INT410 FINAL COURSEWORK

Li Xingyu
Student ID: 2467854

ABSTRACT

This report focuses on the implementation and evaluation of advanced pattern recognition techniques using Restricted Boltzmann Machines (RBM) and Multi-Layer Perceptrons (MLP) on the Labeled Faces in the Wild (LFW) dataset. The study is structured into two primary tasks. Task 1 explores the effect of different RBM parameter configurations on feature extraction and image reconstruction. This includes training the RBM with different parameters and evaluating the performance of the trained models using reconstructed images and loss curves. The second task covers designing, implementing, and comparing an MLP classifier and a KNN classifier using features extracted through the RBM. The performance evaluation of the classifiers is done by accuracy, precision, recall, and F1 score, with further insights through the visualizations of prediction trends and training dynamics. Experimental results show how RBM configurations strongly affect the quality of the features and point out the much better performance of the MLP classifier with respect to KNN in facial image classification tasks. This work underlines the importance of effective unsupervised feature extraction and gives a detailed comparison of the capability of classifiers when dealing with high-dimensional data. The results will form useful guidelines for applications in facial recognition and more general pattern recognition problems.

Keywords: Pattern recognition, Restricted Boltzmann Machines, LFW dataset.

1 INTRODUCTION

Pattern recognition has become a cornerstone of advancements in artificial intelligence, with applications spanning areas like image and speech recognition. For the final project of INT410 Advanced Pattern Recognition, we apply the concepts learned throughout the course to a practical task: image classification using the Labeled Faces in the Wild (LFW) dataset.

This project allows us to deepen our understanding of pattern recognition techniques by working with two specific classifiers: the Restricted Boltzmann Machine (RBM) and the Multilayer Perceptron (MLP). Using the RBM, an unsupervised learning model, we extract features and reconstruct images, experimenting with different parameter settings to understand their impact on model performance. This step is crucial in exploring how deep learning models can be adjusted and optimized for better results.

The next part of the project focuses on classification using features extracted by the RBM. We design and optimize two classifiers: one based on the MLP and another custom classifier of our choice. This stage tests our ability to implement, fine-tune, and evaluate these models to achieve strong classification performance on the LFW dataset.

It would also have significant value because the project would narrow the gap between theoretical knowledge and real-life applications: from practical experience working on an actual dataset, knowledge could be extracted regarding the degree to which different model architectures and different hyperparameters help determine classification accuracy. A close insight into the model designed, trained, and then evaluated would be included in our detailed report of methodology and our findings.

The following project is the practical implementation of pattern recognition. Hence, it is an effective way of learning how machine learning systems can be developed and fine-tuned. The knowledge and experience learned will not only help reinforce our understanding of the important concepts but also prepare us for similar challenges in the future.

2 DATASET

The Labeled Faces in The Wild (LFW) dataset is a comprehensive collection of facial images aimed at studying unconstrained facial recognition. It contains 13233 photos of 5749 people, and 1680 people have at least two different photos. These images were detected and centered using the Viola Jones facial detector, and further processed into multiple versions, including the deep funnel version, which has been shown to improve the accuracy of facial validation. Each image is in 250x250 JPG format.

The dataset also includes metadata to facilitate model training and evaluation, such as pre-defined pairwise or personnel configured 10 fold cross validation segmentation. Specific training and testing sets are provided for matching/mismatched pairs or individual identities, achieving restricted and unrestricted experimental settings.

This dataset is widely used in facial recognition research and serves as a benchmark for evaluating the performance of algorithms in unconstrained environments.

3 RESTRICTED BOLTZMANN MACHINE (RBM)

A variant of artificial neural networks is the Restricted Boltzmann Machine, which is designed for unsupervised learning. It is normally used for feature extraction, dimensionality reduction, and pre-training deep neural networks. An RBM contains two layers of neurons:

- **Visible Layer ($\mathbf{x}$):** Represents the input data, where each neuron corresponds to one feature or pixel in the input.
- **Hidden Layer ($\mathbf{h}$):** Represents latent features that the RBM learns to extract from the input data.

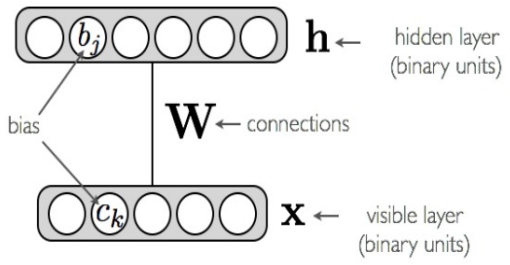
The two layers are connected by a set of weights $\mathbf{W}$, but there are no intra-layer connections (hence the term "restricted"). The RBM models the joint probability distribution of the visible and hidden layers.

