CHAPTER 5
IMPLEMENTATION AND RESULTS

5.1. Implementation

For the DNN algorithm, we utilize the sklearn.datasets library to fetch the ORL dataset. Line number 1 is used to import the Olivetti faces from the sklearn.dataset library. The parameter of return_X_y is set to true, which returns (data, target) instead of a Bunch object. The following line stores the data fetched to the variables X and y.

```
1  from sklearn.datasets import fetch_olivetti_faces
2  X,y = fetch_olivetti_faces(return_X_y=True)
```

The next step is to split the dataset into training and test sets. Here we give the parameter X and y, which contains the data and target. The test_size is going to be varied. The first proportion is 0.1, then 0.2, 0.3, and lastly, 0.4. Set the stratify and random state to improve the precision of the sample and control the shuffling before applying the split.

```
3  train_X, test_X, train_y, test_y = train_test_split(X, y,
   test_size=0.20, stratify=y, random_state=42)
```

Scale the data simply by passing the train_X and test_X variable to preprocessing.scale command.

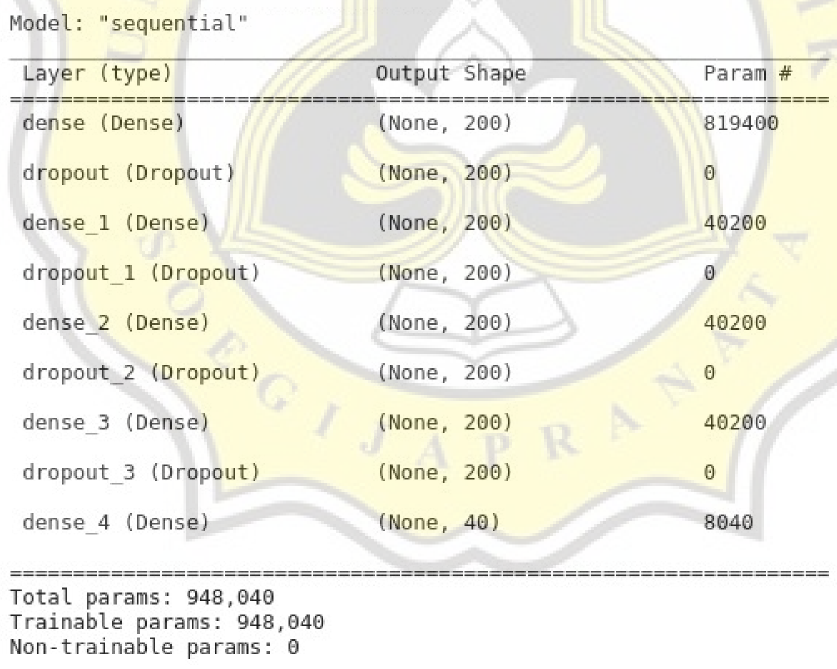
```
4  train_X = preprocessing.scale(train_X)
5  test_X = preprocessing.scale(test_X)
```

Then, create the model architecture. The model used is the sequential model with softmax and ReLU activation functions. Here we add the layers one by one. First, we add the dense layer. The dense layer is the regular deeply connected neural network layer most commonly used for neural networks. The Dense layer supplies all the outputs from the previous layer to all its neurons, and each neuron provides the output to the next layer.

The unit parameter of the dense means the dimensionality of the output space is 200. The input dimension of the first dense layer is 4096, which is the total number of pixels from the dataset, a face image with a size of 64 x 64 pixels. Kernel regularizer is a function applied to the kernel of the weights matrix. Input parameter l2 means that we use the L2 regularization penalty. The L2 regularization penalty is computed as: loss = l2 * reduce_sum(square(x)).

