Department of Robotics & AI, SMME, NUST

RAI-832 Machine Learning

Assignment 02

Muhammad Huzaifa

539641

Submitted to: Dr Jawad Khan

Dated: November 19, 2025.

Contents

Report: Face Recognition Using a Shallow Neural Network

1. Introduction

This assignment focuses on developing a personal face-recognition system using a shallow neural network implemented entirely with NumPy. Face recognition is a controlled binary classification task, where the model must decide whether an input image belongs to the target identity or not. A shallow neural network, which consists of a single hidden layer, offers a simple yet effective approach for learning basic visual patterns from images. In this assignment, the neural network uses a 32-neuron hidden layer with sigmoid activation in both layers. Each image is first converted to grayscale, resized to a fixed dimension of 720×720 pixels, flattened into a feature vector of 518,400 values, and normalized between 0 and 1. This ensures that all inputs follow a consistent format, making training more stable and predictable.

The assignment requires training the model without any external machine-learning libraries, meaning that all components—forward pass, loss computation, backpropagation, gradient updates, and evaluation—are implemented manually. The dataset is divided into training, validation, and testing sets using a 60/20/20 split. The model is trained using binary cross-entropy loss and gradient descent, and its performance is monitored through both training and validation losses. A fixed recognition threshold is applied during prediction to convert the output probability into a binary decision. Only positive examples (images of the target face) are used for training, while testing is performed separately using both positive and negative images to measure real-world performance.

To support evaluation, the assignment also includes a graphical user interface (GUI) that loads the trained model parameters and applies the recognition function to images from the test folder or from the user's device. The GUI displays the prediction, probability score, and image information, and it includes a manual evaluation panel that records actual labels and correctness to compute metrics such as accuracy, precision, recall, false-positive rate, and confusion-matrix statistics. This makes the assignment complete, measurable, and practical, showing how a shallow neural network behaves in controlled face-recognition tasks.

2. Dataset Details

2.1 Gathering and Cleaning the Data

The dataset was created by collecting a total of 739 images of the target face. These images were selected to cover natural variation, including changes in lighting, facial angle, background, and distance from the camera. Additional older images were also included to increase diversity and help the model learn stable facial features. All images were reviewed, and low-quality or blurred samples were removed to ensure consistency. After cleaning, the dataset was shuffled to avoid ordering bias, and random selection was applied automatically during the train–validation–test split. Since this assignment focuses on personal face recognition, only images of the target identity were used for training.

For performance evaluation, the test folder was prepared separately. It contains a mix of the target face under different conditions as well as non-target samples, including other human faces and various objects. This allows the model to be tested not only on familiar identities but also on entirely new and unrelated images, making the evaluation more realistic and meaningful.

2.2 Dataset Size

All images were stored in dataset/me/.

- Total images: 739 Images, all labeled as 1

Dataset split:

- 60% training

- 20% validation

- 20% testing

2.3 Feature Details and Scaling

Each image is:

1. Converted to grayscale

2. Resized to 720×720 pixels

3. Flattened into a vector of 518,400 features

4. Normalized to range 0–1

2.4 Code and Methodology

The full pipeline is built using NumPy only. No ML libraries were used.

Architecture:

Input (518,400) → Hidden (32 neurons, sigmoid) → Output (1 neuron, sigmoid)

Threshold used: 0.9720 for classification.

3. Mathematical Model

3.1 Hypothesis Function

Z1 = W1·X + b1

A1 = sigmoid(Z1)

Z2 = W2·A1 + b2

A2 = sigmoid(Z2)

3.2 Objective Function

Binary cross-entropy:

$$J = -\frac{1}{m}\sum [y\log(\hat{y}) + (1-y)\log(1-\hat{y})]$$

3.3 Parameter Optimization

Parameters updated using manual gradient descent:

$$W = W - \text{learning\_rate} \times dW$$

$$b = b - \text{learning\_rate} \times db$$

4. Model Output

The model outputs probability and a binary decision (Huzaifa / Unrecognized).

Parameters saved in results/model_params.npz.

5. Model Training Details

- Epochs: 200

- Learning rate: 0.01

- Hidden neurons: 32

- Graphs saved to results/training_loss.png.

