Some Helper Function

Softmax function:

```
import numpy as np
def softmax(z):
  """
  Compute the softmax probabilities for a given input matrix.
  Parameters:
  z (numpy.ndarray): Logits (raw scores) of shape (m, n), where

  - m is the number of samples.
  - n is the number of classes.

  Returns:
  numpy.ndarray: Softmax probability matrix of shape (m, n), where
  each row sums to 1 and represents the probability
  distribution over classes.

  Notes:
  - The input to softmax is typically computed as: z = XW + b.
  - Uses numerical stabilization by subtracting the max value per row.
  """
  # Your Code Here.
  z= np.atleast_2d(z)

  z_max = np.max(z, axis=1, keepdims=True)
  exp_z = np.exp(z-z_max)
  softmax_probs = exp_z / np.sum(exp_z,axis=1, keepdims=True)
  return softmax_probs
```

Softmax Test case

This test case checks that each row in the resulting softmax probalities sums to 1, which is the fundamental of property of softmax

```
def test_softmax():
  # Testc case setup
  z_test = np.array([[2.0,1.0,0.1],[1.0,1.0,1.0]])

  # Calculate softmax
  softmax_output = softmax(z_test)

  # Calculate rows sum
  row_sums = np.sum(softmax_output,axis=1)
  # Verify results
  try:
    np.allclose(row_sums,1)
    print(f"Test failed: Row sum are {row_sums}")
    print("Softmax Function passed the test case")
  except AssertionError as e:
    print((f"X Test failed: Row sums are {row_sums}"))
```

```
test_softmax()
```

```
Test failed: Row sum are [1. 1.]
Softmax Function passed the test case
```

```
def predict_softmax(X, W, b):
  """
  Predict the class labels for a set of samples using the trained softmax model.
  Parameters:
  X (numpy.ndarray): Feature matrix of shape (n, d), where n is the number of samples and d is the
  number of features.
  W (numpy.ndarray): Weight matrix of shape (d, c), where c is the number of classes.
  b (numpy.ndarray): Bias vector of shape (c,).
  Returns:
```

```
  numpy.ndarray: Predicted class labels of shape (n,), where each value is the index of the
  predicted class.
  """
  logist = np.dot(X,W)+b #Compute raw scores (logits)
  probabilities = softmax(logist)
  predicted_classes = np.argmax(probabilities,axis=1)
  return predicted_classes
```

Test Function for Prediction Function:

The test function ensures that the predicted class labels have the same number of elements as the input samples, verifying that the model produces a valid output shape.

```
# The test function ensures that the predicted class labels have the same number of elements as the
# input samples, verifying that the model produces a valid output shape.
# Define test case
X_test = np.array([[0.2, 0.8], [0.5, 0.5], [0.9, 0.1]]) # Feature matrix (3 samples, 2 features)
W_test = np.array([[0.4, 0.2, 0.1], [0.3, 0.7, 0.5]]) # Weights (2 features, 3 classes)
b_test = np.array([0.1, 0.2, 0.3]) # Bias (3 classes)
# Expected Output:
# The function should return an array with class labels (0, 1, or 2)
y_pred_test = predict_softmax(X_test, W_test, b_test)
# Validate output shape
assert y_pred_test.shape == (3,), f"Test failed: Expected shape (3,), got {y_pred_test.shape}"
# Print the predicted labels
print("Predicted class labels:", y_pred_test)
```

```
Predicted class labels: [1 1 0]
```

1. Implementation of Loss Function:

```
def loss_softmax(y_pred, y):
  """
  Compute the cross-entropy loss for a single sample.
  Parameters:
  y_pred (numpy.ndarray): Predicted probabilities of shape (c,) for a single sample,

  where c is the number of classes.

  y (numpy.ndarray): True labels (one-hot encoded) of shape (c,), where c is the number of classes.
  Returns:
  float: Cross-entropy loss for the given sample.
  """
  epsilon = 1e-12 #small constant to prevent log(0)
  y_pred = np.clip(y_pred,epsilon,1.0 - epsilon) # Clip values to prvent numerical instability
  loss = -np.sum(y*np.log(y_pred))
  return loss
```

Test case for Loss Function:

This test case Compares loss for correct vs. incorrect predictions.