The next layer is the dropout layer. The Dropout layer is randomly sets input units to 0 at each step during training time. After the model is done, it is compiled with the rmsprop optimizer. For the loss function, this project uses sparse categorical crossentropy. It is the default loss function for multi-class classification problems where each class is assigned a unique integer value from 0 to (num_classes – *1*). The last parameter of this compile command is metrics. These metrics contain a list of metrics to be evaluated by the model during training and testing.

```
model = Sequential([
Dense(units=200,    input_dim=4096,    kernel_regularizer=l2(0.0001),
activation='relu'),
Dropout(0.2),
Dense(units=200,    input_dim=200,    kernel_regularizer=l2(0.0001),
activation='relu'),
Dropout(0.2),
Dense(units=200,    input_dim=200,    kernel_regularizer=l2(0.0001),
activation='relu'),
Dropout(0.1),
Dense(units=200,    input_dim=200,    kernel_regularizer=l2(0.0001),
activation='relu'),
Dropout(0.1),
Dense(units=40, input_dim=200, activation='softmax'),])
model.compile(loss='sparse_categorical_crossentropy',
optimizer='rmsprop',
metrics=['accuracy'])
```

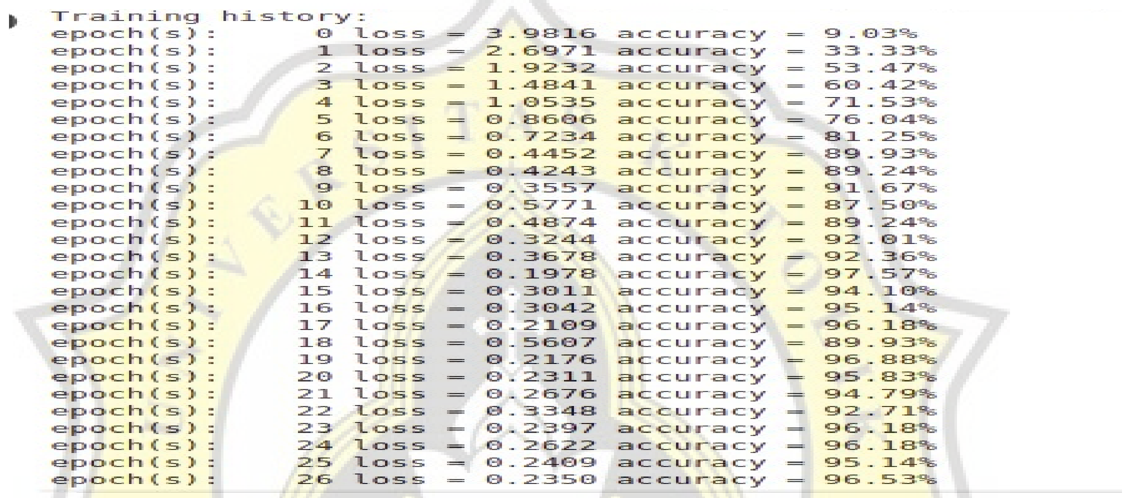


```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 200)               819400

 dropout (Dropout)           (None, 200)               0

 dense_1 (Dense)             (None, 200)               40200

 dropout_1 (Dropout)         (None, 200)               0

 dense_2 (Dense)             (None, 200)               40200

 dropout_2 (Dropout)         (None, 200)               0

 dense_3 (Dense)             (None, 200)               40200

 dropout_3 (Dropout)         (None, 200)               0

 dense_4 (Dense)             (None, 40)                8040

=================================================================
Total params: 948,040
Trainable params: 948,040
Non-trainable params: 0
```