6. Evaluation Metrics

Confusion Matrix:

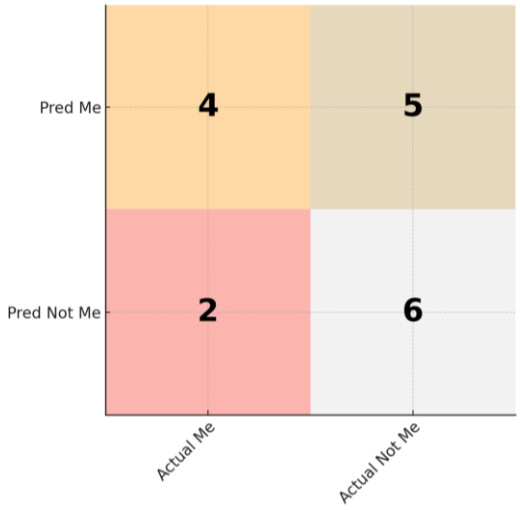

Figure 1 Confusion Matrix

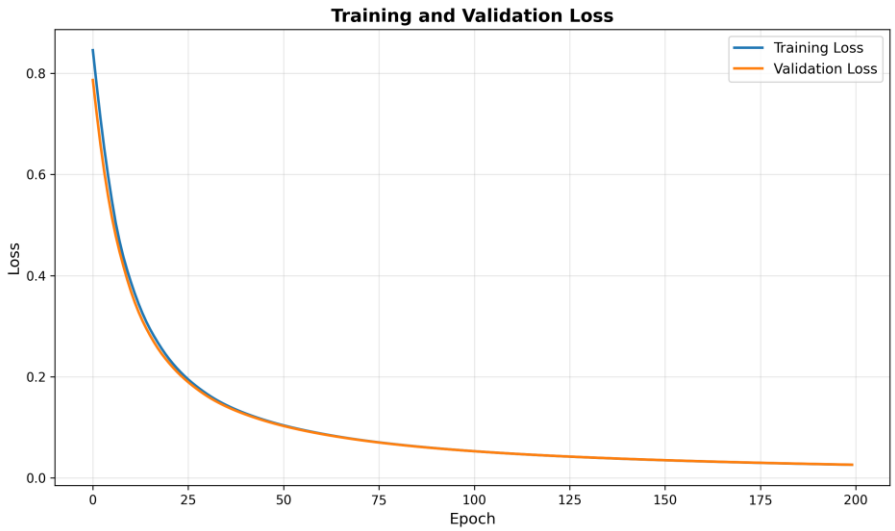

Figure 2 Training and Validation Loss

Accuracy: 58.8%

Precision: 66.7%

Recall: 44.4%

False Positive Rate: 25%

7. Results

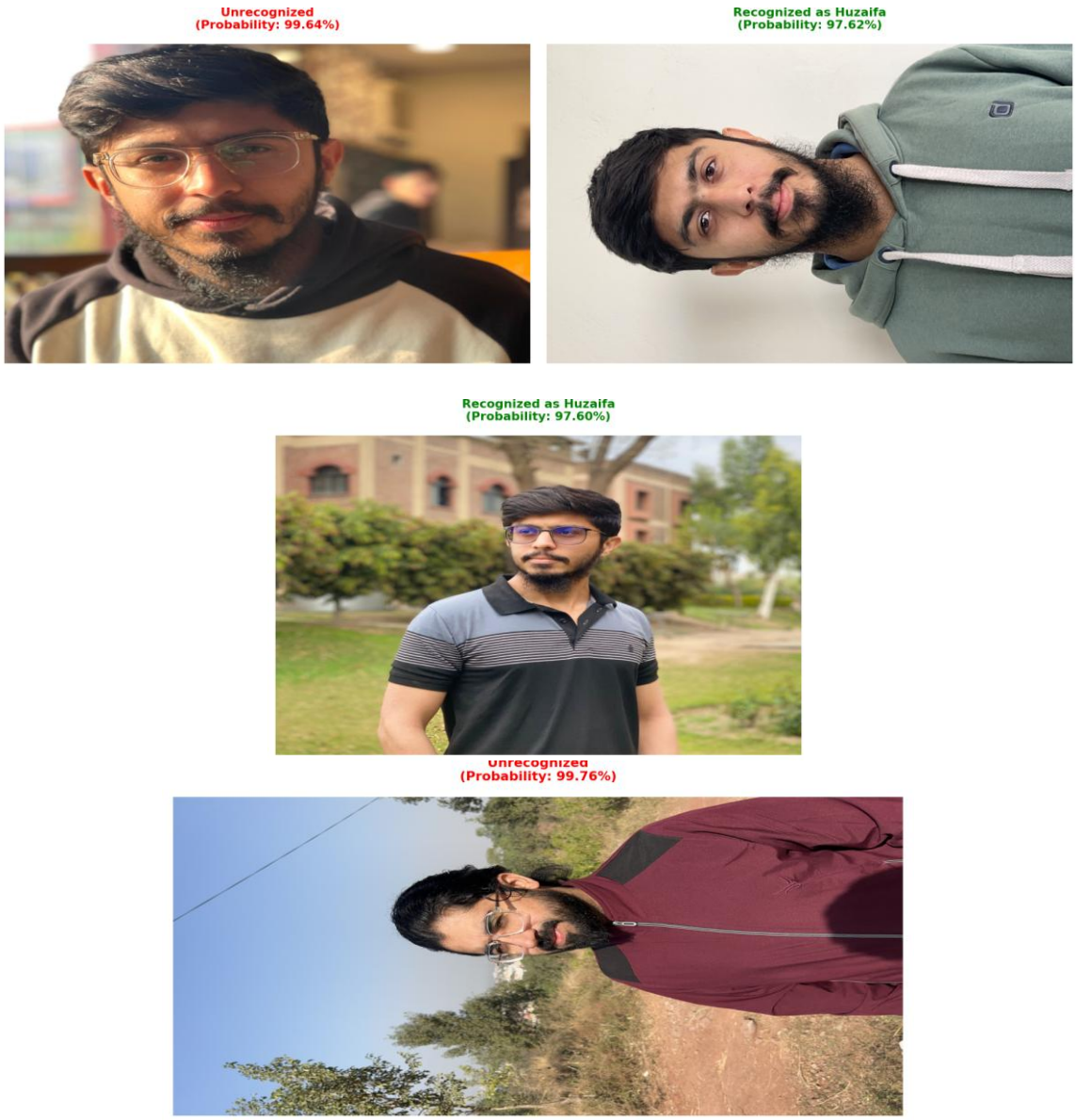

Figure 3 Result on image in test folder

8. GUI Application

A complete GUI was developed:

- Shows predictions on all test images

- Allows loading images from device

- Computes TP, TN, FP, FN

- Displays Accuracy, Precision, Recall, FPR

The GUI helps evaluate real-world behavior of the model.

9. Annex A

To run the complete pipeline, main.py is run without arguments, it trains the model on images from dataset/me/, then displays predictions on all images in dataset/test/ one by one using matplotlib.

```
python main.py
```

Running the Training Script

This command trains the shallow neural network and saves the model parameters:

```
python main.py --train
```

Running Prediction on a Single Image

Use this command and give the path of the image:

```
python main.py --predict path/to/image.jpg
```

Running the GUI Application

This loads the graphical interface for testing images and viewing predictions:

```
python main.py --gui
```

Default Paths

- Trained model is saved at: results/model_params.npz
- Test images should be placed in: dataset/test/

10. Annex B

The training script implements a shallow neural network using only NumPy and follows a complete workflow that includes loading images from the dataset, converting them to grayscale, resizing them to 720×720 pixels, normalizing the pixel values, and flattening each image into a fixed-length vector. The processed dataset is randomly divided into 60% training, 20% validation, and 20% testing. The network itself consists of one hidden layer with 32 sigmoid neurons and an output layer with a single sigmoid neuron for binary classification. During training, the script performs a forward pass, computes binary cross-entropy loss, calculates gradients through manual backpropagation, and updates all parameters using gradient descent for 200 epochs. Training and validation losses are recorded and plotted, and after training completes, the final learned parameters (W1, b1, W2, b2) are saved in model_params.npz, which is later used by the prediction and GUI modules.