- Expects low loss for correct predictions.
- Expects high loss for incorrect predictions.

```
import numpy as np
# This test case Compares loss for correct vs. incorrect predictions.
# Expects low loss for correct predictions.
# Expects high loss for incorrect predictions.
# Define correct predictions (low loss scenario)
y_true_correct = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]]) # True one-hot labels
y_pred_correct = np.array([[0.9, 0.05, 0.05],
[0.1, 0.85, 0.05],
[0.05, 0.1, 0.85]]) # High confidence in the correct class

# Define incorrect predictions (high loss scenario)
y_pred_incorrect = np.array([[0.05, 0.05, 0.9], # Highly confident in the wrong class

[0.1, 0.05, 0.85],
```

```
[0.85, 0.1, 0.05]])

# Compute loss for both cases
loss_correct = loss_softmax(y_pred_correct, y_true_correct)
loss_incorrect = loss_softmax(y_pred_incorrect, y_true_correct)
# Validate that incorrect predictions lead to a higher loss
assert loss_correct < loss_incorrect, f"Test failed: Expected loss_correct < loss_incorrect, but got{loss_correct:.4f} >= {loss_incorrect:.4
# Print results
print(f"Cross-Entropy Loss (Correct Predictions): {loss_correct:.4f}")
print(f"Cross-Entropy Loss (Incorrect Predictions): {loss_incorrect:.4f}")
```

```
Cross-Entropy Loss (Correct Predictions): 0.4304
Cross-Entropy Loss (Incorrect Predictions): 8.9872
```

2. Implementation of Cost Function:

```
def cost_softmax(X, y, W, b):
  """
  Compute the average softmax regression cost (cross-entropy loss) over all samples.
  Parameters:
  X (numpy.ndarray): Feature matrix of shape (n, d), where n is the number of samples and d is the
  number of features.
  y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c), where n is the number of
  samples and c is the number of classes.
  W (numpy.ndarray): Weight matrix of shape (d, c).
  b (numpy.ndarray): Bias vector of shape (c,).
  Returns:
  float: Average softmax cost (cross-entropy loss) over all samples.

  """
  # Compute logits (Linear Transformation)
  logits = np.dot(X,W)+b
  # Compute predicted probalities using softmax
  predictions = softmax(logits)
  # Calculate cross-entropy loss
  cost = loss_softmax(predictions,y)
  return cost
```

Test Case for Cost Function:

The test case assures that the cost for the incorrect prediction should be higher than for the correct prediction, confirming that the cost function behaves as expected.

```
import numpy as np

# Example 1: Correct Prediction (Closer predictions)
X_correct = np.array([[1.0, 0.0], [0.0, 1.0]])  # Feature matrix for correct predictions
y_correct = np.array([[1, 0], [0, 1]])  # True labels (one-hot encoded, matching predictions)
W_correct = np.array([[5.0, -2.0], [-3.0, 5.0]])  # Weights for correct prediction
b_correct = np.array([0.1, 0.1])  # Bias for correct prediction

# Example 2: Incorrect Prediction (Far off predictions)
X_incorrect = np.array([[0.1, 0.9], [0.8, 0.2]])  # Feature matrix for incorrect predictions
y_incorrect = np.array([[1, 0], [0, 1]])  # True labels (one-hot encoded, incorrect predictions)
W_incorrect = np.array([[0.1, 2.0], [1.5, 0.3]])  # Weights for incorrect prediction
b_incorrect = np.array([0.5, 0.6])  # Bias for incorrect prediction

# Compute cost for correct predictions
cost_correct = cost_softmax(X_correct, y_correct, W_correct, b_correct)

# Compute cost for incorrect predictions
cost_incorrect = cost_softmax(X_incorrect, y_incorrect, W_incorrect, b_incorrect)

# Check if the cost for incorrect predictions is greater than for correct predictions
assert cost_incorrect > cost_correct, f"Test failed: Incorrect cost {cost_incorrect} is not greater than correct cost {cost_correct}"

# Print the costs for verification
print("Cost for correct prediction:", cost_correct)
print("Cost for incorrect prediction:", cost_incorrect)

print("Test passed!")
```

```
Cost for correct prediction: 0.0012468728266698647
Cost for incorrect prediction: 0.5986172271889223
Test passed!
```