Figure 5.1: DNN Model Summary

After that, train the model and count the time consumed to run this code. This training step will be repeated by changing the dataset split ratio parameters. First, apply the 90:10 ratio by passing 0.1 to the validation split parameter. In the same way, use the 80:20, 70:30, and 60:40 ratios. Evaluate the model for each training.

```
h  =  model.fit(train_X,  train_y,  batch_size=50,  epochs=num_epochs,
validation_split = 0.2, verbose=0)
finish = time.perf_counter()
print(f"Training finished in {finish - start:0.4f} seconds\n")
print("Training history: ")
for i in range(num_epochs):
los = h.history['loss'][i]
acc = h.history['accuracy'][i] * 100
print("epoch: %5d loss = %0.4f acc = %0.2f%%" \
% (i, los, acc))

eval = model.evaluate(test_X, test_y, verbose=0)
print("\nEvaluation on test data: \nloss = %0.4f \
accuracy = %0.2f%%" % (eval[0], eval[1]*100) )
```



```
Training history:
epoch(s):      0 loss = 3.9816 accuracy = 9.03%
epoch(s):      1 loss = 2.6971 accuracy = 33.33%
epoch(s):      2 loss = 1.9232 accuracy = 53.47%
epoch(s):      3 loss = 1.4841 accuracy = 60.42%
epoch(s):      4 loss = 1.0535 accuracy = 71.53%
epoch(s):      5 loss = 0.8606 accuracy = 76.04%
epoch(s):      6 loss = 0.7234 accuracy = 81.25%
epoch(s):      7 loss = 0.4452 accuracy = 89.93%
epoch(s):      8 loss = 0.4243 accuracy = 89.24%
epoch(s):      9 loss = 0.3557 accuracy = 91.67%
epoch(s):     10 loss = 0.5771 accuracy = 87.50%
epoch(s):     11 loss = 0.4874 accuracy = 89.24%
epoch(s):     12 loss = 0.3244 accuracy = 92.01%
epoch(s):     13 loss = 0.3678 accuracy = 92.36%
epoch(s):     14 loss = 0.1978 accuracy = 97.57%
epoch(s):     15 loss = 0.3011 accuracy = 94.10%
epoch(s):     16 loss = 0.3042 accuracy = 95.14%
epoch(s):     17 loss = 0.2109 accuracy = 96.18%
epoch(s):     18 loss = 0.5607 accuracy = 89.93%
epoch(s):     19 loss = 0.2176 accuracy = 96.88%
epoch(s):     20 loss = 0.2311 accuracy = 95.83%
epoch(s):     21 loss = 0.2676 accuracy = 94.79%
epoch(s):     22 loss = 0.3348 accuracy = 92.71%
epoch(s):     23 loss = 0.2397 accuracy = 96.18%
epoch(s):     24 loss = 0.2622 accuracy = 96.18%
epoch(s):     25 loss = 0.2409 accuracy = 95.14%
epoch(s):     26 loss = 0.2350 accuracy = 96.53%
```

Figure 5.2: Training History

Now after the DNN algorithm is implemented, it is time to implement the PCA algorithm. The first thing to do is to split the dataset into the training and testing set. After that, find the mean face of the training set.

```
mean_face = np.zeros((1,height*width))
print(mean_face)

for i in training:
mean_face = np.add(mean_face,i)

mean_face = np.divide(mean_face,len(train_images)).flatten()
```

Then, find the normalized matrix by subtracting each face of the training set from the mean face. This normalized image contains only the unique features of the faces.

```
for i in range(len(train_images)):
normalised_training[i] = np.subtract(training[i],mean_face)
```

The next step is to find the eigenvalues and eigenvectors. To find these values, we need to count the covariance of the normalized image first. The covariance matrix is a square matrix denoting the covariance of the elements with each other.

```
cov_matrix = np.cov(normalised_training)
eigenvalues, eigenvectors, = np.linalg.eig(cov_matrix)
```

Sort the Eigenvalues in the descending order and the corresponding Eigenvector to arrange the principal component in descending order of their variability. Select K number of Eigenfaces to reduce the data to K number variables.

```
eigen_pairs = [(eigenvalues[index], eigenvectors[:,index]) for index in range(len(eigenvalues))]
eigen_pairs.sort(reverse=True)
sort_eigenvalues = [eigen_pairs[index][0] for index in range(len(eigenvalues))]
sort_eigvectors = [eigen_pairs[index][1] for index in range(len(eigenvalues))]
reduced_data = np.array(sort_eigvectors[:7]).transpose()
```

Then, transform the data by dot the transposed reduced data and training data. Then transpose the result. By transposing the outcome of the dot product, the data is reduced to lower dimensions from higher dimensions.

```
proj_data = np.dot(training.transpose(),reduced_data)
proj_data = proj_data.transpose()
```