Code:

```
"""
Training script for shallow neural network face detection.
Implements neural network from scratch using only NumPy.
"""

import numpy as np
import matplotlib.pyplot as plt
import os
import sys

# Add project root to path to import utils
# This allows the script to be run from project root: python code/train.py
script_dir = os.path.dirname(os.path.abspath(__file__))
project_root = os.path.dirname(script_dir)
if project_root not in sys.path:
    sys.path.insert(0, project_root)

from code.utils import load_dataset, split_dataset

# Recognition threshold constant - change this value to adjust classification threshold
RECOGNITION_THRESHOLD = 0.9750


class ShallowNeuralNetwork:
    """
    Shallow neural network with one hidden layer.
    Architecture: Input -> Hidden -> Output
```

```
    Activation: Sigmoid for both layers
    """

    def __init__(self, input_size, hidden_size, output_size,
learning_rate=0.01):
        """
        Initialize the neural network.

        Parameters:
        -----------
        input_size : int
            Number of input features (flattened image size)
        hidden_size : int
            Number of neurons in hidden layer
        output_size : int
            Number of output neurons (1 for binary classification)
        learning_rate : float
            Learning rate for gradient descent
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Initialize weights with small random values
        # W1: weights from input to hidden layer
        # b1: biases for hidden layer
        # W2: weights from hidden to output layer
        # b2: bias for output layer
        np.random.seed(42)  # For reproducibility
        self.W1 = np.random.randn(hidden_size, input_size) * 0.1
        self.b1 = np.zeros((hidden_size, 1))
        self.W2 = np.random.randn(output_size, hidden_size) * 0.1
        self.b2 = np.zeros((output_size, 1))


    def sigmoid(self, x):
        """
        Sigmoid activation function.
        σ(x) = 1 / (1 + exp(-x))
        """
        # Clip x to avoid overflow
        x = np.clip(x, -500, 500)
        return 1 / (1 + np.exp(-x))


    def sigmoid_derivative(self, x):
        """
```

```
    Derivative of sigmoid function.
    σ'(x) = σ(x) * (1 - σ(x))
    """
    s = self.sigmoid(x)
    return s * (1 - s)


def forward_pass(self, X):
    """
    Forward pass through the network.

    Parameters:
    -----------
    X : numpy.ndarray
        Input data (n_samples, n_features)

    Returns:
    --------
    Z1 : hidden layer pre-activation
    A1 : hidden layer activation
    Z2 : output layer pre-activation
    A2 : output layer activation (predictions)
    """
    # Reshape X if needed (ensure it's 2D)
    if X.ndim == 1:
        X = X.reshape(1, -1)

    # Input to hidden layer
    # Z1 = W1 * X^T + b1
    Z1 = np.dot(self.W1, X.T) + self.b1
    A1 = self.sigmoid(Z1)

    # Hidden to output layer
    # Z2 = W2 * A1 + b2
    Z2 = np.dot(self.W2, A1) + self.b2
    A2 = self.sigmoid(Z2)

    return Z1, A1, Z2, A2


def backward_pass(self, X, y, Z1, A1, Z2, A2):
    """
    Backward pass (backpropagation) to compute gradients.

    Parameters:
    -----------
    X : numpy.ndarray
        Input data (n_samples, n_features)
```

```
        y : numpy.ndarray
            True labels (n_samples,)
        Z1 : numpy.ndarray
            Hidden layer pre-activation
        A1 : numpy.ndarray
            Hidden layer activation
        Z2 : numpy.ndarray
            Output layer pre-activation
        A2 : numpy.ndarray
            Output layer activation (predictions)

        Returns:
        --------
        dW1, db1, dW2, db2 : gradients for weight updates
        """
        m = X.shape[0]  # Number of samples

        # Reshape y if needed
        if y.ndim == 1:
            y = y.reshape(1, -1)
        else:
            y = y.T

        # Output layer error
        # dZ2 = A2 - y (derivative of binary cross-entropy loss with
sigmoid)
        dZ2 = A2 - y

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

        # Hidden layer error
        dA1 = np.dot(self.W2.T, dZ2)
        dZ1 = dA1 * self.sigmoid_derivative(Z1)

        # Gradients for hidden layer
        dW1 = (1 / m) * np.dot(dZ1, X)
        db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)

        return dW1, db1, dW2, db2


    def update_weights(self, dW1, db1, dW2, db2):
        """
        Update weights using gradient descent.

        Parameters:
```

```
        -----------
        dW1, db1, dW2, db2 : gradients
        """
        self.W1 -= self.learning_rate * dW1
        self.b1 -= self.learning_rate * db1
        self.W2 -= self.learning_rate * dW2
        self.b2 -= self.learning_rate * db2


    def compute_loss(self, y_true, y_pred):
        """
        Compute binary cross-entropy loss.