Implementation of Compute Gradients

```
def compute_gradient_softmax(X, y, W, b):
  """
  Compute the gradients of the cost function with respect to weights and biases.
  Parameters:
  X (numpy.ndarray): Feature matrix of shape (n, d).
  y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c).
  W (numpy.ndarray): Weight matrix of shape (d, c).
  b (numpy.ndarray): Bias vector of shape (c,).
  Returns:
  tuple: Gradients with respect to weights (d, c) and biases (c,).
  """
  n,d = X.shape
  z= np.dot(X,W)+b
  y_pred = softmax(z)
  grad_W = np.dot(X.T, (y_pred-y))/n # Calculate with respect to weights
  grad_b = np.sum(y_pred-y, axis=0) /n #Calculate with respect to biases
  return grad_W, grad_b
```

Test case for compute_gradient function:

The test checks if the gradients from the function are close enough to the manually computed gradients using np.allclose, which accounts for potential floating-point discrepancies.

```
import numpy as np

# Define a simple feature matrix and true labels
X_test = np.array([[0.2, 0.8], [0.5, 0.5], [0.9, 0.1]])  # Feature matrix (3 samples, 2 features)
y_test = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])  # True labels (one-hot encoded, 3 classes)

# Define weight matrix and bias vector
W_test = np.array([[0.4, 0.2, 0.1], [0.3, 0.7, 0.5]])  # Weights (2 features, 3 classes)
b_test = np.array([0.1, 0.2, 0.3])  # Bias (3 classes)

# Compute the gradients using the function
grad_W, grad_b = compute_gradient_softmax(X_test, y_test, W_test, b_test)
```

```
# Manually compute the predicted probabilities (using softmax function)
z_test = np.dot(X_test, W_test) + b_test
y_pred_test = softmax(z_test)

# Compute the manually computed gradients
grad_W_manual = np.dot(X_test.T, (y_pred_test - y_test)) / X_test.shape[0]
grad_b_manual = np.sum(y_pred_test - y_test, axis=0) / X_test.shape[0]

# Assert that the gradients computed by the function match the manually computed gradients
assert np.allclose(grad_W, grad_W_manual), f"Test failed: Gradients w.r.t. W are not equal.\nExpected: {grad_W_manual}\nGot: {grad_W}"
assert np.allclose(grad_b, grad_b_manual), f"Test failed: Gradients w.r.t. b are not equal.\nExpected: {grad_b_manual}\nGot: {grad_b}"

# Print the gradients for verification
print("Gradient w.r.t. W:", grad_W)
print("Gradient w.r.t. b:", grad_b)

print("Test passed!")
```

```
Gradient w.r.t. W: [[ 0.1031051   0.01805685 -0.12116196]
 [-0.13600547  0.00679023  0.12921524]]
Gradient w.r.t. b: [-0.03290036  0.02484708  0.00805328]
Test passed!
```

Implementing Gradient Descent:

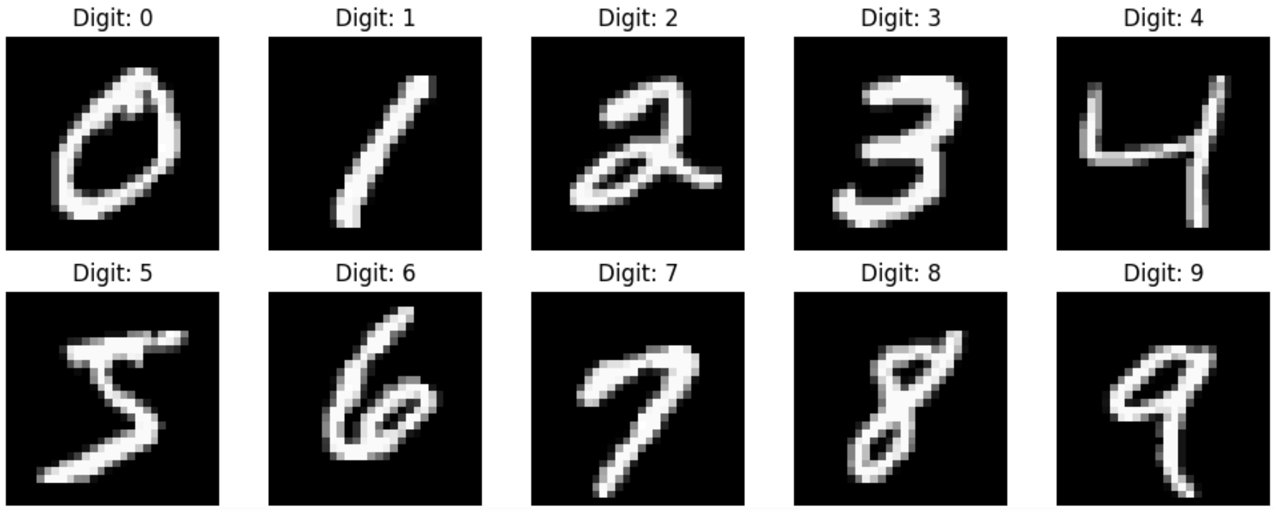
```
def gradient_descent_softmax(X,W,y,b,alpha,n_iter,show_cost=False):
    """
    Perform gradient descent to optimize the weights and biases.

    Parameters:
    X (numpy.ndarray): Feature matrix of shape (n, d).
    y (numpy.ndarray): True labels (one-hot encoded) of shape (n, c).
    W (numpy.ndarray): Weight matrix of shape (d, c).
    b (numpy.ndarray): Bias vector of shape (c,).
    alpha (float): Learning rate.
    n_iter (int): Number of iterations.
    show_cost (bool): Whether to display the cost at intervals.

    Returns:
    tuple: Optimized weights, biases, and cost history.
    """

    cost_history = []

    for i in range(n_iter):
      grad_W,grad_b = compute_gradient_softmax(X,y,W,b)

      # Updating Weights and biases using the gradients
      W = W-  alpha * grad_W
      b = b- alpha * grad_b

      # Compute and store cost
      cost = cost_softmax(X,y,W,b)
      cost_history.append(cost)

      if show_cost and (i%100==0 or i==n_iter-1):
        print(f"Iteration {i}: Cost = {cost:.6f}")

    return W,b, cost_history
```

Preparing Dataset:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split


def load_and_prepare_minst(csv_file,test_size=0.2,random_state=42):
    """
    Reads the MNIST CSV file, splits data into train/test sets, and plots one image per class.
```

```
    Arguments:
    csv_file (str)       : Path to the CSV file containing MNIST data.
    test_size (float)    : Proportion of the data to use as the test set (default: 0.2).
    random_state (int)   : Random seed for reproducibility (default: 42).

    Returns:
    X_train, X_test, y_train, y_test : Split dataset.
    """
    # Load dataset
    df = pd.read_csv(csv_file)

    # Separate labels and features
    y = df.iloc[:,0].values #First column is the label
    x = df.iloc[:,1:].values # Remaining columns are pixel values

    # Noramlize pixels values (optional but recommend)
    X = x/255.0

    # Split data into train and test sets

    X_train, X_test, y_train,y_test = train_test_split(X,y,test_size=test_size,random_state=random_state)

    # Plot one sample image per class
    plot_sample_images(X,y)

    return X_train,X_test,y_train,y_test


def plot_sample_images(X,y):
    """
    Plots one sample image for each digit class (0-9).

    Arguments:
    X (np.ndarray): Feature matrix containing pixel values.
    y (np.ndarray): Labels corresponding to images.
    """
    plt.figure(figsize=(10,4))
    unique_classes = np.unique(y) # Get unique class labels

    for i, digit in enumerate(unique_classes):
      index = np.where(y==digit)[0][0] # Find first occurance of the class
      image = X[index].reshape(28,28)  # Reshape 1D array to 28x28

      plt.subplot(2,5,i+1)
      plt.imshow(image,cmap='gray')
      plt.title(f'Digit: {digit}')
      plt.axis('off')

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

```
csv_file = '/content/drive/MyDrive/Artificial_Intelligence/Week2/Worksheet2/mnist_dataset.csv'
X_train,X_test,y_train,y_test = load_and_prepare_minst(csv_file)
```



A Quick debugging Step:

```
# Assert that X and y have matching lengths
assert len(X_train) == len(y_train), f"Error: X and y have different lengths! X={len(X_train)} , y={len(y_train)}"
print("Move forward: Dimension of Feture Matrix X and label vector y matched.")
```

```
Move forward: Dimension of Feture Matrix X and label vector y matched.
```

