COS30018 – Intelligent Systems

Week 7 tutorial

This week we will cover the following items:

- A practical pipeline to build an actual Deep Learning application

1. Introduction

In this week, we will look at how to build a facial recognition system with Convolutional Neural Network (CNN) using TensorFlow. A practical pipeline that people in the industry are actually using to build a ML/DL project is demonstrated in this tutorial. You can apply the same approach for any ML/DL project that you encounter in the future.

Face recognition is the general task of identifying and verifying people from photographs of their face. This is a one-to-many mapping for a given face against a database of known faces, to find out who is the person in a given image.

The first part is always dealing with the data: we will look at how to search for a dataset, process the data and apply data augmentations (if necessary) to produce more data for training. Some common augmentations are adding or removing brightness, saturation, and more angles of the faces by flipping them and rotating them. Data in AI is even more important than the models. In real life, Data Scientists spend almost 80% of their time on preparing and managing data for their tasks.

The second part is building a convolutional neural network (CNN) model: This is a type of artificial neural network (ANN) used in image recognition and processing that is specially designed to process pixel data. This step involves pulling the pretrained model, tuning the parameters, compiling and training the model. The pretrained model we will use is called FaceNet, developed in 2015 by Google researchers. In this tutorial, you will discover how to develop a face detection system using FaceNet and Support Vector Machine (SVM) classifier to identify people from photographs.

The last part is to evaluate and testing the model performance with multiple evaluation metrics, using test data and unseen data.

After completing this tutorial, you will:

- Understand about FaceNet face recognition model.
- Know how to prepare a face detection dataset including extracting faces and extracting face features.
- Know how to fit, evaluate, and demonstrate an SVM model to predict identities from faces embedding.
- Have a wonderful project for your CV.

Part 1: Data pre-processing

a. Getting the dataset

A great and free resource to find datasets is Kaggle. This is the biggest Machine Learning and Data Science Community, you can probably find datasets for any ML/DL task that exists until now. Usually when you work in a big company, they will have their own datasets, otherwise if you work for startups, Kaggle is a very useful site for you to grab datasets.

The model in this tutorial will be trained and tested using the '5 Celebrity Faces Dataset' that contains many photographs of five different celebrities. It includes photos of: Ben Affleck, Elton John, Jerry Seinfeld, Madonna, and Mindy Kaling.

The dataset was prepared and made available for free download on Kaggle here: https://www.kaggle.com/datasets/dansbecker/5-celebrity-faces-dataset. You just need to create an account on Kaggle and download the dataset directly from there. I also provide the dataset on Canvas. Download an unzip the dataset in your local directory with the folder name '5-celebrity-faces-dataset'. Then you can import os module to list the contents of a directory.

```
import os
print(os.listdir("your-local-director../data/"))
```

You should now have a directory with the following structure:

```
5-celebrity-faces-dataset
├── train
│   ├── ben_afflek
│   ├── elton_john
│   ├── jerry_seinfeld
│   ├── madonna
│   └── mindy_kaling
└── val
    ├── ben_afflek
    ├── elton_john
    ├── jerry_seinfeld
    ├── madonna
    └── mindy_kaling
```

Figure 1. Dataset folder structure

We can see that there is a training dataset and a validation or test dataset.

Looking at some of the photos in the directories, we can see that the photos provide faces with a range of orientations, lighting, and in various sizes. Importantly, each photo contains one face of the person.

Figure 2. Example train images

We will use this dataset as the basis for our classifier, trained on the *'train'* dataset only and classify faces in the *'val'* dataset. You can use this same structure to develop a classifier with your own photographs.

b. Data pre-processing: Extract face in the images

The first step is to detect the face in each photograph and reduce the dataset to a series of faces only.

Face detection is the process of automatically locating faces in a photograph and localizing them by drawing a bounding box around their extent.

