Face Recognition using CNN

Step1:

At the first, you should input the required libraries:

```
import keras
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout
from keras.optimizers import Adam
from keras.callbacks import TensorBoard

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

from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import accuracy_score
from keras.utils import np_utils
import itertools
```

Step2:

- Load Dataset :

After loading the Dataset you have to normalize every image.

Note: an image is a Uint8 matrix of pixels and for calculation, you need to convert the format of the image to float or double

```
#load dataset
data = np.load('ORL_faces.npz')

# load the "Train Images"
x_train = data['trainX']
#normalize every image
x_train = np.array(x_train,dtype='float32')/255

x_test = data['testX']
x_test = np.array(x_test,dtype='float32')/255

# load the Label of Images
y_train= data['trainY']
y_test= data['testY']

# show the train and test Data format
print('x_train : {}'.format(x_train[:]))
print('Y-train shape: {}'.format(y_train))
print('x_test shape: {}'.format(x_test.shape))
```

```
x_train : [[0.1882353  0.19215687 0.1764706  ... 0.18431373 0.18039216 0.18039216]
 [0.23529412 0.23529412 0.24313726 ... 0.1254902  0.13333334 0.13333334]
 [0.15294118 0.17254902 0.20784314 ... 0.11372549 0.10196079 0.11372549]
```

```
 ...
 [0.44705883 0.45882353 0.44705883 ... 0.38431373 0.3764706  0.38431373]
 [0.4117647  0.4117647  0.41960785 ... 0.21176471 0.18431373 0.16078432]
 [0.45490196 0.44705883 0.45882353 ... 0.37254903 0.39215687 0.39607844]]
Y-train shape: [ 0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  1  1
  2  2  2  2  2  2  2  2  2  2  2  2  3  3  3  3  3  3  3  3  3  3  3  3
  4  4  4  4  4  4  4  4  4  4  4  4  5  5  5  5  5  5  5  5  5  5  5  5
  6  6  6  6  6  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  7  7  7  7
  8  8  8  8  8  8  8  8  8  8  8  8  9  9  9  9  9  9  9  9  9  9  9  9
 10 10 10 10 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11 11 11 11 11
 12 12 12 12 12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13 13 13 13 13
 14 14 14 14 14 14 14 14 14 14 14 14 15 15 15 15 15 15 15 15 15 15 15 15
 16 16 16 16 16 16 16 16 16 16 16 16 17 17 17 17 17 17 17 17 17 17 17 17
 18 18 18 18 18 18 18 18 18 18 18 18 19 19 19 19 19 19 19 19 19 19 19 19]
x_test shape: (160, 10304)
```

Step 3

Split DataSet : Validation data and Train

Validation DataSet: this data set is used to minimize overfitting.If the accuracy over the training data set increases, but the accuracy over then validation data set stays the same or decreases, then you're overfitting your neural network and you should stop training.

- Note: we usually use 30 percent of every dataset as the validation data but Here we only used 5 percent because the number of images in this dataset is very low.

```
x_train, x_valid, y_train, y_valid= train_test_split(
    x_train, y_train, test_size=.05, random_state=1234,)
```

Step 4

for using the CNN, we need to change The size of images (The size of images must be the same)

```
im_rows=112
im_cols=92
batch_size=512
im_shape=(im_rows, im_cols, 1)

#change the size of images
x_train = x_train.reshape(x_train.shape[0], *im_shape)
x_test = x_test.reshape(x_test.shape[0], *im_shape)
x_valid = x_valid.reshape(x_valid.shape[0], *im_shape)

print('x_train shape: {}'.format(y_train.shape[0]))
print('x_test shape: {}'.format(y_test.shape))
```

```
x_train shape: 228
x_test shape: (160,)
```

Step 5

Build CNN model: CNN have 3 main layer:

- 1-Convolotional layer
- 2- pooling layer
- 3- fully connected layer

  we could build a new architecture of CNN by changing the number and position of layers.

```
#filters= the depth of output image or kernels

cnn_model= Sequential([
    Conv2D(filters=36, kernel_size=7, activation='relu', input_shape= im_shape),
```

```
    MaxPooling2D(pool_size=2),
    Conv2D(filters=54, kernel_size=5, activation='relu', input_shape= im_shape),
    MaxPooling2D(pool_size=2),
    Flatten(),
    Dense(2024, activation='relu'),
     Dropout(0.5),
    Dense(1024, activation='relu'),
    Dropout(0.5),
    Dense(512, activation='relu'),
    Dropout(0.5),
    #20 is the number of outputs
    Dense(20, activation='softmax')
])

cnn_model.compile(
    loss='sparse_categorical_crossentropy',#'categorical_crossentropy',
    optimizer=Adam(lr=0.0001),
    metrics=['accuracy']
)
```

```
/usr/local/lib/python3.8/dist-packages/keras/optimizers/optimizer_v2/adam.py:110: UserWarning: The `lr` argument is deprecated
  super(Adam, self).__init__(name, **kwargs)
```

Show the model's parameters.

```
cnn_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 106, 86, 36)       1800

 max_pooling2d (MaxPooling2D  (None, 53, 43, 36)       0
 )

 conv2d_1 (Conv2D)           (None, 49, 39, 54)        48654

 max_pooling2d_1 (MaxPooling  (None, 24, 19, 54)       0
 2D)

 flatten (Flatten)           (None, 24624)             0

 dense (Dense)               (None, 2024)              49841000

 dropout (Dropout)           (None, 2024)              0

 dense_1 (Dense)             (None, 1024)              2073600

 dropout_1 (Dropout)         (None, 1024)              0

 dense_2 (Dense)             (None, 512)               524800

 dropout_2 (Dropout)         (None, 512)               0

 dense_3 (Dense)             (None, 20)                10260

=================================================================
Total params: 52,500,114
Trainable params: 52,500,114
Non-trainable params: 0
_________________________________________________________________
```