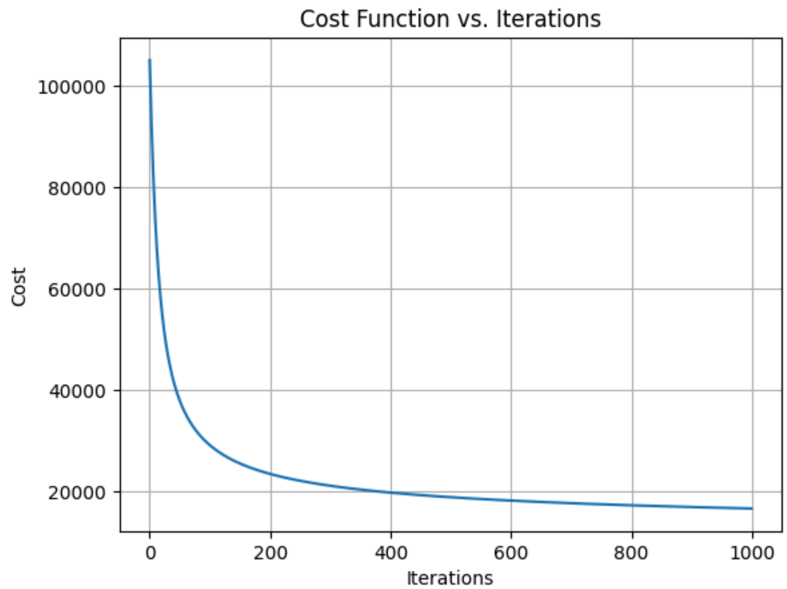
Train the Model:

```
print(f"Training data shape: {X_train.shape}")
print(f"Test data shape: {X_test.shape}")
```

```
Training data shape: (48000, 784)
Test data shape: (12000, 784)
```

```
from sklearn.preprocessing import OneHotEncoder

# Check if y_train is one-hot encoded
if len(y_train.shape) == 1:
    encoder = OneHotEncoder(sparse_output=False)  # Use sparse_output=False for newer versions of sklearn
    y_train = encoder.fit_transform(y_train.reshape(-1, 1))  # One-hot encode labels
    y_test = encoder.transform(y_test.reshape(-1, 1))  # One-hot encode test labels

# Now y_train is one-hot encoded, and we can proceed to use it
d = X_train.shape[1]  # Number of features (columns in X_train)
c = y_train.shape[1]  # Number of classes (columns in y_train after one-hot encoding)

# Initialize weights with small random values and biases with zeros
W = np.random.randn(d, c) * 0.01  # Small random weights initialized
b = np.zeros(c)  # Bias initialized to 0

# Set hyperparameters for gradient descent
alpha = 0.1  # Learning rate
n_iter = 1000  # Number of iterations to run gradient descent


# print(f'X shape: {X_train.shape}, W shape: {W.shape}, b shape: {b.shape}')

# Train the model using gradient descent
W_opt, b_opt, cost_history = gradient_descent_softmax(X_train, W, y_train, b, alpha, n_iter, show_cost=True)

# Plot the cost history to visualize the convergence
plt.plot(cost_history)
plt.title('Cost Function vs. Iterations')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.grid(True)
plt.show()
```

```
Iteration 0: Cost = 105034.594116
Iteration 100: Cost = 29162.110983
Iteration 200: Cost = 23508.684417
Iteration 300: Cost = 21173.055160
Iteration 400: Cost = 19823.680666
Iteration 500: Cost = 18917.027185
Iteration 600: Cost = 18252.888972
Iteration 700: Cost = 17738.443725
Iteration 800: Cost = 17324.089057
Iteration 900: Cost = 16980.610905
Iteration 999: Cost = 16692.231772
```