        Parameters:
        -----------
        y_true : numpy.ndarray
            True labels
        y_pred : numpy.ndarray
            Predicted probabilities

        Returns:
        --------
        loss : float
            Mean loss value
        """
        # Clip predictions to avoid log(0)
        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)

        # Reshape if needed
        if y_true.ndim == 1:
            y_true = y_true.reshape(1, -1)
        else:
            y_true = y_true.T

        m = y_true.shape[1]

        # Binary cross-entropy loss
        loss = -(1 / m) * np.sum(y_true * np.log(y_pred) + (1 - y_true)
* np.log(1 - y_pred))

        return loss


    def compute_error(self, y_true, y_pred):
        """
        Compute mean absolute error.

        Parameters:
```

```
        -----------
        y_true : numpy.ndarray
            True labels
        y_pred : numpy.ndarray
            Predicted probabilities

        Returns:
        --------
        error : float
            Mean absolute error
        """
        if y_true.ndim == 1:
            y_true = y_true.reshape(-1, 1)
        if y_pred.ndim == 1:
            y_pred = y_pred.reshape(-1, 1)

        error = np.mean(np.abs(y_true - y_pred))
        return error


    def  train(self,  X_train,  y_train,  X_val,  y_val,  epochs=1000,
verbose=True):
        """
        Train the neural network.

        Parameters:
        -----------
        X_train : numpy.ndarray
            Training features
        y_train : numpy.ndarray
            Training labels
        X_val : numpy.ndarray
            Validation features
        y_val : numpy.ndarray
            Validation labels
        epochs : int
            Number of training epochs
        verbose : bool
            Whether to print progress

        Returns:
        --------
        train_losses : list
            Training loss per epoch
        val_losses : list
            Validation loss per epoch
        """
        train_losses = []
```

```
        val_losses = []

        if verbose:
            print(f"\nStarting training for {epochs} epochs...")
            print("Progress will be shown every 10 epochs.\n")

        for epoch in range(epochs):
            # Forward pass
            Z1, A1, Z2, A2 = self.forward_pass(X_train)

            # Compute loss
            train_loss = self.compute_loss(y_train, A2)
            train_losses.append(train_loss)

            # Backward pass
            dW1, db1, dW2, db2 = self.backward_pass(X_train, y_train,
Z1, A1, Z2, A2)

            # Update weights
            self.update_weights(dW1, db1, dW2, db2)

            # Validation loss
            _, _, _, A2_val = self.forward_pass(X_val)
            val_loss = self.compute_loss(y_val, A2_val)
            val_losses.append(val_loss)

            # Print progress every 10 epochs
            if verbose:
                if (epoch + 1) % 10 == 0 or epoch == 0:
                    print(f"Epoch {epoch + 1}/{epochs} - Train Loss:
{train_loss:.4f}, Val Loss: {val_loss:.4f}")
                elif (epoch + 1) == epochs:
                    print(f"Epoch {epoch + 1}/{epochs} - Train Loss:
{train_loss:.4f}, Val Loss: {val_loss:.4f}")

        if verbose:
            print("\nTraining completed!\n")

        return train_losses, val_losses


    def predict_proba(self, X):
        """
        Predict probabilities for input data.

        Parameters:
        -----------
        X : numpy.ndarray
            Input features
```

```
        Returns:
        --------
        probabilities : numpy.ndarray
            Predicted probabilities
        """
        _, _, _, A2 = self.forward_pass(X)
        return A2.flatten()


    def predict(self, X, threshold=None):
        """
        Predict binary class labels.

        Parameters:
        -----------
        X : numpy.ndarray
            Input features
        threshold : float, optional
            Classification threshold (default: RECOGNITION_THRESHOLD)

        Returns:
        --------
        predictions : numpy.ndarray
            Binary predictions (0 or 1)
        """
        if threshold is None:
            threshold = RECOGNITION_THRESHOLD
        probabilities = self.predict_proba(X)
        return (probabilities >= threshold).astype(int)


    def save_model(self, filepath):
        """
        Save model parameters to file.

        Parameters:
        -----------
        filepath : str
            Path to save model
        """
        np.savez(filepath,    W1=self.W1,    b1=self.b1,    W2=self.W2,
b2=self.b2)
        print(f"Model saved to {filepath}")


    def load_model(self, filepath):
        """
```

```
        Load model parameters from file.

        Parameters:
        -----------
        filepath : str
            Path to load model from
        """