$$p(\mathbf{h}|\mathbf{x}) = \prod_j p(h_j|\mathbf{x})$$

$$p(h_j = 1|\mathbf{x}) = \frac{1}{1+\exp(-(b_j + \mathbf{W}_j.\mathbf{x}))}$$

$$= \text{sigm}(b_j + \mathbf{W}_j.\mathbf{x})$$

jth row of W



$$p(\mathbf{x}|\mathbf{h}) = \prod_k p(x_k|\mathbf{h})$$

$$p(x_k = 1|\mathbf{h}) = \frac{1}{1+\exp(-(c_k + \mathbf{h}^\top \mathbf{W}_{\cdot k}))}$$

$$= \text{sigm}(c_k + \mathbf{h}^\top \mathbf{W}_{\cdot k})$$

kth column of W

$$E(\mathbf{x}, \mathbf{h}) = -\mathbf{h}^\top \mathbf{W}\mathbf{x} - \mathbf{c}^\top \mathbf{x} - \mathbf{b}^\top \mathbf{h}$$

$$= -\sum_j \sum_k W_{j,k} h_j x_k - \sum_k c_k x_k - \sum_j b_j h_j$$

Figure 1: RMB Fuctions and structure

3.1 ENERGY FUNCTION

The energy function $E(\mathbf{x}, \mathbf{h})$ defines the relationship between the visible layer $\mathbf{x}$ and the hidden layer $\mathbf{h}$:

$$E(\mathbf{x}, \mathbf{h}) = -\mathbf{h}^\top \mathbf{W}\mathbf{x} - \mathbf{c}^\top \mathbf{x} - \mathbf{b}^\top \mathbf{h}$$

where:

- $\mathbf{W}$ is the weight matrix connecting visible and hidden units.
- $\mathbf{b}$ is the bias vector for the hidden layer.
- $\mathbf{c}$ is the bias vector for the visible layer.

3.2 Probability Distributions

The RBM uses the following probability distributions:

- Conditional Probability of Hidden Units ($p(h_j = 1|\mathbf{x})$):
$$p(h_j = 1|\mathbf{x}) = \sigma(b_j + \mathbf{W}_j \cdot \mathbf{x})$$
where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid activation function, b_j is the bias of the j-th hidden unit, and $\mathbf{W}_j$ is the j-th row of the weight matrix.
- Conditional Probability of Visible Units ($p(x_k = 1|\mathbf{h})$):
$$p(x_k = 1|\mathbf{h}) = \sigma(c_k + \mathbf{W}_k^\top \cdot \mathbf{h})$$
where c_k is the bias of the k-th visible unit, and $\mathbf{W}_k$ is the k-th column of the weight matrix.

3.3 Training Process

The RBM is trained using a method called **Contrastive Divergence (CD)**. The objective is to minimize the difference between the data distribution and the model distribution. The steps are as follows:

1. Initialize the visible layer with the input data $\mathbf{x}$.
2. Compute the hidden layer activations $\mathbf{h}$ using the conditional probabilities $p(h_j = 1|\mathbf{x})$.
3. Reconstruct the visible layer $\mathbf{x}'$ from the hidden layer activations $\mathbf{h}$.
4. Compute the hidden layer activations $\mathbf{h}'$ from the reconstructed visible layer $\mathbf{x}'$.
5. Update the weights $\mathbf{W}$ and biases $\mathbf{b}$, $\mathbf{c}$ using the following gradient:
$$\Delta\mathbf{W} = \eta \left(\langle \mathbf{x}\mathbf{h}^\top \rangle_{\text{data}} - \langle \mathbf{x}'\mathbf{h}'^\top \rangle_{\text{model}} \right)$$
where η is the learning rate, and $\langle \cdot \rangle$ denotes the expectation.

4 Multilayer Perceptron (MLP)

The Multilayer Perceptron is an artificial neural network that is composed of several layers interconnected by neurons. It finds broad applications in supervised learning tasks of classification and regression problems, since it models complicated relationships between inputs and outputs. The MLP works by iteratively applying a combination of linear transformations and nonlinear activation functions.

4.1 Structure of MLP

As illustrated in the figure:

- **Input Layer:** The input layer consists of d input neurons, corresponding to the features of the input data. An additional bias term ($+1$) is often included, making the effective input dimension $d + 1$.
- **Hidden Layer(s):** The hidden layer contains m neurons, each of which computes a weighted sum of the input values followed by a nonlinear activation function. The number of hidden layers and the number of neurons in each layer are hyperparameters that determine the model's complexity.
- **Output Layer:** The output layer contains M neurons, each representing one output variable. For classification tasks, the output is often interpreted as probabilities over M classes, while for regression tasks, the output represents continuous values.

$$W = \arg\min_W \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{M}[y_k(x^n) - t_k^n]^2$$

$$y_k(x) = g[v_k(x)]$$

$$v_k(x) = \sum_{j=1}^{m} w_{kj}\, h_j + w_{k0}$$

$$h_j(x) = g[u_j(x)]$$

$$u_j(x) = \sum_{i=1}^{d} w_{ji}\, x_i + w_{j0}$$

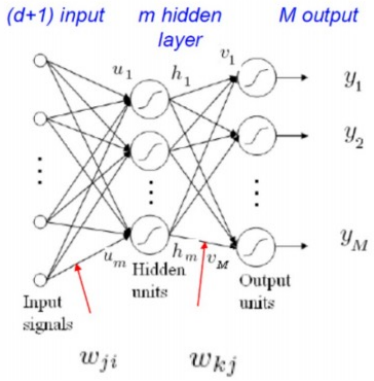


Figure 2: MPL Fuctions and structure

4.2 Mathematical Formulation

The computations in the MLP can be broken down into several layers:

Input to Hidden Layer: Each hidden unit h_j is computed as:

$$u_j(x) = \sum_{i=1}^{d} w_{ji}x_i + w_{j0}$$

where x_i are the input features, w_{ji} are the weights connecting the i-th input to the j-th hidden neuron, and w_{j0} is the bias for the j-th hidden neuron. The activation of the hidden neuron is given by:

$$h_j(x) = g(u_j(x))$$

where $g(\cdot)$ is the activation function, typically a nonlinear function such as the sigmoid, ReLU, or tanh.

Hidden to Output Layer: Each output neuron y_k is computed as:

$$v_k(x) = \sum_{j=1}^{m} w_{kj}h_j + w_{k0}$$

where w_{kj} are the weights connecting the j-th hidden neuron to the k-th output neuron, and w_{k0} is the bias for the k-th output neuron. The output activation is:

$$y_k(x) = g(v_k(x))$$

For classification tasks, a softmax activation function is commonly applied at the output layer.

4.3 Training Objective

The MLP is trained by minimizing a loss function over a dataset $\{(x^n, t^n)\}_{n=1}^N$, where x^n represents the input and t^n represents the corresponding target. The loss function, typically the mean squared error (MSE) or cross-entropy loss, is given by:

$$W = \arg\min_W \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{M}[y_k(x^n) - t_k^n]^2$$

where W represents all the weights in the network.

4.4 Backpropagation and Optimization

The weights of the network are updated using the backpropagation algorithm, which computes the gradient of the loss function with respect to each weight. Optimization algorithms such as stochastic gradient descent (SGD) or Adam are used to adjust the weights to minimize the loss.

5 KNN Classifer

5.1 Structure and Algorithm

K-Nearest Neighbors is a simple and intuitive machine learning algorithm that has been widely used for both classification and regression problems. It is an instance-based learning algorithm that does not have an explicit training process but directly uses the whole training dataset for prediction. KNN classifies a test sample by comparing the test sample to the training samples based on the distance. First, it calculates the distance between the test sample and all training samples and selects the K nearest samples. Then, a vote is conducted among these K neighbors where the majority decides the forecasted label of the test sample. KNN is a nonparametric algorithm that assumes no specific distribution of data; therefore, it offers a high degree of flexibility.

5.2 Mathematical Method and Distance Calculation

The core of the KNN algorithm lies in the distance metric, with Euclidean distance being the most commonly used measure. For any two samples $\mathbf{x} = (x_1, x_2, ..., x_n)$ and $\mathbf{y} = (y_1, y_2, ..., y_n)$, the Euclidean distance is defined as:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Using this formula, KNN can measure the "similarity" between samples. When selecting the K nearest neighbors, the algorithm sorts these distances from smallest to largest and picks the K closest training samples. The algorithm then uses a majority vote among the K neighbors' labels to determine the predicted label. Furthermore, to improve KNN's performance on various datasets, it is common practice to standardize the data (such as Z-score normalization) to ensure that each feature contributes equally to the distance calculation.

5.3 Challenges of KNN

A significant advantage of the KNN algorithm is its simplicity and interpretability. It does not require training a model and directly makes predictions based on the data. However, it also has some drawbacks, especially for large datasets. Calculating the distance between each test sample and all the training samples can be time-consuming, and the computational complexity of KNN increases significantly as the dataset grows. Additionally, KNN is sensitive to the scale of the data, which is why standardization is necessary to avoid biases due to differences in feature scales. For high-dimensional data, KNN may suffer from the "curse of dimensionality," where the distances between samples become indistinguishable, affecting the algorithm's performance.

Despite these challenges, KNN remains a powerful tool, especially for tasks with small datasets and less complex feature spaces. With appropriate K-value selection and data preprocessing, KNN can perform excellently in many practical applications, such as image recognition and text classification.

6 STEPS OF MY PROGRAM

6.1 TASK1

(1) DATA LOADING AND PREPROCESSING

The LFW dataset is loaded using the LFW function, with individuals having at least 70 images included and images resized to 40% of their original dimensions, then split into training and testing sets with 10% allocated for testing, and a few images along with their labels are visualized.

```python
# load the LFW data (Check the loading function and provide appropriate parameters)
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
X = lfw_people.data
X_plot = lfw_people.images
y = lfw_people.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, stratify=y, random_state=42)
target_names = lfw_people.target_names
num_classes = max(y)+1

# Define how many images you want to display
n_images = 4

# Create a grid plot with matplotlib
fig, ax = plt.subplots(1, n_images, figsize=(15, 10))
for i in range(n_images):
    ax[i].imshow(X_plot[i], cmap='gray')  # Display images in grayscale
    ax[i].set_title(target_names[y[i]])  # Set the title to the person's name
    ax[i].axis('off')  # Turn off axis numbering
plt.show()
```