In this tutorial, we will also use the Multi-Task Cascaded Convolutional Neural Network, or MTCNN, for face detection. This is a state-of-the-art deep learning model for face detection, described in the 2016 paper titled "Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks."

This is another Deep Learning model, beside the FaceNet model that we will use later for Face Recognition. So you can see that a practical ML/DL project in real life may require multiple models to work together. An output of this model will be used as an input of another model.

Thanks to the development of our AI community, MTCNN now can be downloaded and imported directly and be ready to use, we do not need to code the model again.

First, install it via pip:

```
pip install mtcnn
```

We can confirm that the library was installed correctly by importing the library and printing the version, for example:

```
# confirm mtcnn was installed correctly
import mtcnn
# print version
print(mtcnn.__version__)
```

Running the example prints the current version of the library:

```
0.1.0
```

We can use the mtcnn library to create a face detector and extract faces for our use with the FaceNet face detector models in subsequent sections.

The first step is to load an image as a NumPy array, which we can achieve using the PIL library and the `open()` function. We will also convert the image to RGB, just in case the image has an alpha channel or is black and white.

```
import numpy as np
from PIL import Image

# load image from file
file_path = "your-local-director../train/ben_afflek/httpcsvkmeuaeccjpg.jpg"
image = Image.open(file_path)
# convert to RGB, if needed
image = image.convert('RGB')
# convert to array
pixels = np.asarray(image)
```

To display the image, we can use imread() function from the cv2 library to read the image file. This function returns a NumPy array representing the image. Then we use the pyplot module from matplotlib library to plot the image.

```
import cv2 # opencv
from matplotlib import pyplot as plt

img = cv2.imread(file_path)
plt.imshow(img, cmap = 'gray', interpolation = 'bicubic')
plt.xticks([]), plt.yticks([])  # to hide tick values on X and Y axis
plt.show()
```

You will then see the image:

Next, we can create an MTCNN face detector class and use it to detect all faces in the loaded photograph.

```
# create the detector, using default weights
detector = MTCNN()
# detect faces in the image
results = detector.detect_faces(pixels)
```

The result is a list of bounding boxes, where each bounding box defines a lower-left-corner of the bounding box, as well as the width and height. For our example image, there is only one face. Thus, if you print “results”, you will see the information below:

```
[{'box': [14, 27, 80, 95], 'confidence': 0.9999995231628418, 'keypoints': {'left_eye': (28, 74), 'right_eye': (53, 6
1), 'nose': (38, 90), 'mouth_left': (45, 109), 'mouth_right': (65, 98)}}]
```

We can determine the pixel coordinates of the bounding box as follows. Sometimes the library will return a negative pixel index, and I think this is a bug. We can fix this by taking the absolute value of the coordinates.

```
# extract the bounding box from the first face
x1, y1, width, height = results[0]['box']
# bug fix
x1, y1 = abs(x1), abs(y1)
x2, y2 = x1 + width, y1 + height
```

We can use these coordinates to extract the face.

```
# extract the face
face = pixels[y1:y2, x1:x2]
```

You can visualise the boxes by using patches module from matplotlib library.

```
import matplotlib.patches as patches

# Display the image
plt.imshow(cv2.cvtColor(pixels, cv2.COLOR_BGR2RGB))

# Plot the bounding boxes
for result in results:
    x, y, w, h = result['box']
    rect = patches.Rectangle((x, y), w, h, linewidth=2, edgecolor='r', facecolor='none')
    plt.gca().add_patch(rect)

# Set plot limits to show the entire image with bounding boxes
plt.xlim(0, pixels.shape[1])
plt.ylim(pixels.shape[0], 0)

plt.axis('off')  # Turn off axis labels and ticks
plt.show()
```

After running the above code, you will see the image below:

We can then use the PIL library to resize this small image of the face to the required size; specifically, the model expects square input faces with the shape 160×160.

```
# resize pixels to the model size
image = Image.fromarray(face)
image = image.resize((160, 160))
face_array = asarray(image)
```