        data = np.load(filepath)
        self.W1 = data['W1']
        self.b1 = data['b1']
        self.W2 = data['W2']
        self.b2 = data['b2']
        print(f"Model loaded from {filepath}")


def main():
    """
    Main training function.
    """
    # Set random seed for reproducibility
    np.random.seed(42)

    # Get the project root directory (parent of code directory)
    script_dir = os.path.dirname(os.path.abspath(__file__))
    project_root = os.path.dirname(script_dir)

    # Configuration
    faces_folder = os.path.join(project_root, "dataset", "me")
    target_size = (720, 720)  # Image size
    hidden_size = 32  # Number of neurons in hidden layer
    learning_rate = 0.01
    epochs = 200

    # Output paths
    results_dir = os.path.join(project_root, "results")
    model_path = os.path.join(results_dir, "model_params.npz")
    plot_path = os.path.join(results_dir, "training_loss.png")
    errors_path = os.path.join(results_dir, "errors.txt")

    # Create results directory if it doesn't exist
    os.makedirs(results_dir, exist_ok=True)

    print("=" * 50)
    print("Huzaifa Face Recognition System - Training")
    print("=" * 50)
    print("Training model to recognize Huzaifa's face")
    print("-" * 50)
```

```
    # Load dataset (only from "me" folder)
    print("Loading dataset...")
    X, y = load_dataset(faces_folder, None, target_size=target_size)

    if len(X) == 0:
        print("ERROR: No images found in dataset folder!")
        print("Please add images to:")
        print("  - dataset/me/ (Huzaifa's photos)")
        return

    # Split dataset
    print("\nSplitting dataset...")
    X_train, y_train, X_val, y_val, X_test, y_test = split_dataset(
        X, y, train_ratio=0.6, val_ratio=0.2, test_ratio=0.2
    )

    # Initialize network
    input_size = X_train.shape[1]  # Flattened image size
    output_size = 1  # Binary classification

    print(f"\nInitializing neural network...")
    print(f"  - Input size: {input_size}")
    print(f"  - Hidden size: {hidden_size}")
    print(f"  - Output size: {output_size}")
    print(f"  - Learning rate: {learning_rate}")
    print(f"  - Epochs: {epochs}")

    nn = ShallowNeuralNetwork(input_size, hidden_size, output_size,
learning_rate)

    # Train network
    print(f"\n{'='*50}")
    print("Starting Training Process")
    print(f"{'='*50}")
    train_losses, val_losses = nn.train(X_train, y_train, X_val, y_val,
epochs=epochs, verbose=True)

    # Save model
    print(f"\nSaving model...")
    nn.save_model(model_path)

    # Plot training loss
    print(f"\nPlotting training loss...")
    plt.figure(figsize=(10, 6))
    plt.plot(train_losses, label='Training Loss', linewidth=2)
    plt.plot(val_losses, label='Validation Loss', linewidth=2)
    plt.xlabel('Epoch', fontsize=12)
    plt.ylabel('Loss', fontsize=12)
```

```
    plt.title('Training and Validation Loss', fontsize=14,
fontweight='bold')
    plt.legend(fontsize=11)
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.savefig(plot_path, dpi=300, bbox_inches='tight')
    print(f"Plot saved to {plot_path}")
    plt.close()

    # Evaluate on all sets
    print(f"\nEvaluating model...")

    # Training set
    y_train_pred = nn.predict_proba(X_train)
    train_error = nn.compute_error(y_train, y_train_pred)
    train_accuracy = np.mean((y_train_pred >=
RECOGNITION_THRESHOLD).astype(int) == y_train)

    # Validation set
    y_val_pred = nn.predict_proba(X_val)
    val_error = nn.compute_error(y_val, y_val_pred)
    val_accuracy = np.mean((y_val_pred >=
RECOGNITION_THRESHOLD).astype(int) == y_val)

    # Test set
    y_test_pred = nn.predict_proba(X_test)
    test_error = nn.compute_error(y_test, y_test_pred)
    test_accuracy = np.mean((y_test_pred >=
RECOGNITION_THRESHOLD).astype(int) == y_test)

    # Print results
    print(f"\nResults:")
    print(f"  Training - Error: {train_error:.4f}, Accuracy:
{train_accuracy:.4f}")
    print(f"  Validation - Error: {val_error:.4f}, Accuracy:
{val_accuracy:.4f}")
    print(f"  Test - Error: {test_error:.4f}, Accuracy:
{test_accuracy:.4f}")

    # Save errors to file
    with open(errors_path, 'w') as f:
        f.write("Mean Errors and Accuracy\n")
        f.write("=" * 50 + "\n\n")
        f.write(f"Training Set:\n")
        f.write(f"  Mean Error: {train_error:.6f}\n")
        f.write(f"  Accuracy: {train_accuracy:.6f}\n\n")
        f.write(f"Validation Set:\n")
        f.write(f"  Mean Error: {val_error:.6f}\n")
        f.write(f"  Accuracy: {val_accuracy:.6f}\n\n")
```

```
            f.write(f"Test Set:\n")
            f.write(f"  Mean Error: {test_error:.6f}\n")
            f.write(f"  Accuracy: {test_accuracy:.6f}\n")

    print(f"\nErrors saved to {errors_path}")
    print("\nTraining completed successfully!")


if __name__ == "__main__":
    main()
```