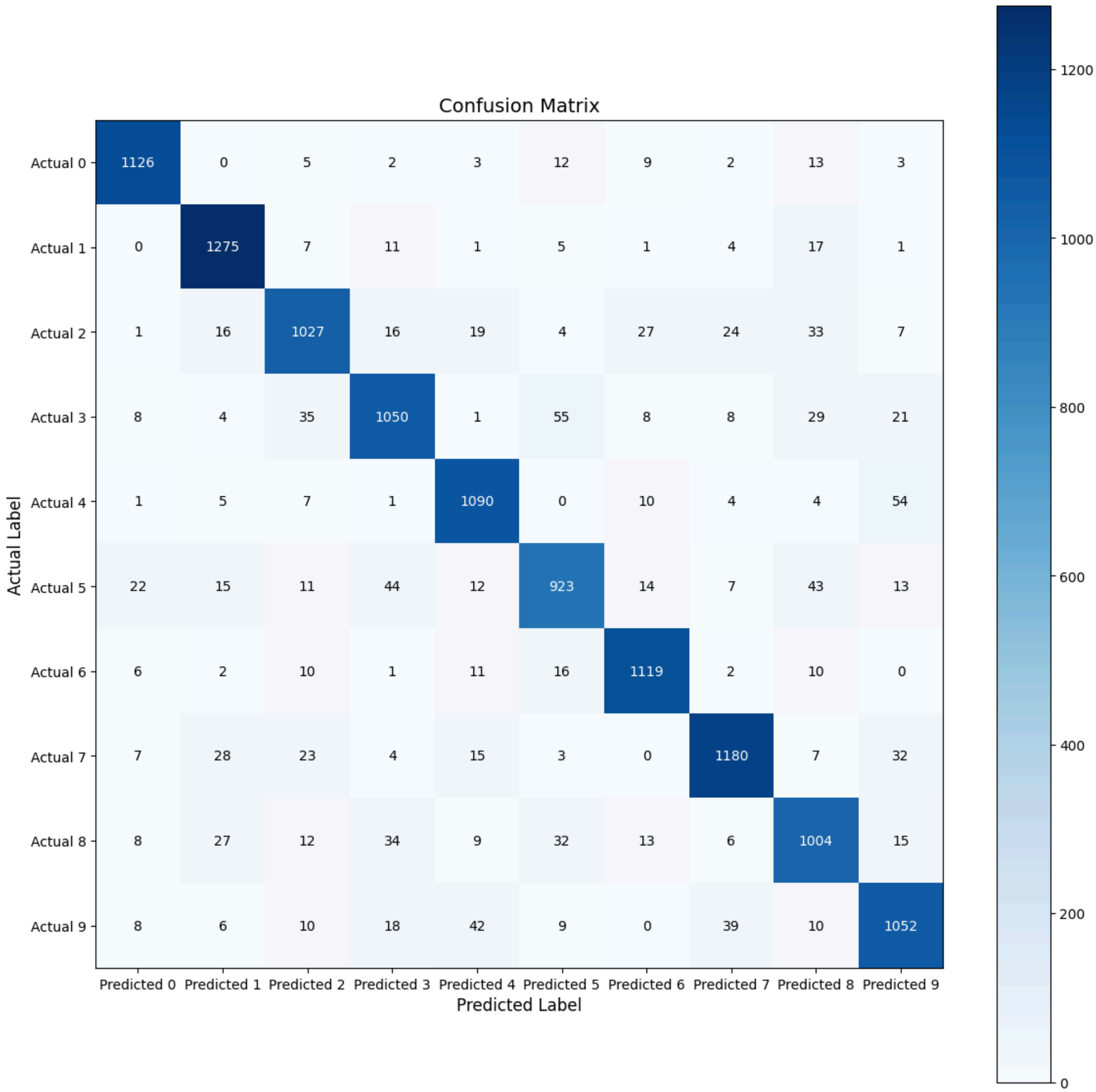


Evalutaing the model

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score


# Evaluation Function
def evaluate_classification(y_true,y_pred):
    """
    Evaluate classification performance using confusion matrix, precision, recall, and F1-score.

    Parameters:
    y_true (numpy.ndarray): True labels
    y_pred (numpy.ndarray): Predicted labels

    Returns:
    tuple: Confusion matrix, precision, recall, F1 score
    """

    #  Compute confusion matrix
    cm = confusion_matrix(y_true,y_pred)

    # Compute precision, recall , and F1-score
    precision = precision_score(y_true,y_pred,average='weighted')
    recall = recall_score(y_true,y_pred,average='weighted')
    f1 = f1_score(y_true,y_pred,average='weighted')

    return cm,precision,recall,f1


# Prdict on the test score
y_pred_test = predict_softmax(X_test,W_opt,b_opt)

# Evaluate accuracy
y_test_labels = np.argmax(y_test,axis=1) # True labels in numeric form

# Evaluate the model
cm,precision,recall,f1 = evaluate_classification(y_test_labels,y_pred_test)
```

```
# Print the evlauation metrics
print("\nConfusion Matrix:")
print(cm)
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-score: {f1:.2f}")