Figure 3: Data Loading and Preprocessing

(2) RESTRICTED BOLTZMANN MACHINE INITIALIZATION

The RBM model initiates its parameters by assigning small random values drawn from a normal distribution to the weights, as illustrated in the provided code snippet. Concurrently, the biases for both the visible and hidden units are set to zero, ensuring the model commences training with unbiased weights, while the introduction of a modest degree of randomness facilitates the learning process.

```python
class RestrictedBoltzmannMachine:
    def __init__(self, num_visible, num_hidden):
        # First, initialize the weights and biases.
        # Typically, weights are initialized to small random values, and biases can be initialized to zero.
        self.num_visible = num_visible
        self.num_hidden = num_hidden
        self.weights = np.random.normal(0, 0.1, size=(num_visible, num_hidden))
        self.visible_bias = np.zeros(num_visible)
        self.hidden_bias = np.zeros(num_hidden)
    # Define the sigmoid function
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
    # Implement a step of the Contrastive Divergence (CD) algorithm to update the weights and biases.
    def contrastive_divergence(self, batch, learning_rate, k=1):
        positive_hidden_probabilities = self.sigmoid(np.dot(batch, self.weights) + self.hidden_bias)
        positive_hidden_states = (
                    positive_hidden_probabilities > np.random.rand(*positive_hidden_probabilities.shape)).astype(float)
        negative_visible_probabilities = self.sigmoid(
            np.dot(positive_hidden_states, self.weights.T) + self.visible_bias)
        negative_hidden_probabilities = self.sigmoid(
            np.dot(negative_visible_probabilities, self.weights) + self.hidden_bias)

        weight_update = np.dot(batch.T, positive_hidden_probabilities) - np.dot(negative_visible_probabilities.T,
                                                                                negative_hidden_probabilities)
        self.weights += learning_rate * weight_update / batch.shape[0]

        self.visible_bias += learning_rate * np.mean(batch - negative_visible_probabilities, axis=0)
        self.hidden_bias += learning_rate * np.mean(positive_hidden_probabilities - negative_hidden_probabilities,
                                                    axis=0)
        error = np.mean((batch - negative_visible_probabilities) ** 2)
        return error
```

Figure 4: Data Loading and Preprocessing

```python
# Initialize and train the RBM model
num_visible = X_train.shape[1]
num_hidden = 256  # 设置隐藏单元数量，可以根据需要调整
rbm = RestrictedBoltzmannMachine(num_visible, num_hidden)
```

Figure 5: Data Loading and Preprocessing

(3) Divergence Training

Forward Pass

In the forward pass, the activation probabilities of the hidden units are computed using the sigmoid function applied to the weighted sum of the visible units and the hidden biases:

$$p(h_j = 1|x) = \text{sigmoid}(b_j + W_j \cdot x)$$

Negative Phase (Reconstruction)

In the negative phase, the visible units are reconstructed from the hidden states. The activation probabilities of the visible units are computed using the sigmoid function applied to the weighted sum of the hidden units and the visible biases:

$$p(v_k = 1|h) = \text{sigmoid}(c_k + W_k^T \cdot h)$$

Weight and Bias Updates

The weights and biases are updated to minimize the reconstruction error. This is achieved by adjusting the weights based on the difference between the expected values of the visible-hidden connections in the positive and negative phases:

$$\Delta W = \eta(\langle vh \rangle_{\text{data}} - \langle vh \rangle_{\text{model}})$$

The biases for the visible and hidden units are updated in a similar manner based on their respective reconstruction differences.

```python
    def train(self, data, learning_rate=0.01, num_epochs=3000, batch_size=32):
            np.random.shuffle(data)
            total_batches = 0
            total_error = 0
            for i in range(0, data.shape[0], batch_size):
                batch = data[i:i + batch_size]
                total_batches += 1
                total_error += self.contrastive_divergence(batch, learning_rate)

            avg_error = total_error / total_batches
            if (epoch + 1) % 10 == 0:
                print(f"Epoch {epoch + 1}, Average Reconstruction Error: {avg_error}")

                # show figure dynamicly
                if epoch + 1 in [100, 500, 1000, 2000]:
                    sample_reconstructed = self.reconstruct(data[:5])
                    print(sample_reconstructed)

                    plot_images(data[:5], sample_reconstructed, num_images=5)

    # Define the process calculating the probabilities for hidden nodes.

    def extract(self, data):
        hidden_probabilities = self.sigmoid(np.dot(data, self.weights) + self.hidden_bias)
        return hidden_probabilities

    def reconstruct(self, data):
        hidden_probabilities = self.extract(data)
        hidden_states = (hidden_probabilities > np.random.rand(*hidden_probabilities.shape)).astype(float)
        reconstructed_data = self.sigmoid(np.dot(hidden_states, self.weights.T) + self.visible_bias)
        return reconstructed_data
```

Figure 6: Divergence Training and Model Training

(4) Training the RBM

The RBM is trained over multiple epochs using a specified learning rate and batch size. During training, the model iteratively adjusts its weights and biases to minimize the reconstruction error. The key parameters are as follows:

- **Learning rate**: Controls the step size for weight and bias updates.
- **Number of epochs**: Specifies the total training iterations.
- **Batch size**: Defines the number of samples per training step.