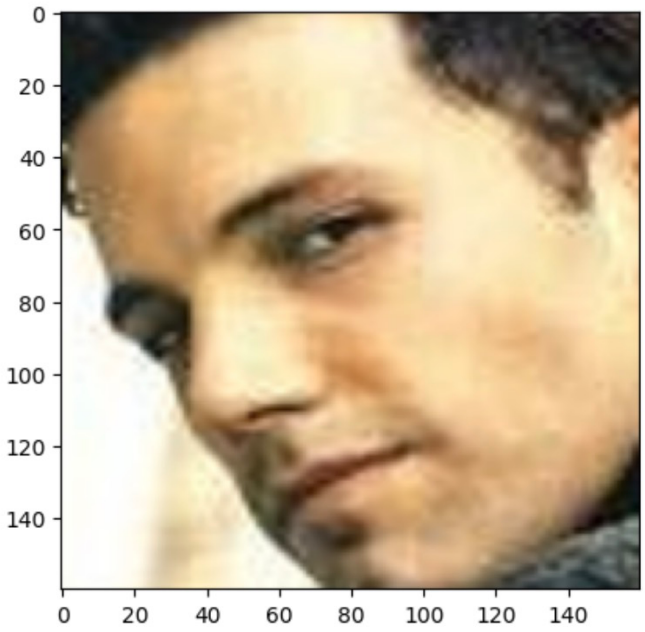
Tying all of this together, the function extract_face() will load a photograph from the loaded filename and return the extracted face represented by an array. The array has a shape of (height, width, 3). Where, height represents the height of the image in pixels, width represents the width of the image in pixels, and 3 is the number of color channels (red, green, and blue).

```
from numpy import asarray

# extract a single face from a given photograph
def extract_face(filename, required_size=(160, 160)):
    # load image from file
    image = Image.open(filename)
    # convert to RGB, if needed
    image = image.convert('RGB')
    # convert to array
    pixels = asarray(image)
    # create the detector, using default weights
    detector = MTCNN()
    # detect faces in the image
    results = detector.detect_faces(pixels)
    # extract the bounding box from the first face
    x1, y1, width, height = results[0]['box']
    # bug fix
    x1, y1 = abs(x1), abs(y1)
    x2, y2 = x1 + width, y1 + height
    # extract the face
    face = pixels[y1:y2, x1:x2]
    # resize pixels to the model size
    image = Image.fromarray(face)
    image = image.resize(required_size)
    face_array = asarray(image)
    return face_array

# load the photo and extract the face
pixels = extract_face('')
plt.imshow(pixels)
plt.show()
print(pixels.shape)
```

After running the above code, you should see the extracted face and the shape of the face array.



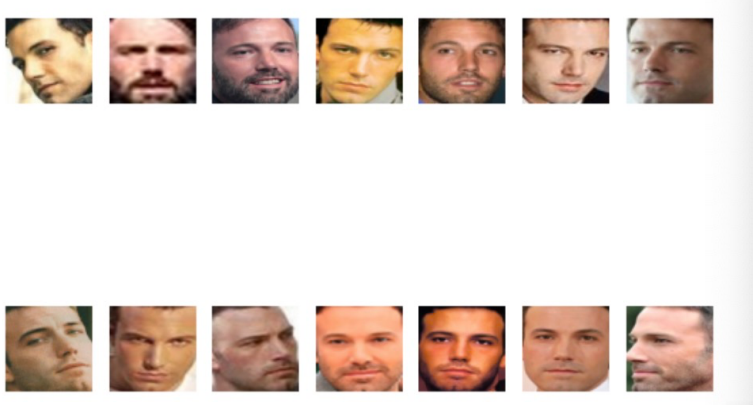

```
(160, 160, 3)
```

We can use this function to extract faces as needed in the next section that can be provided as input to the FaceNet model. Let's try to run the code below for "ben_afflek" to see what it produces. There are 14 photographs of Ben Affleck in that folder. We can detect the face in each photograph, and create a plot with 14 faces, with two rows of seven images each.

```
from os import listdir

# specify folder to plot
folder = 'your directory../ben_afflek/'
i = 1
# enumerate files
for filename in listdir(folder):
    # path
    path = folder + filename
    # get face
    face = extract_face(path)
    print(i, face.shape)
    # plot
    plt.subplot(2, 7, i)
    plt.axis('off')
    plt.imshow(face)
    i += 1
plt.show()
```