11. Annex C

The prediction script loads the trained model parameters from the saved file (model_params.npz), reconstructs the shallow neural network, and applies it to new input images. For each image, the script performs the same preprocessing steps used during training, including grayscale conversion, resizing to 720×720, flattening, and normalization. The script then runs a forward pass through the network to compute the output probability and uses the predefined recognition threshold to convert this probability into a binary decision. It includes helper functions for predicting a single feature vector, loading and predicting images directly from file paths, and returning both the final classification and probability score. This prediction module is used by both the command-line interface and the GUI to ensure consistent and reproducible recognition results across all parts of the assignment.

Code:

```
"""
Prediction function for face detection.
Loads trained model and makes predictions on new images.
"""

import numpy as np
import os
import sys

# Add project root to path to import utils
# This allows the script to be run from project root: python code/predict.py
script_dir = os.path.dirname(os.path.abspath(__file__))
project_root = os.path.dirname(script_dir)
if project_root not in sys.path:
    sys.path.insert(0, project_root)

# Import recognition threshold constant
from code.train import RECOGNITION_THRESHOLD

class ShallowNeuralNetwork:
    """
    Shallow neural network with one hidden layer.
    Same architecture as in train.py for loading saved models.
    """

    def __init__(self, input_size, hidden_size, output_size):
        """
        Initialize the neural network.
        """
```

```
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize with zeros (will be loaded from file)
        self.W1 = None
        self.b1 = None
        self.W2 = None
        self.b2 = None


    def sigmoid(self, x):
        """
        Sigmoid activation function.
        """
        x = np.clip(x, -500, 500)
        return 1 / (1 + np.exp(-x))


    def forward_pass(self, X):
        """
        Forward pass through the network.
        """
        if X.ndim == 1:
            X = X.reshape(1, -1)

        Z1 = np.dot(self.W1, X.T) + self.b1
        A1 = self.sigmoid(Z1)

        Z2 = np.dot(self.W2, A1) + self.b2
        A2 = self.sigmoid(Z2)

        return A2


    def load_model(self, filepath):
        """
        Load model parameters from file.
        """
        data = np.load(filepath)
        self.W1 = data['W1']
        self.b1 = data['b1']
        self.W2 = data['W2']
        self.b2 = data['b2']


    def predict_proba(self, X):
        """
```

```
        Predict probabilities for input data.
        """
        A2 = self.forward_pass(X)
        return A2.flatten()


    def predict(self, X, threshold=None):
        """
        Predict binary class labels.
        """
        if threshold is None:
            threshold = RECOGNITION_THRESHOLD
        probabilities = self.predict_proba(X)
        return (probabilities >= threshold).astype(int)

def prediction(features, model_path=None, threshold=None):
    """
     Prediction function that takes image features as input and outputs
0 or 1.

    Parameters:
    -----------
    features : numpy.ndarray
         Image features (flattened image vector) - can be single sample
or batch
              Shape: (n_features,) for single sample or (n_samples,
n_features) for batch
    model_path : str
                 Path to saved model parameters file (default:
results/model_params.npz relative to project root)
    threshold : float, optional
        Classification threshold (default: RECOGNITION_THRESHOLD)

    Returns:
    --------
    predictions : numpy.ndarray or int
        Binary predictions (0 or 1)
        Returns int for single sample, array for batch
    """
    # Set default model path if not provided
    if model_path is None:
        script_dir = os.path.dirname(os.path.abspath(__file__))
        project_root = os.path.dirname(script_dir)
                model_path = os.path.join(project_root, "results",
"model_params.npz")

    # Load model
    if not os.path.exists(model_path):
```

```
          raise FileNotFoundError(f"Model file not found: {model_path}.
Please train the model first.")

    # Load model first to detect expected input size
    data = np.load(model_path)
    W1 = data['W1']
     model_input_size = W1.shape[1]   # Model expects this many input
features

    # Determine input size from features
    if features.ndim == 1:
        input_size = features.shape[0]
        features = features.reshape(1, -1)
    else:
        input_size = features.shape[1]

    # Check if feature size matches model's expected size
    if input_size != model_input_size:
        import math
        img_dim = int(math.sqrt(model_input_size))
        raise ValueError(
              f"Feature size mismatch: Model expects {model_input_size}
features "
                  f"({img_dim}x{img_dim} image), but got {input_size}
features. "
                f"Please ensure the input features match the model's
training size."
        )

    # Model parameters (should match training configuration)
    hidden_size = 32
    output_size = 1

    # Initialize and load model
    nn = ShallowNeuralNetwork(input_size, hidden_size, output_size)
    nn.load_model(model_path)

    # Use default threshold if not provided
    if threshold is None:
        threshold = RECOGNITION_THRESHOLD

    # Make prediction
    predictions = nn.predict(features, threshold=threshold)

    # Return single value for single sample, array for batch
    if predictions.size == 1:
        return int(predictions[0])
    else:
        return predictions
```

```
def load_and_predict_image(image_path, model_path=None):
    """
    Helper function to load an image and make prediction.

    Parameters:
    -----------
    image_path : str
        Path to image file
    model_path : str
                 Path to saved model parameters file (default:
results/model_params.npz)

    Returns:
    --------
    prediction : int