(5) Reconstruction and Visualization

After training, the RBM reconstructs test images by propagating them through the network. The reconstruction is used to evaluate how well the model captures the underlying structure of the input data.

The original and reconstructed images are displayed side-by-side for visual comparison. This allows

6.2 Task2

(1) Loading Dataset

The LFW (Labeled Faces in the Wild) dataset is loaded using the following code, then the features and labels are extracted.

```
# Load the LFW dataset
lfw_data = datasets.fetch_lfw_people(min_faces_per_person=70, resize=0.4)
features = lfw_data.data
labels = lfw_data.target
```

Figure 7: Loading Dataset

(2) RBM Feature Extractor Class

`RBMFeatureExtractor` class is defined to extract features from the data using a pre-trained RBM model, and this class uses an existing RBM model (trained at task1) to extract features from the dataset.

```
class RBMFeatureExtractor:
    def __init__(self, rbm_model):
        self.rbm_model = rbm_model

    def extract_features(self, data):
        features = self.rbm_model.extract(data)
        return features
```

Figure 8: RBM Feature Extractor Class

(3) Training the MLP Classifier

The `train_mlp_classifier` function is defined to train a Multi-Layer Perceptron (MLP) classifier, that splits the dataset into a training and validation set, then trains the MLP model and returns the predictions for evaluation.

```python
def train_mlp_classifier(features, labels, hidden_dim):
    from sklearn.neural_network import MLPClassifier
    mlp = MLPClassifier(hidden_layer_sizes=(hidden_dim,), max_iter=10000, random_state=33)
    X_train, X_val, y_train, y_val = train_test_split(features, labels, test_size=0.1, random_state=33)
    mlp.fit(X_train, y_train)
    y_pred = mlp.predict(X_val)
    return y_val, y_pred
```

Figure 9: Training the MLP Classifier

(4) Training the KNN Classifier

Similarly, the `train_knn_classifier` function is defined to train a K-Nearest Neighbors (KNN) classifier, this function uses a KNN classifier with a specified number of neighbors and scales the data before training.

```python
def train_knn_classifier(features, labels, n_neighbors=3):
    # Train
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    X_train, X_val, y_train, y_val = train_test_split(features, labels, test_size=0.1, random_state=33)

    # Standardize
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_val = scaler.transform(X_val)

    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_val)

    return y_val, y_pred
```

Figure 10: Training the MLP Classifier

(5) Evaluating Classifier Performance

The `evaluate_classifier` function computes the performance metrics. This function calculates accuracy, precision, recall, and F1-score for the classifier's predictions on the validation set.

```python
def evaluate_classifier(true, pred, average='weighted'):
    acc = accuracy_score(true, pred)
    pre = precision_score(true, pred, average=average)
    recall = recall_score(true, pred, average=average)
    f1 = f1_score(true, pred, average=average)
    print(f"Accuracy: {acc}")
    print(f"Precision: {pre}")
    print(f"Recall: {recall}")
    print(f"F1 Score: {f1}")
```

Figure 11: Evaluating Classifier Performance

(6) Feature Extraction and Training

Firstly, the pre-trained RBM model is used to extract features from the dataset. Then, the MLP and KNN classifiers are trained. Finally, the performance of both models is evaluated using the `evaluate_classifier` function:

```python
feature_extractor = RBMFeatureExtractor(rbm)  # Using q1 trained rbm
features = feature_extractor.extract_features(features)

# Train MLP
hidden_dim = 2048
y_val, mlp_predictions = train_mlp_classifier(features, labels, hidden_dim)
print("-----------------------")
# Evaluate MLP
print("MLP Model Performance:")
evaluate_classifier(y_val, mlp_predictions)
print("-----------------------")

# Train KNN
y_val, knn_predictions = train_knn_classifier(features, labels)
# Evaluate KNN
print("KNN Model Performance:")
evaluate_classifier(y_val, knn_predictions)
print("-----------------------")
```

Figure 12: Feature Extraction and Training

(7) Visualizations of Classify Results

Using the matplotlib library to draw images. The horizontal coordinate is the sample number, and the vertical coordinate is the predicted label value. I drew two pictures. The first picture draws the results of the two classifiers on one image, which can be used to compare the performance of the model. The second picture draws the results of the two classifiers on two different images.

7 Analysis of Results

7.1 Task1

The original images of four LFW datsets images are shown at blow.

Figure 13: Original Images

Then, I used the LFW dataset for RBM training and displayed the reconstructed images after 100, 500, 1000, 2000, and 3000 training cycles. The reconstructed image and loss image after 3000 iterations will be displayed on the next two pages.