Running the example takes a moment and reports the progress of each loaded photograph along the way and the shape of the NumPy array containing the face pixel data. A figure is created containing the faces detected in the Ben Affleck directory. We can see that each face was correctly detected and that we have a range of lighting, skin tones, and orientations in the detected faces.



Next, we can extend this example to step over each subdirectory for a given dataset (e.g. 'train' or 'val'), extract the faces, and prepare a dataset with the name as the output label for each detected face.

The `load_faces()` function below will load all of the faces into a list for a given directory, e.g. '5-celebrity-faces-dataset/train/ben_afflek/'.

```
# load images and extract faces for all images in a directory
def load_faces(directory):
    faces = list()
    # enumerate files
    for filename in listdir(directory):
        # path
        path = directory + filename
        # get face
        face = extract_face(path)
        # store
        faces.append(face)
    return faces
```

c. Data pre-processing: Prepare the datasets

The `load_dataset()` function below takes a directory name such as '5-celebrity-faces-dataset/train/' and detects faces for each subdirectory (celebrity), assigning labels to each detected face. We can call the `load_faces()` function for each subdirectory in the 'train' or 'val' folders. Each face has one label, the name of the celebrity, which we can take from the directory name. `load_dataset()` returns the X and y elements of the dataset as NumPy arrays.

```
def load_dataset(directory):
    X, y = list(), list()
    # enumerate folders, on per class
    for subdir in listdir(directory):
        # path
        path = directory + subdir + '/'
        # skip any files that might be in the dir
        if not isdir(path):
            continue
        # load all faces in the subdirectory
        faces = load_faces(path)
        # create labels
        labels = [subdir for _ in range(len(faces))]
        # summarize progress
        print('>loaded %d examples for class: %s' % (len(faces), subdir))
        # store
        X.extend(faces)
        y.extend(labels)
    return asarray(X), asarray(y)
```

We can then call this function for the 'train' and 'val' folders to load all of the data, then save the results in a single compressed NumPy array file via the savez_compressed() function.

```
# load train dataset
trainX, trainy = load_dataset('5-celebrity-faces-dataset/train/')
print(trainX.shape, trainy.shape)
# load test dataset
testX, testy = load_dataset('5-celebrity-faces-dataset/val/')
print(testX.shape, testy.shape)
# save arrays to one file in compressed format
savez_compressed('5-celebrity-faces-dataset.npz', trainX, trainy, testX,
testy)
```

The complete example of detecting all of the faces in the 5 Celebrity Faces Dataset. First, all of the photos in the 'train' dataset are loaded, then faces are extracted, resulting in 93 samples with square face input and a class label string as output. Then the 'val' dataset is loaded, providing 25 samples that can be used as a test dataset.

Part 2: Building the Face Recognition model

d. Load FaceNet model:

We will use the pre-trained Keras FaceNet model provided by Hiroki Taniai in this tutorial. The model was trained on MS-Celeb-1M dataset and expects input images to be color, to have their pixel values whitened (standardized across all three channels), and to have a square shape of 160×160 pixels. The model can be downloaded from here:

https://drive.google.com/drive/folders/1pwQ3H4aJ8a6yyJHZkTwtjcL4wYWQb7bn

Download the model file and place it in your current working directory with the filename 'facenet_keras.h5'.