Find the weight for each normalized data. This weight tells us how important that particular Eigenface is in contributing to the mean face.

```
w = np.array([np.dot(proj_data,i) for i in normalised_training])
```

Finally, recognize all the test images and analyze the accuracy of the recognition results.

```
def recogniser(img, train_images,proj_data,w):
global count,highest_min,num_images,correct_pred
unknown_face = plt.imread('drive/MyDrive/orl/'+img)
num_images += 1
unknown_face_vector = np.array(unknown_face, dtype='float64').flatten()
normalised_uface_vector = np.subtract(unknown_face_vector,mean_face)
w_unknown = np.dot(proj_data, normalised_uface_vector)
diff = w - w_unknown
norms = np.linalg.norm(diff, axis=1)
index = np.argmin(norms)
min(norms)
t1 = 99999999999
t0 = 99999999999
```

```
if norms[index] < t1:
plt.subplot(80,10,1+count)
if norms[index] < t0: # Face is found
if img.split('_')[1] == train_images[index].split('_')[1]:
plt.title('Matched:'+'.'.join(train_images[index].split('.')[:2]),
color='g')
plt.imshow(imread('drive/MyDrive/orl/'+train_images[index]),
cmap='gray')
correct_pred += 1
else:
plt.title('Mismatched:'+'.'.join(train_images[index].split('.')[:2]),
color='b')
plt.imshow(imread('drive/MyDrive/orl/'+train_images[index]),
cmap='gray')
plt.subplots_adjust(right=1.2, top=2.5)
count+=1
fig = plt.figure(figsize=(15, 15))
for i in range(len(test_images)):
recogniser(test_images[i], train_images,proj_data,w)
```

5.2. Results

The experiments are done several times with different dataset split ratio parameters for both algorithms. All of the results can be seen from the tables below.

Table 5.1: DNN Algorithm Results

	Iteration	Loss	Accuracy
Training : 90% Testing : 10%	I	0.2729	97.50%
	II	0.1495	97.50%
	III	0.7175	90.00%
	IV	0.6295	92.50%
	V	0.5469	95.00%
	Accuracy average :		94.50%
Training : 80% Testing : 20%	I	0.5062	92.50%
	II	0.2632	97.50%
	III	0.6995	92.50%
	IV	0.2753	95.00%
	V	0.5852	90.00%
	Accuracy average :		93.50%
Training : 70% Testing : 30%	I	0.4724	90.00%
	II	0.1589	97.50%
	III	0.5163	92.50%
	IV	0.9885	92.50%
	V	2.4577	85.00%
	Accuracy average :		91.50%
Training : 60% Testing : 40%	I	0.9663	92.50%
	II	0.2881	95.00%
	III	0.7447	90.00%
	IV	0.9142	87.50%
	V	1.2623	87.50%
	Accuracy average :		90.50%
Training : 50% Testing : 50%	I	0.8710	91.50%
	II	1.2444	87.50%
	III	1.1506	87.00%
	IV	1.1535	88.00%
	V	1.4483	88.00%
	Accuracy average :		88.40%