Training only 100 rounds of RBM, the reconstructed image is full of noise points and cannot express the facial features of the face, basically unable to distinguish the identity information of the face from the original image.

After 500 rounds of training with RBM, the reconstructed image is still full of significant noise points, barely able to express the facial features of the face, and it can be found that the reconstructed image has similarities with the original image.

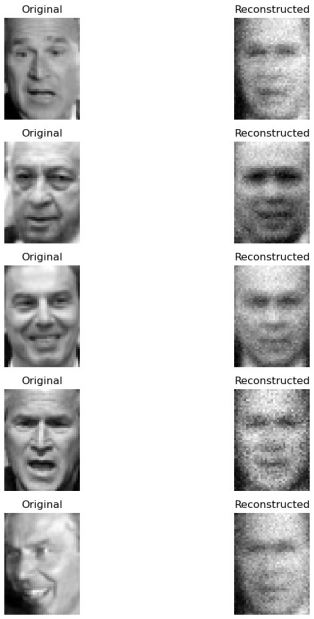


Figure 14: Reconstructed Image after 100 Epoches

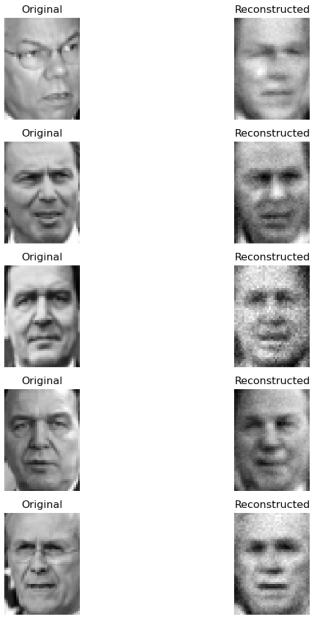


Figure 15: Reconstructed Image after 500 Epoches

After 1000 rounds of training with RBM, the reconstructed image is filled with a small number of noise points, which can express the facial features of the face. It can be found that the reconstructed image has many similarities with the original image.

After 2000 rounds of training with RBM, the reconstructed images produced significantly fewer noise points, which could better express the facial features of the face. It was found that the reconstructed images had many similar features to the original image of the face.

After 3000 rounds of training with RBM, the reconstructed images produced fewer outliers, and the ability to express facial features of the face was not significantly improved compared to 2000 rounds.

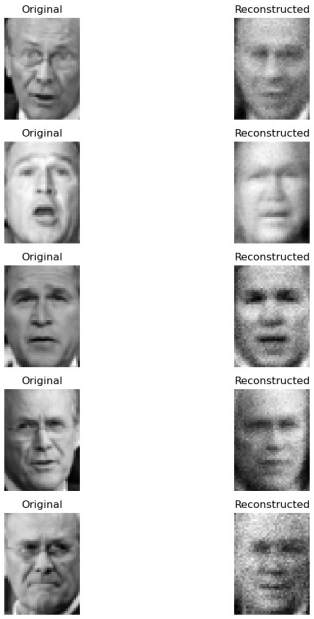


Figure 16: Reconstructed Image after 1000 Epoches

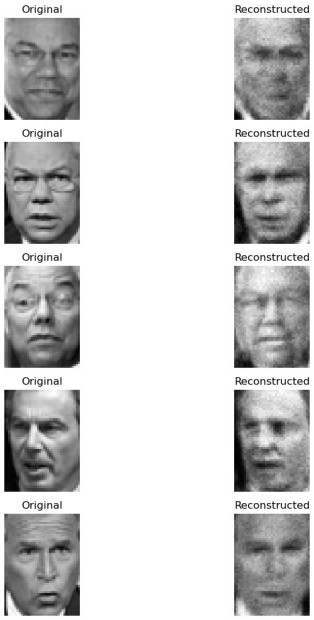


Figure 17: Reconstructed Image after 2000 Epoches

Moreover, it was found that the reconstructed images had many similar features to the original image of the face.

From the loss image, it can be seen that with training, the loss continues to decrease and has not yet converged. However, continuing training has limited improvement in the model's ability to extract features, and some of the images generated in 3000 rounds have already shown distortion.