Please use the followed statement to check if Keras is installed:

```
import keras
print("Keras version:", keras.__version__)
```

If Keras is successfully installed, you will see the version, e.g.,

```
Keras version: 2.13.1
```

Up to this point, we have completed the data pre-processing file, let's create a new file to build our model by using keras-facenet. keras-facenet is a simple wrapper around this implementation of FaceNet. Let's first install keras-facenet by using: pip install keras-facenet.

We can load the `FaceNet` model directly in keras-facenet, for example:

```
# example of loading the keras facenet model
from keras_facenet import FaceNet

facenet_model = FaceNet()
print('Loaded Model')
```

e. Create face embeddings:

A face embedding is a vector that represents the features extracted from the face. This can then be compared with the vectors generated for other faces. For example, another vector that is close (by some measure) may be the same person, whereas another vector that is far (by some measure) may be a different person.

The classifier model that we want to develop will take a face embedding as input and predict the identity of the face. The FaceNet model will generate this embedding for a given image of a face.

First, we can load our detected faces dataset using the `load()` NumPy function.

```
# load the face dataset
data = load('5-celebrity-faces-dataset.npz')
trainX, trainy, testX, testy = data['arr_0'], data['arr_1'],
data['arr_2'], data['arr_3']
print('Loaded: ', trainX.shape, trainy.shape, testX.shape, testy.shape)
```

Next, we can load our FaceNet model ready for converting faces into face embeddings.

```
# load the facenet model
facenet_model = FaceNet()
print('Loaded Model')
```

We can then enumerate each face in the train and test datasets to get its embedding.

To get an embedding, the pixel values of the image need to be suitably prepared to meet the

expectations of the FaceNet model.

```
# scale pixel values
face_pixels = face_pixels.astype('float32')
```

In order to make an embedding for one example in Keras, we must expand the dimensions so that the face array is one sample.

```
# transform face into one sample
samples = np.expand_dims(face_pixels, axis=0)
```

We can then use the model to extract the embedding vector.

```
# get embedding vector
yhat = model. embeddings(samples)
return yhat[0]
```

The `get_embedding()` function defined below implements these behaviors and will return a face embedding given a single image of a face and the loaded FaceNet model.

```
# get the face embedding for one face
def get_embedding(model, face_pixels):
    # scale pixel values
    face_pixels = face_pixels.astype('float32')
    # transform face into one sample
    sample = np.expand_dims(face_pixels, axis=0)
    # get embedding vector
    yhat = model.embeddings(sample)
    return yhat[0]
```

The complete example of converting each face into a face embedding in the train and test datasets is listed below.

```
from numpy import load

# load the face dataset
data = load('5-celebrity-faces-dataset.npz')
trainX, trainy, testX, testy = data['arr_0'], data['arr_1'],data['arr_2'], data['arr_3']
print('Loaded: ', trainX.shape, trainy.shape, testX.shape, testy.shape)
# load the facenet model
facenet_model = FaceNet()
print('Loaded Model')
# convert each face in the train set to an embedding
emdTrainX = list()
for face_pixels in trainX:
    embedding = get_embedding(facenet_model, face_pixels)
    emdTrainX.append(embedding)
emdTrainX = asarray(emdTrainX)
print(emdTrainX.shape)
# convert each face in the test set to an embedding
emdTestX = list()
for face_pixels in testX:
    embedding = get_embedding(facenet_model, face_pixels)
    emdTestX.append(embedding)
emdTestX = asarray(emdTestX)
print(emdTestX.shape)
# save arrays to one file in compressed format
savez_compressed('5-celebrity-faces-embeddings.npz', emdTrainX, trainy, emdTestX, testy)
```

We can see that the face dataset was loaded correctly and so was the model. The train dataset was then transformed into 93 face embeddings. The 25 examples in the test dataset were also suitably converted to face embeddings.

The resulting datasets were then saved to a compressed NumPy array with the name '5-celebrity-faces-embeddings.npz' in the current working directory. We are now ready to develop our face classifier system.