# Visualizing the Confusion Matrix
fig, ax = plt.subplots(figsize=(12,12))
cax = ax.imshow(cm,cmap="Blues") # Use a color map for better visualization

# Dynamic number of classes
num_classes = cm.shape[0]
ax.set_xticks(range(num_classes))
ax.set_yticks(range(num_classes))
ax.set_xticklabels([f'Predicted {i}' for i in range(num_classes)])
ax.set_yticklabels([f'Actual {i}' for i in range(num_classes)])


# Add labels to each cell in the confusion matrix
for i in range(cm.shape[0]):
  for j in range(cm.shape[1]):
    ax.text(j,i,cm[i,j], ha='center', va='center', color='white' if cm[i,j] > np.max(cm) / 2 else 'black')



# Add labels to each cell in the confusion matrix
ax.grid(False)
plt.title('Confusion Matrix', fontsize=14)
plt.xlabel('Predicted Label', fontsize=12)
plt.ylabel('Actual Label', fontsize=12)


# Adjust layout
plt.tight_layout()
plt.colorbar(cax)
plt.show()
```

```
Confusion Matrix:
[[1126    0    5    2    3   12    9    2   13    3]
 [   0 1275    7   11    1    5    1    4   17    1]
 [   1   16 1027   16   19    4   27   24   33    7]
 [   8    4   35 1050    1   55    8    8   29   21]
 [   1    5    7    1 1090    0   10    4    4   54]
 [  22   15   11   44   12  923   14    7   43   13]
 [   6    2   10    1   11   16 1119    2   10    0]
 [   7   28   23    4   15    3    0 1180    7   32]
 [   8   27   12   34    9   32   13    6 1004   15]
 [   8    6   10   18   42    9    0   39   10 1052]]
Precision: 0.90
Recall: 0.90
F1-score: 0.90
```



```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification, make_circles
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Set random seed for reproducibility
np.random.seed(42)

# Generate linearly separable dataset
X_linear_separable, y_linear_separable = make_classification(n_samples=200, n_features=2,
n_informative=2,
n_redundant=0, n_clusters_per_class=1,
random_state=42)

# Split the data into training and testing sets
X_train_linear, X_test_linear, y_train_linear, y_test_linear = train_test_split(
X_linear_separable, y_linear_separable, test_size=0.2, random_state=42
)

# Train logistic regression model on linearly separable data
logistic_model_linear_separable = LogisticRegression()
logistic_model_linear_separable.fit(X_train_linear, y_train_linear)

# Generate non-linearly separable dataset (circles)
X_non_linear_separable, y_non_linear_separable = make_circles(n_samples=200, noise=0.1, factor=0.5,
random_state=42)

# Split the data into training and testing sets
X_train_non_linear, X_test_non_linear, y_train_non_linear, y_test_non_linear = train_test_split(
X_non_linear_separable, y_non_linear_separable, test_size=0.2, random_state=42
)

# Train logistic regression model on non-linearly separable data
logistic_model_non_linear_separable = LogisticRegression()
logistic_model_non_linear_separable.fit(X_train_non_linear, y_train_non_linear)

# Plot decision boundaries for linearly and non-linearly separable data
def plot_decision_boundary(ax, model, X, y, title):
    h = .02 # step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    ax.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.Paired)
    ax.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.Paired)  # Corrected here
    ax.set_title(title)
    ax.set_xlabel('Feature 1')  # Corrected here
    ax.set_ylabel('Feature 2')  # Corrected here

# Create subplots
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Plot decision boundary for linearly separable data (Training)
plot_decision_boundary(axes[0, 0], logistic_model_linear_separable, X_train_linear, y_train_linear,
'Linearly Separable Data (Training)')  # Corrected here

# Plot decision boundary for linearly separable data (Testing)
plot_decision_boundary(axes[0, 1], logistic_model_linear_separable, X_test_linear, y_test_linear,
'Linearly Separable Data (Testing)')  # Corrected here

# Plot decision boundary for non-linearly separable data (Training)
plot_decision_boundary(axes[1, 0], logistic_model_non_linear_separable, X_train_non_linear,
```