Step 6

Train the Model

- Note: You can change the number of epochs

```
history=cnn_model.fit(
    np.array(x_train), np.array(y_train), batch_size=512,
    epochs=250, verbose=2,
    validation_data=(np.array(x_valid),np.array(y_valid)),
)
```

```
Epoch 1/250
1/1 - 10s - loss: 3.0025 - accuracy: 0.0439 - val_loss: 2.9890 - val_accuracy: 0.0833 - 10s/epoch - 10s/step
Epoch 2/250
1/1 - 9s - loss: 2.9947 - accuracy: 0.0702 - val_loss: 2.9775 - val_accuracy: 0.0833 - 9s/epoch - 9s/step
Epoch 3/250
1/1 - 8s - loss: 3.0263 - accuracy: 0.0658 - val_loss: 2.9745 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 4/250
1/1 - 8s - loss: 2.9759 - accuracy: 0.0789 - val_loss: 2.9739 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 5/250
1/1 - 8s - loss: 2.9693 - accuracy: 0.1009 - val_loss: 2.9740 - val_accuracy: 0.2500 - 8s/epoch - 8s/step
Epoch 6/250
1/1 - 8s - loss: 2.9890 - accuracy: 0.0526 - val_loss: 2.9724 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 7/250
1/1 - 8s - loss: 2.9638 - accuracy: 0.0965 - val_loss: 2.9751 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 8/250
1/1 - 8s - loss: 2.9988 - accuracy: 0.0439 - val_loss: 2.9756 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 9/250
1/1 - 8s - loss: 2.9734 - accuracy: 0.1053 - val_loss: 2.9770 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 10/250
1/1 - 8s - loss: 2.9736 - accuracy: 0.0570 - val_loss: 2.9777 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 11/250
1/1 - 8s - loss: 2.9675 - accuracy: 0.0746 - val_loss: 2.9778 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 12/250
1/1 - 8s - loss: 2.9637 - accuracy: 0.1053 - val_loss: 2.9778 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 13/250
1/1 - 8s - loss: 2.9593 - accuracy: 0.0877 - val_loss: 2.9781 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 14/250
1/1 - 8s - loss: 2.9796 - accuracy: 0.0570 - val_loss: 2.9788 - val_accuracy: 0.1667 - 8s/epoch - 8s/step
Epoch 15/250
1/1 - 8s - loss: 2.9691 - accuracy: 0.0921 - val_loss: 2.9786 - val_accuracy: 0.1667 - 8s/epoch - 8s/step
Epoch 16/250
1/1 - 8s - loss: 2.9512 - accuracy: 0.0877 - val_loss: 2.9762 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 17/250
1/1 - 8s - loss: 2.9501 - accuracy: 0.0877 - val_loss: 2.9708 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 18/250
1/1 - 8s - loss: 2.9135 - accuracy: 0.1404 - val_loss: 2.9634 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 19/250
1/1 - 8s - loss: 2.9169 - accuracy: 0.1447 - val_loss: 2.9564 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 20/250
1/1 - 9s - loss: 2.9097 - accuracy: 0.1447 - val_loss: 2.9497 - val_accuracy: 0.0833 - 9s/epoch - 9s/step
Epoch 21/250
1/1 - 8s - loss: 2.8950 - accuracy: 0.1535 - val_loss: 2.9430 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 22/250
1/1 - 8s - loss: 2.9077 - accuracy: 0.1360 - val_loss: 2.9342 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 23/250
1/1 - 8s - loss: 2.8698 - accuracy: 0.1798 - val_loss: 2.9248 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 24/250
1/1 - 8s - loss: 2.8822 - accuracy: 0.1447 - val_loss: 2.9133 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 25/250
1/1 - 8s - loss: 2.8664 - accuracy: 0.1842 - val_loss: 2.9003 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 26/250
1/1 - 8s - loss: 2.8584 - accuracy: 0.2149 - val_loss: 2.8841 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 27/250
1/1 - 8s - loss: 2.8527 - accuracy: 0.1096 - val_loss: 2.8660 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 28/250
1/1 - 8s - loss: 2.8052 - accuracy: 0.2105 - val_loss: 2.8442 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 29/250
1/1 - 8s - loss: 2.7975 - accuracy: 0.1974 - val_loss: 2.8190 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
```

Evaluate the test data

```
scor = cnn_model.evaluate( np.array(x_test),  np.array(y_test), verbose=0)

print('test los {:.4f}'.format(scor[0]))
print('test acc {:.4f}'.format(scor[1]))
```