Part 3: Evaluate and testing the model.

In this section, we will develop a model to classify face embeddings as one of the known celebrities in the 5 Celebrity Faces Dataset.

f. Perform Face classification

First, we must load the face embeddings dataset.

```
# load the face dataset
data = load('5-celebrity-faces-embeddings.npz')
emdTrainX, trainy, emdTestX, testy = data['arr_0'], data['arr_1'], data['arr_2'], data['arr_3']
print('Dataset: train=%d, test=%d' % (emdTrainX.shape[0], emdTestX.shape[0]))
```

Next, the data requires some minor preparation prior to modeling. It is a good practice to normalize the face embedding vectors. It is a good practice because the vectors are often compared to each other using a distance metric. In this context, vector normalization means scaling the values until the length or magnitude of the vectors is 1 or unit length. This can be achieved using the Normalizer class in scikit-learn.

```
# normalize input vectors
in_encoder = Normalizer()
emdTrainX_norm = in_encoder.transform(emdTrainX)
emdTestX_norm = in_encoder.transform(emdTestX)
```

Then, the string target variables for each celebrity name need to be converted to integers. This can be achieved via the LabelEncoder class in scikit-learn.

```
# label encode targets
out_encoder = LabelEncoder()
out_encoder.fit(trainy)
trainy_enc = out_encoder.transform(trainy)
testy_enc = out_encoder.transform(testy)
```

Next, we can fit a model. It is common to use a Linear Support Vector Machine (SVM) when working with normalized face embedding inputs. This is because the method is very effective at separating the face embedding vectors. We can fit a linear SVM to the training data using the SVC class in scikit-learn and setting the 'kernel' attribute to 'linear'. We may also want probabilities later when making predictions, which can be configured by setting 'probability' to 'True'.

```
# fit model
model = SVC(kernel='linear', probability=True)
model.fit(emdTrainX_norm, trainy_enc)
```

Next, we can evaluate the model. This can be achieved by using the fit model to make a prediction for each example in the train and test datasets and then calculating the classification accuracy.

```
# predict
yhat_train = model.predict(emdTrainX_norm)
yhat_test = model.predict(emdTestX_norm)
# score
score_train = accuracy_score(trainy_enc, yhat_train)
score_test = accuracy_score(testy_enc, yhat_test)
# summarise
print('Accuracy: train=%.3f, test=%.3f' % (score_train*100, score_test*100))
```

Tying all of this together, the complete example of fitting a Linear SVM on the face embeddings for the 5 Celebrity Faces Dataset is listed below.

```
# Perform Face classification

from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import Normalizer
from sklearn.svm import SVC

# load the face dataset
data = load('5-celebrity-faces-embeddings.npz')
emdTrainX, trainy, emdTestX, testy = data['arr_0'], data['arr_1'], data['arr_2'], data['arr_3']
print('Dataset: train=%d, test=%d' % (emdTrainX.shape[0], emdTestX.shape[0]))

# normalize input vectors
in_encoder = Normalizer()
emdTrainX_norm = in_encoder.transform(emdTrainX)
emdTestX_norm = in_encoder.transform(emdTestX)

# label encode targets
out_encoder = LabelEncoder()
out_encoder.fit(trainy)
trainy_enc = out_encoder.transform(trainy)
testy_enc = out_encoder.transform(testy)

# fit model
model = SVC(kernel='linear', probability=True)
model.fit(emdTrainX_norm, trainy_enc)

# predict
yhat_train = model.predict(emdTrainX_norm)
yhat_test = model.predict(emdTestX_norm)
# score
score_train = accuracy_score(trainy_enc, yhat_train)
score_test = accuracy_score(testy_enc, yhat_test)
# summarise
print('Accuracy: train=%.3f, test=%.3f' % (score_train*100, score_test*100))
```

Next, the model is evaluated on the train and test dataset, showing perfect classification accuracy. This is not surprising given the size of the dataset and the power of the face detection and face recognition models used.

```
Dataset: train=93, test=25
Accuracy: train=100.000, test=100.000
```

g. Perform manual prediction

We can make it more interesting by plotting the original face and the prediction.