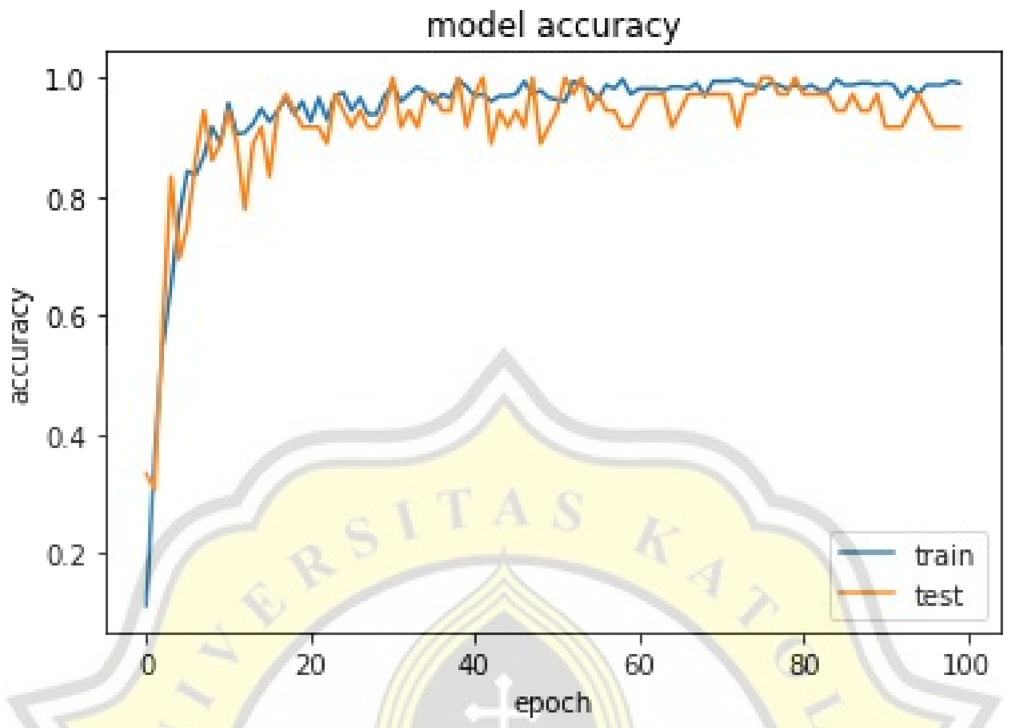


Figure 5.3: DNN Accuracy Model

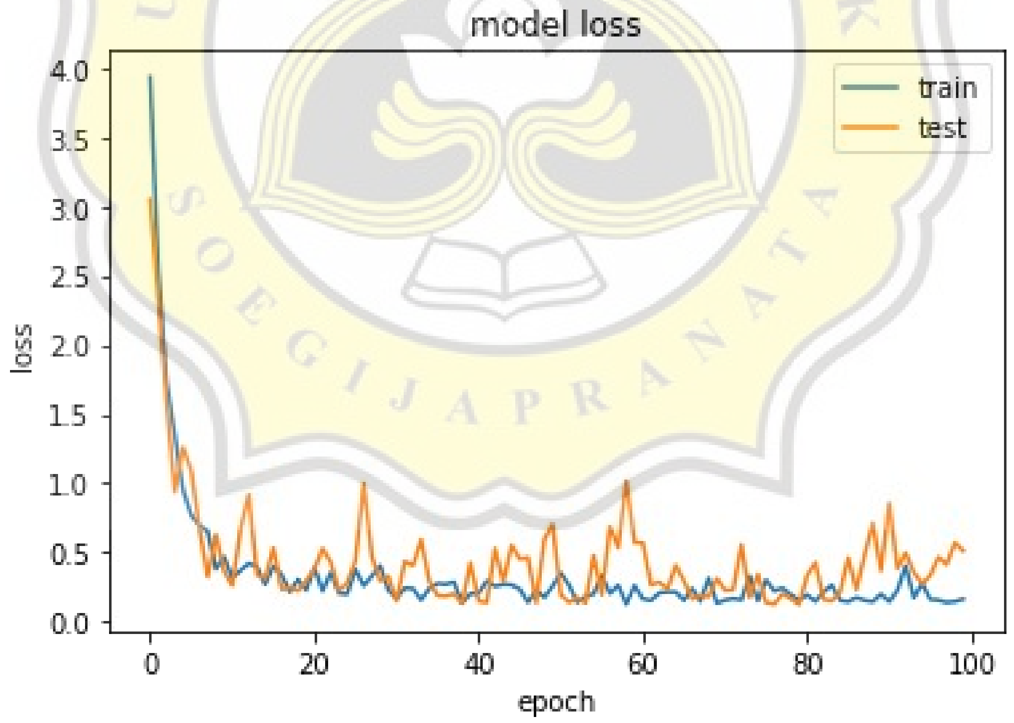


Figure 5.4: DNN Loss Model

The results above show that the larger the training data, the more accurate the DNN algorithm will be, so we can say that the DNN would be more optimum when the training data is more extensive. The highest accuracy is obtained from experiments with training ratios of 90% and testing of 10%. The accuracy values of these experiments varied from 90% to 97.50%, and the accuracy average after five trials is 94.50%. The accuracy and loss model shows that our model is not underfitting nor overfitting because the train and test are correlated.