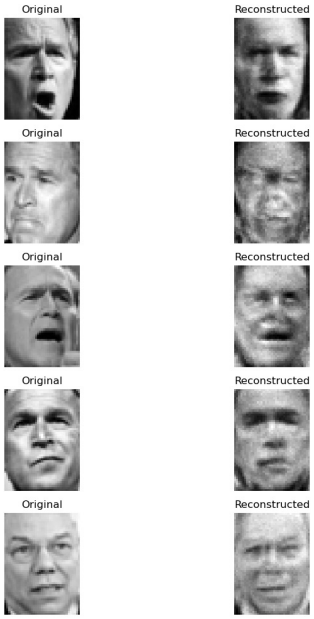


Figure 18: Reconstructed Image after 3000 Epoches

```
Epoch 2880, Average Reconstruction Error: 0.00954051586529421
Epoch 2890, Average Reconstruction Error: 0.009436408891568642
Epoch 2900, Average Reconstruction Error: 0.009490721355969475
Epoch 2910, Average Reconstruction Error: 0.009441852246745505
Epoch 2920, Average Reconstruction Error: 0.00946828856221332
Epoch 2930, Average Reconstruction Error: 0.009414993703091259
Epoch 2940, Average Reconstruction Error: 0.009364909579572863
Epoch 2950, Average Reconstruction Error: 0.00933737021034398
Epoch 2960, Average Reconstruction Error: 0.00933185798276201
Epoch 2970, Average Reconstruction Error: 0.009315030952892133
Epoch 2980, Average Reconstruction Error: 0.009253132646587248
Epoch 2990, Average Reconstruction Error: 0.00933734748274452
Epoch 3000, Average Reconstruction Error: 0.009258994532109141
```

Figure 19: Training Loss Images

7.2 TASK2

MLP model and KNN model performance shown at Figure20.

The MLP model achieves the following results:

- **Accuracy:** 0.81, indicating good overall classification performance.
- **Precision:** 0.81, meaning the model is correct 81% of the time when predicting positive classes.
- **Recall:** 0.81, meaning the model identifies 81% of relevant instances.
- **F1 Score:** 0.80, showing a balanced performance between precision and recall.

The KNN model has the following results:

- **Accuracy:** 0.54, indicating poor performance, with only half of the samples correctly classified.
- **Precision:** 0.56, showing a lower reliability in positive class predictions.
- **Recall:** 0.54, meaning it identifies only 54% of relevant instances.

- **F1 Score:** 0.53, reflecting a lower overall performance compared to the MLP model.

```
y_val, mlp_predictions = train_mlp_classifier(features, labels, hidden_dim)
print("-----------------------")
# Evaluate MLP
print("MLP Model Performance:")
evaluate_classifier(y_val, mlp_predictions)
print("-----------------------")

# Train KNN
y_val, knn_predictions = train_knn_classifier(features, labels)
# Evaluate KNN
print("KNN Model Performance:")
evaluate_classifier(y_val, knn_predictions)
print("-----------------------")
```

✓ [54] 20s 233ms

```
-----------------------
MLP Model Performance:
Accuracy: 0.8062015503875969
Precision: 0.8118955334071614
Recall: 0.8062015503875969
F1 Score: 0.7980967686239093
-----------------------
KNN Model Performance:
Accuracy: 0.5426356589147286
Precision: 0.559722971779777
Recall: 0.5426356589147286
F1 Score: 0.5287743720028262
-----------------------
```

Figure 20: Task2 Running Result

We can get the conclusion that the MLP model outperforms the KNN model across all evaluation metrics. KNN's lower accuracy and F1 score may result from its sensitivity to data scale, the choice of K, and challenges in handling high-dimensional data.

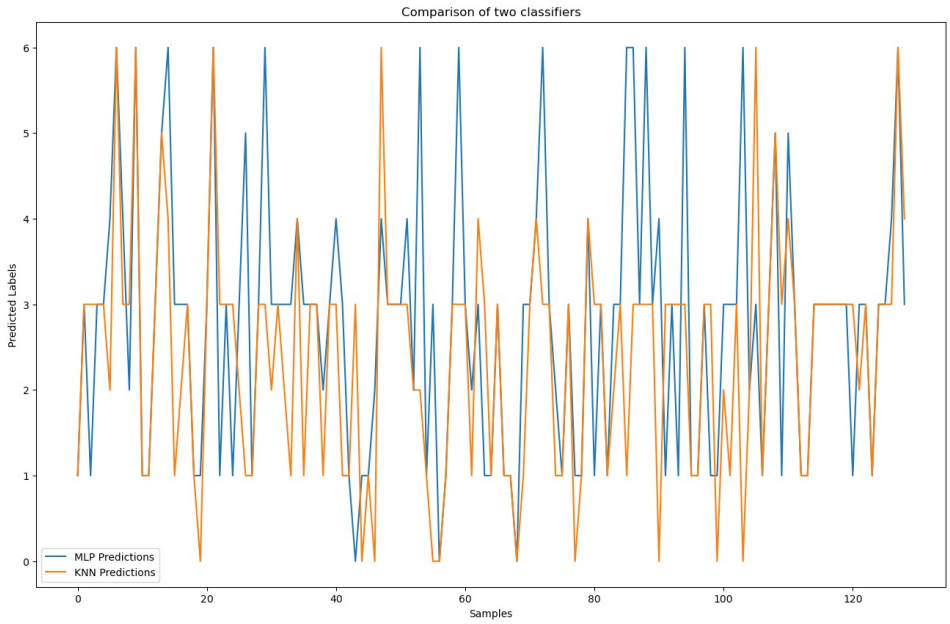


Figure 21: Comparison of two classifiers

Figure21 provides a comparative analysis of the predictions made by the Multi-Layer Perceptron (MLP) and K-Nearest Neighbors (KNN) classifiers on the same dataset. In the plot, the MLP predictions, represented by the blue line, exhibit considerable fluctuations, with predicted labels ranging

from 1 to 6, suggesting high variability. In contrast, the KNN predictions, depicted by the orange line, demonstrate a more stable trend, with predicted labels fluctuating primarily between 2 and 4. While both classifiers show agreement in certain regions of the plot, significant discrepancies are observed in others, emphasizing the different prediction patterns of the two classifiers.