First, we need to load the face dataset, specifically the faces in the test dataset. We could also load the original photos to make it even more interesting.

```
# select a random face from test set
selection = choice([i for i in range(testX.shape[0])])
random_face = testX[selection]
random_face_emd = emdTestX_norm[selection]
random_face_class = testy_enc[selection]
random_face_name = out_encoder.inverse_transform([random_face_class])
```

Next, we can use the face embedding as an input to make a single prediction with the fit model. We can predict both the class integer and the probability of the prediction.

```
# prediction for the face
samples = np.expand_dims(random_face_emd, axis=0)
yhat_class = model.predict(samples)
yhat_prob = model.predict_proba(samples)
```

We can then get the name for the predicted class integer, and the probability for this prediction.

```
# get name
class_index = yhat_class[0]
class_probability = yhat_prob[0,class_index] * 100
predict_names = out_encoder.inverse_transform(yhat_class)
```

We can then print this information.

```
print('Predicted: %s (%.3f)' % (predict_names[0], class_probability))
print('Expected: %s' % random_face_name[0])
```

We can also plot the face pixels along with the predicted name and probability.

```
# plot face
plt.imshow(random_face)
title = '%s (%.3f)' % (predict_names[0], class_probability)
plt.title(title)
plt.show()
```

Tying all of this together, the complete example for predicting the identity for a given unseen photo in the test dataset is listed below.

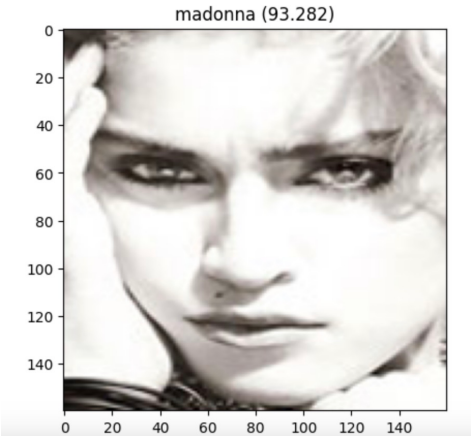
```
from random import choice
# select a random face from test set
selection = choice([i for i in range(testX.shape[0])])
random_face = testX[selection]
random_face_emd = emdTestX_norm[selection]
random_face_class = testy_enc[selection]
random_face_name = out_encoder.inverse_transform([random_face_class])

# prediction for the face
samples = np.expand_dims(random_face_emd, axis=0)
yhat_class = model.predict(samples)
yhat_prob = model.predict_proba(samples)
# get name
class_index = yhat_class[0]
class_probability = yhat_prob[0,class_index] * 100
predict_names = out_encoder.inverse_transform(yhat_class)

print('Predicted: %s (%.3f)' % (predict_names[0], class_probability))
print('Expected: %s' % random_face_name[0])

# plot face
plt.imshow(random_face)
title = '%s (%.3f)' % (predict_names[0], class_probability)
plt.title(title)
plt.show()
```

A different random example from the test dataset will be selected each time the code is run. A plot of the chosen face is also created, showing the predicted name and probability in the image title.



Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.

References:

- Paper: FaceNet: A Unified Embedding for Face Recognition and Clustering, 2015.
- OpenFace PyTorch Project.
- OpenFace Keras Project, GitHub.
- Keras FaceNet Project, GitHub.
- MS-Celeb 1M Dataset.
- Jason Brownlee, *How to Develop a Face Recognition System Using FaceNet in Keras*, Machine Learning Mastery, Available from https://machinelearningmastery.com/how-to-develop-a-face-recognition-system-using-facenet-in-keras-and-an-svm-classifier/, accessed Sep 6th, 2022.