.A1.dot(self.W2) + self.b2
        self.A2 = self.softmax(self.Z2)
        return self.A2

    def compute_loss(self, y_true, y_pred):
        """
        Compute cross-entropy loss.
        """
        m = y_true.shape[0]
        # Avoid log(0)
        log_likelihood = -np.log(y_pred[np.arange(m), np.argmax(y_true, a
        loss = np.sum(log_likelihood) / m
        return loss

    def backward(self, X, y, learning_rate):
        """
        Backpropagation to update weights.
        Returns:
            dW1, db1, dW2, db2
        """
        m = X.shape[0]

        # dZ2 = A2 - y
        dZ2 = self.A2 - y

        # dW2 = A1^T dZ2
        dW2 = (self.A1.T).dot(dZ2) / m
        db2 = np.sum(dZ2, axis=0, keepdims=True) / m

        # dA1 = dZ2 W2^T
        dA1 = dZ2.dot(self.W2.T)

        # dZ1 = dA1 * relu'(Z1)
        dZ1 = dA1 * self.relu_derivative(self.Z1)

        # dW1 = X^T dZ1
        dW1 = X.T.dot(dZ1) / m
        db1 = np.sum(dZ1, axis=0, keepdims=True) / m

        # Update parameters
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2

        return dW1, db1, dW2, db2

    def predict(self, X):
        """
        Predict class labels.
        """
        A2 = self.forward(X)
        return np.argmax(A2, axis=1)

    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            # Forward pass
            y_pred = self.forward(X)
            loss = self.compute_loss(y, y_pred)

            # Backward pass (and update parameters)
            self.backward(X, y, learning_rate)

            # Print loss every 10 epochs or on the last epoch
            if epoch % 10 == 0 or epoch == epochs - 1:
                print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}")
```

The following code evaluates the performance of your network on X_test. You can expect an accuracy of around 90%.

In [5]:
```
# Initialize model
input_size = X_train.shape[1]
hidden_size = 256  # You can choose a suitable value
output_size = 10  # Number of classes
```

```python
model = TwoLayerNN(input_size, hidden_size, output_size)

# Training the model
epochs = 100
learning_rate = 0.5
model.train(X_train, y_train, epochs, learning_rate)

# Evaluate on test data
predictions = model.predict(X_test)
# print(predictions.shape)
accuracy = np.mean(predictions == np.argmax(y_test, axis=1))
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Epoch 1/100, Loss: 2.3008
Epoch 11/100, Loss: 1.7980
Epoch 21/100, Loss: 0.8264
Epoch 31/100, Loss: 0.8380
Epoch 41/100, Loss: 0.5112
Epoch 51/100, Loss: 0.4713
Epoch 61/100, Loss: 0.4140
Epoch 71/100, Loss: 0.3930
Epoch 81/100, Loss: 0.3480
Epoch 91/100, Loss: 0.3319
Epoch 100/100, Loss: 0.3218
Test Accuracy: 90.88%
```