Table 5.2: PCA Algorithm Result

	K	Correct Prediction	Accuracy (correct prediction / total test image x 100%)
Training : 90%	5	31	31 / 40 = 77.50%
	7	33	33 / 40 = 82.50%
	10	34	34 / 40 = 85.00%
	12	36	36 / 40 = 90.00%
	16	38	38 / 40 = 95.00%
Training : 80%	5	32	32 / 40 = 80.00%
	7	33	33 / 40 = 82.50%
	10	34	34 / 40 = 85.00%
	12	36	36 / 40 = 90.00%
	16	38	38 / 40 = 95.00%
Training : 70%	5	28	28 / 40 = 70.00 %
	7	30	30 / 40 = 75.00 %
	10	34	34 / 40 = 85.00%
	12	36	36 / 40 = 90.00%
	16	38	38 / 40 = 95.00%
Training : 60%	5	27	27 / 40 = 67.50%
	7	31	31 / 40 = 77.50%
	10	35	35 / 40 = 87.50%
	12	36	36 / 40 = 90.00%
	16	38	38 / 40 = 95.00%
Training : 50%	5	30	30 / 40 = 75.00%
	7	32	32 / 40 = 80.00%
	10	36	36 / 40 = 90.00%
	12	36	36 / 40 = 90.00%
	16	37	37 / 40 = 92.50%

From the results above, we can see that the PCA algorithm's highest accuracy is 95%. It is also noticed that the value of the K variable influences the accuracy results. However, there is no certain value of K; it depends on the dataset used. The value of this variable must go through trial and error. It is also seen that number of data used for training does not significantly affect the accuracy, so we can say that the PCA algorithm is good to use when the dataset is small.