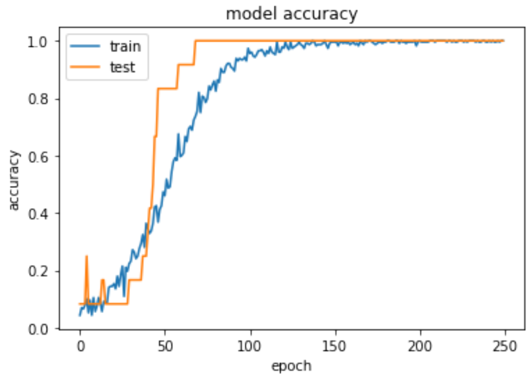
```
test los 0.3214
test acc 0.9500
```

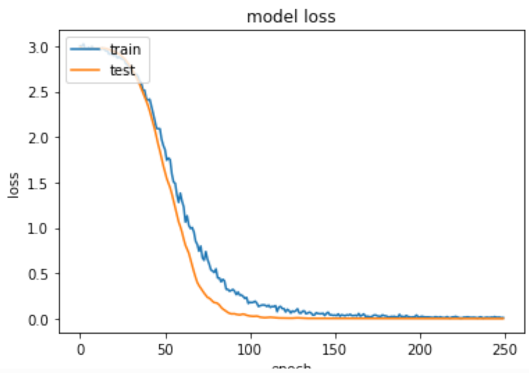
Step 7

plot the result

```
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```





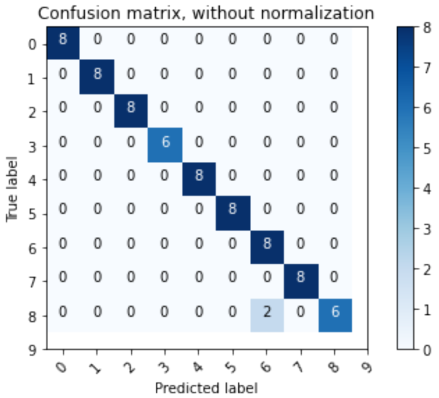
step 8

Plot Confusion Matrix

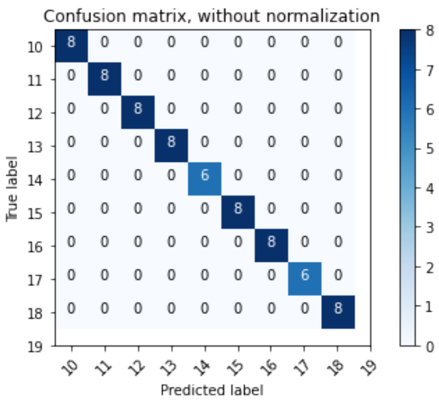
```
predicted =np.array( cnn_model.predict(x_test))
print(predicted)
print(y_test)
ynew = np.argmax(cnn_model.predict(x_test), axis=-1)


Acc=accuracy_score(y_test, ynew)
print("accuracy : ")
print(Acc)
#/tn, fp, fn, tp = confusion_matrix(np.array(y_test), ynew).ravel()
cnf_matrix=confusion_matrix(np.array(y_test), ynew)
```

```
y_test1 = np_utils.to_categorical(y_test, 20)



def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        #print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    #print(cm)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()


print('Confusion matrix, without normalization')
print(cnf_matrix)

plt.figure()
plot_confusion_matrix(cnf_matrix[1:10,1:10], classes=[0,1,2,3,4,5,6,7,8,9],
                      title='Confusion matrix, without normalization')

plt.figure()
plot_confusion_matrix(cnf_matrix[11:20,11:20], classes=[10,11,12,13,14,15,16,17,18,19],
                      title='Confusion matrix, without normalization')

print("Confusion matrix:\n%s" % confusion_matrix(np.array(y_test), ynew))
print(classification_report(np.array(y_test), ynew))
```

```
5/5 [==============================] - 2s 322ms/step
[[9.9546921e-01 1.9353812e-04 7.9316116e-08 ... 3.3380311e-05
  3.6349343e-03 3.7619247e-07]
 [9.6505862e-01 4.4584442e-05 4.9042995e-08 ... 2.2253296e-05
  2.3999320e-04 3.0376427e-08]
 [9.8446137e-01 2.5217005e-05 1.2214008e-06 ... 3.2328474e-04
  7.6428393e-04 7.3747998e-07]
 ...
 [3.5669402e-07 1.3151069e-05 6.9416594e-03 ... 3.9353850e-04
  8.3770874e-07 9.1356450e-01]
 [8.1324352e-08 3.1203381e-06 4.9745995e-03 ... 4.6892710e-05
  4.0027339e-07 9.8212498e-01]
 [2.4835536e-10 1.2370619e-07 2.1639163e-08 ... 2.9042346e-07
  2.9384781e-10 9.9999315e-01]]
[ 0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2
  3  3  3  3  3  3  3  3  4  4  4  4  4  4  4  4  5  5  5  5  5  5  5  5
  6  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  8  8  8  8  8  8  8  8
  9  9  9  9  9  9  9  9 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11
 12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13 14 14 14 14 14 14 14 14
 15 15 15 15 15 15 15 15 16 16 16 16 16 16 16 16 17 17 17 17 17 17 17 17
 18 18 18 18 18 18 18 18 19 19 19 19 19 19 19 19]
5/5 [==============================] - 2s 314ms/step
accuracy :
0.95
Confusion matrix, without normalization
[[8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0]
 [0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 2 0 6 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0]
 [2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0]
 [0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 6 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8]]
Confusion matrix, without normalization
```




```
Confusion matrix, without normalization
```




```
Confusion matrix:
[[8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0]
```