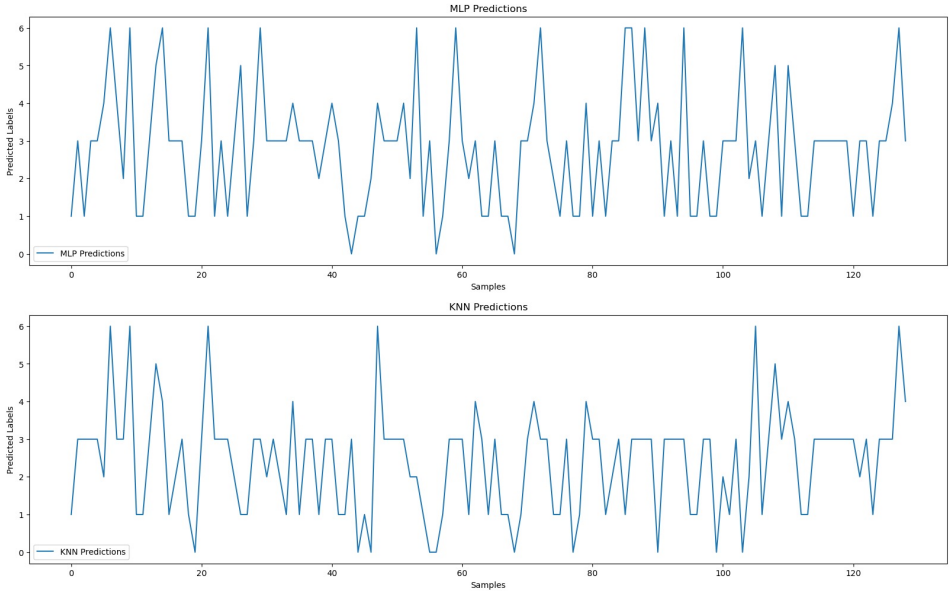


Figure 22: Comparison of two classifiers

Figure22 presents the prediction outputs of the Multi-Layer Perceptron (MLP) and K-Nearest Neighbors (KNN) classifiers separately for the given dataset. The MLP predictions exhibit frequent and rapid fluctuations, with predicted values ranging between labels 1 and 6, reflecting a highly variable prediction pattern. Similarly, KNN predictions also demonstrate fluctuations, but the predicted values are more concentrated, primarily oscillating between labels 1 and 3. Both classifiers display noticeable volatility in their predictions; however, KNN shows a tendency to predict lower label values compared to MLP.

The results from the experiments highlight the strengths and weaknesses of the two classifiers, Multi-Layer Perceptron (MLP) and K-Nearest Neighbors (KNN), in handling the given dataset.

The MLP classifier outperformed KNN across all metrics, achieving an accuracy of 81% and an F1 score of 0.80, indicating its ability to model complex data patterns effectively.In contrast, the KNN classifier showed weaker performance, with an accuracy of only 54%, reflecting its limitations in handling high-dimensional data and sensitivity to feature scaling.

In conclusion, for the given dataset and classification task, the MLP model is the better-performing algorithm. However, careful consideration of computational efficiency and task-specific requirements is essential when selecting a classifier, especially when working with high-dimensional data.

8 CONCLUSION

In a nutshell, this project researches in-depth the new advanced pattern recognition techniques based on the LFW database and two most important implementations of the Restricted Boltzmann Machine and Multi-Layer Perceptron classifiers. After conducting a series of systematic experiments and performance evaluations of the model, we have reaped several useful lessons from the experiment and the features of this model for real-world face recognition applications.

Our results show that though RBM is simple, it could extract meaningful features from facial images, necessary for effective reconstruction of images. Notably, the number of hidden units and the period of training make much difference to the capability of RBM to capture the underlying structure in data. Further, in comparison, MLP classifiers proved dominance over KNN on accuracy, precision, recall, and F1 score. It would indicate that, on a particular dataset and classification problem, the complexity of the data is modeled more properly by the MLP model.

However, it is also important to realize the limitations in this lab. The small sample size in this project may not reflect reality in the real diverse world of facial recognition. Moreover, there may exist more optimal hyperparameters for RBM and the choices of classifiers to possibly do better.

This assignment contributes beyond mere academic research into a deeper understanding of unsupervised feature learning and perhaps its application in facial recognition systems. Future work may involve the fining of model architectures, exploration of different hyperparameter settings, and testing these models on larger and more varied datasets.

Thanks for Dr.Yang's guidance and the answers of TA to questions and problems proposed in the class of Advanced Pattern Recognition.

APPENDIX

Here is my code for INT410 final coursework, you can find it on my Github and I also attach it at Google Colab.

You can access the repository at INT-410-Coursework on GitHub.

You can run my program at colab 410 final.