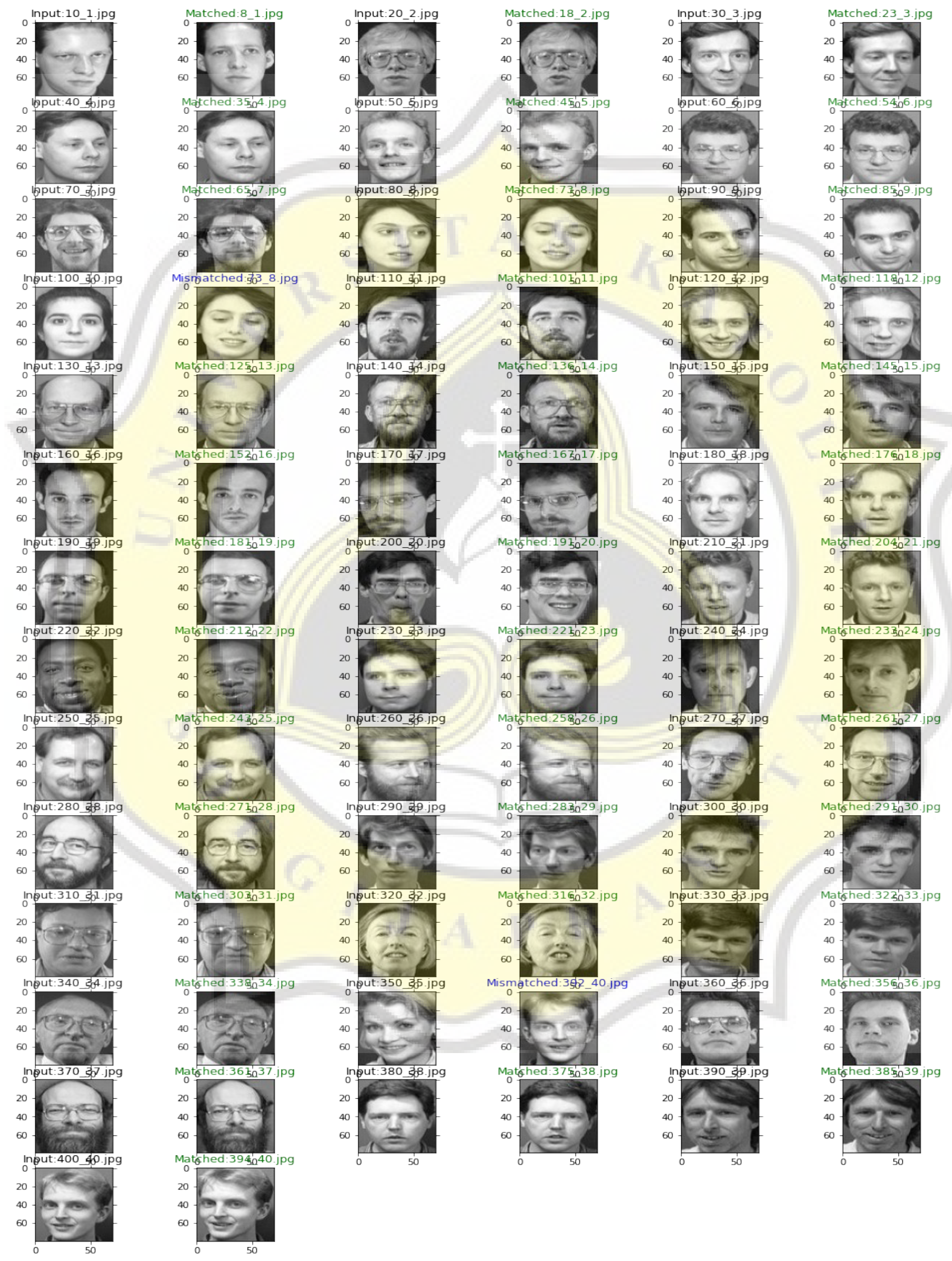


Figure 5.5: Face Recognition with PCA Results

Figure 5.4 is a test with taking 80% of the dataset as training data. We can see that the PCA algorithm achieved an accuracy of 95% with the input of 40 testing images. There are two misrecognized images: a person with id ten who is mistakenly recognized as a person with id eight and a woman with id 35 who is recognized as a man with id 40 instead.

5.3. Comparison

The accuracy of both algorithms is relatively reliable. The minimum accuracy of the DNN algorithm is 87.00%, using 50% training data from the total dataset. Meanwhile, the PCA algorithm works better in certain K values. With 50% training data, the highest accuracy achieved by the DNN algorithms is 91.50%. With the same ratio of training data, the PCA algorithm accuracy can reach 92.5%.

As we can see, the accuracy of the two algorithms is only slightly different. The highest accuracy of the DNN algorithm is 97.50%, and the maximum accuracy of the PCA algorithm is 95%. But there are other factors we need to consider in choosing an algorithm, such as time of implementation. The project's code is compiled with Google Colab, which has 12GB of RAM.

Table 5.3: Highest Accuracy of DNN Algorithm

	Iteration	Loss	Accuracy
Training : 90% Testing : 10%	I	0.2729	97.50%
	II	0.1495	97.50%
	III	0.7175	90.00%
	IV	0.6295	92.50%
	V	0.5469	95.00%
	Accuracy average :		94.50%

Table 5.4: Highest Accuracy of PCA Algorithm

	K	Correct Prediction	Accuracy (correct prediction / total test image x 100%)
Training : 90%	16	38	38 / 40 = 95.00%
Training : 80%	16	38	38 / 40 = 95.00%
Training : 70%	16	38	38 / 40 = 95.00%
Training : 60%	16	38	38 / 40 = 95.00%

The DNN algorithm takes about half a minute to train data with 4096 dimensionalities, while the PCA algorithm needs three-quarters of a minute. There is no significant difference in running time because the dataset used is relatively small. But still, the DNN takes less training and testing time than the PCA algorithm. Both algorithms did not have any difficulties in terms of computation due to the small and neat dataset.