        Binary prediction (0 or 1)
    probability : float
        Predicted probability
    """
    from PIL import Image

    # Set default model path if not provided
    if model_path is None:
        script_dir = os.path.dirname(os.path.abspath(__file__))
        project_root = os.path.dirname(script_dir)
                model_path = os.path.join(project_root, "results",
"model_params.npz")

    # Load model first to detect expected input size
    if not os.path.exists(model_path):
         raise FileNotFoundError(f"Model file not found: {model_path}.
Please train the model first.")

    data = np.load(model_path)
    W1 = data['W1']
     model_input_size = W1.shape[1]   # Model expects this many input
features

    # Calculate expected image dimensions from model input size
    import math
    img_dim = int(math.sqrt(model_input_size))
    model_target_size = (img_dim, img_dim)

    # Load and preprocess image
    img = Image.open(image_path)
    if img.mode != 'L':
        img = img.convert('L')
```

```
      img = img.resize(model_target_size, Image.Resampling.LANCZOS)   # 
Resize to match model's expected size
    img_array = np.array(img, dtype=np.float64)
    img_flat = img_array.flatten()
    img_normalized = img_flat / 255.0

    # Get prediction (this will check if model exists)
    pred = prediction(img_normalized, model_path)

    # Get probability for more detailed output
    input_size = model_input_size
    hidden_size = 32
    output_size = 1

    nn = ShallowNeuralNetwork(input_size, hidden_size, output_size)
    nn.load_model(model_path)
    prob = nn.predict_proba(img_normalized.reshape(1, -1))[0]

    return pred, prob

def            recognize_huzaifa(image_path,            model_path=None, 
recognition_threshold=None):
    """
    Binary face recognition function for Huzaifa.
    Returns one of two categories based on probability:
    - "Recognized as Huzaifa" (probability > threshold)
    - "Unrecognized" (probability <= threshold)

    Parameters:
    -----------
    image_path : str
        Path to image file
    model_path : str
                  Path   to   saved   model   parameters   file   (default: 
results/model_params.npz)
    recognition_threshold : float, optional
             Probability  threshold  above  which  Huzaifa  is  recognized 
(default: RECOGNITION_THRESHOLD)

    Returns:
    --------
    recognition_result : str
        Either "Recognized as Huzaifa" or "Unrecognized"
    probability : float
        Predicted probability
    """
    from PIL import Image

    # Set default model path if not provided
```

```
    if model_path is None:
        script_dir = os.path.dirname(os.path.abspath(__file__))
        project_root = os.path.dirname(script_dir)
                model_path = os.path.join(project_root, "results",
"model_params.npz")

    # Check if model exists
    if not os.path.exists(model_path):
          raise FileNotFoundError(f"Model file not found: {model_path}.
Please train the model first.")

    # Load model first to detect expected input size
    data = np.load(model_path)
    W1 = data['W1']
     model_input_size = W1.shape[1]  # Model expects this many input
features

    # Calculate expected image dimensions from model input size
    # input_size = width * height, so we can determine the size
    import math
    img_dim = int(math.sqrt(model_input_size))
    model_target_size = (img_dim, img_dim)

      # Load and preprocess image (standardize to grayscale, resize,
normalize)
    img = Image.open(image_path)
    if img.mode != 'L':
        img = img.convert('L')  # Convert to grayscale
     img = img.resize(model_target_size, Image.Resampling.LANCZOS)  #
Resize to match model's expected size
    img_array = np.array(img, dtype=np.float64)
    img_flat = img_array.flatten()
    img_normalized = img_flat / 255.0  # Normalize to [0, 1]

    # Verify size matches
    if img_normalized.shape[0] != model_input_size:
        raise ValueError(
               f"Image size mismatch: Model expects {model_input_size}
features "
             f"({model_target_size[0]}x{model_target_size[1]}), but got
{img_normalized.shape[0]}. "
             f"Please retrain the model with the current image size or
use images matching the model's training size."
        )

    # Get probability
    input_size = model_input_size
    hidden_size = 32
    output_size = 1
```

```
    nn = ShallowNeuralNetwork(input_size, hidden_size, output_size)
    nn.load_model(model_path)
    prob = nn.predict_proba(img_normalized.reshape(1, -1))[0]

    # Use default threshold if not provided
    if recognition_threshold is None:
        recognition_threshold = RECOGNITION_THRESHOLD

    # Binary classification based on probability threshold
    if prob > recognition_threshold:
        result = "Recognized as Huzaifa"
    else:
        result = "Unrecognized"

    return result, prob

if __name__ == "__main__":
    """
    Example usage of prediction function.
    """
    # Get default model path
    script_dir = os.path.dirname(os.path.abspath(__file__))
    project_root = os.path.dirname(script_dir)
             model_path    =    os.path.join(project_root,    "results",
"model_params.npz")

    if not os.path.exists(model_path):
           print("Model not found. Please train the model first using
train.py")
        print(f"Expected model at: {model_path}")
    else:
         # Example with dummy features (should be replaced with actual
image features)
        # In practice, you would load and preprocess an image first
        print("Example usage of prediction() function:")
        print("=" * 50)

           # Create dummy features (720x720 grayscale image = 518,400
features)
        dummy_features = np.random.rand(518400)

        # Make prediction
        pred = prediction(dummy_features, model_path)
        print(f"Prediction for dummy features: {pred}")
        print("\nTo predict on real images, use:")
                  print("     from   code.predict   import   prediction,
load_and_predict_image")
```

```
                                    print("          pred,     prob     =
load_and_predict_image('path/to/image.jpg')")
```