CHAPTER 12

Face Detection and Recognition

Face detection is the process of detecting a face in an image or video.

Face recognition is the process of detecting face in an image and then using algorithms to identify who the face belongs to. Face recognition is thus a form of person identification.

You first need to extract features from the image for training the machine learning classifier to identify faces in the image. Not only are these systems nonsubjective, but they are also *automatic*—no hand labeling of facial features is required. You simply extract features from the faces, train your classifier, and then use it to identify subsequent faces.

Since for face recognition you first need to detect a face from the image, you can think of face recognition as a two-phase stage.

- *Stage 1*: Detect the presence of faces in an image or video stream using methods such as Haar cascades, HOG + Linear SVM, deep learning, or any other algorithm that can localize faces.
- *Stage 2*: Take each of the faces detected during the localization phase and learn whom the face belongs to—this is where you actually assign a name to a face.

© Navin Kumar Manaswi 2018
N. K. Manaswi, *Deep Learning with Applications Using Python*,
https://doi.org/10.1007/978-1-4842-3516-4_12

Face Detection, Face Recognition, and Face Analysis

There is a difference between face detection, face recognition, and face analysis.

- *Face detection*: This is the technique of finding all the human faces in an image.
- *Face recognition*: This is the next step after face detection. In face recognition, you identify which face belongs to which person using an existing image repository.
- *Face analysis*: A face is examined, and some inference is taken out such as age, complexion, and so on.

OpenCV

OpenCV provides three methods for face recognition (see Figure 12-1):

- Eigenfaces
- Local binary pattern histograms (LBPHs)
- Fisherfaces

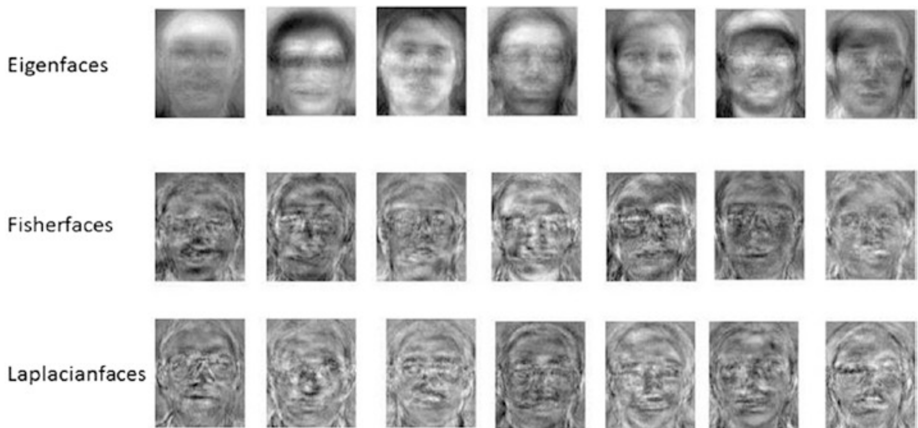

Figure 12-1. *Applying OpenCV methods to faces*

All three methods recognize a face by comparing the face with some training set of known faces. For training, you supply the algorithm with faces and label them with the person they belong to. When you use the algorithm to recognize some unknown face, it uses the model trained on the training set to make the recognition. Each of the three aforementioned methods uses the training set a bit differently.

Laplacian faces can be another way to recognize face.

Eigenfaces

The eigenfaces algorithm uses principal component analysis to construct a low-dimensional representation of face images, which you will use as features for the corresponding face images (Figure 12-2).

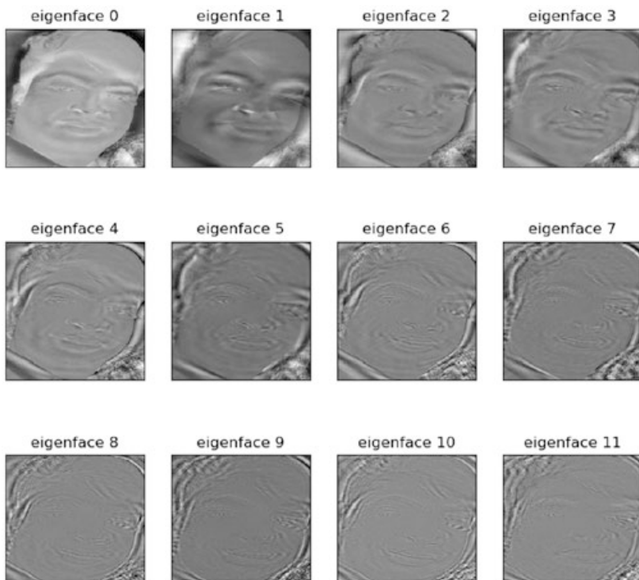

Figure 12-2. *Applying Eigenvalue decomposition and extracting 11 eigenfaces with the largest magnitude*

For this, you collect a data set of faces with multiple face images of each person you want to recognize—it's like having multiple *training examples* of an image class you want to label in image classification. With this data set of face images, presumed to be the same width and height and ideally with their eyes and facial structures aligned at the same (x, y) coordinates, you apply an eigenvalue decomposition of the data set, keeping the eigenvectors with the largest corresponding eigenvalues.

Given these eigenvectors, a face can then be represented as a linear combination of what Kirby and Sirovich called *eigenfaces*. The eigenfaces algorithm looks at the whole data set.

LBPH

You can analyze each image independently in LBPH. The LBPH method is somewhat simpler, in the sense that you characterize each image in the data set locally; when a new unknown image is provided, you perform the same analysis on it and compare the result to each of the images in the data set. The way that you analyze the images is by characterizing the local patterns in each location in the image.

While the eigenfaces algorithm relies on PCA to construct a low-dimensional representation of face images, the local binary pattern (LBP) method relies on, as the name suggests, feature extraction.

First introduced by Ahonen et al. in the 2006 paper "Face Recognition with Local Binary Patterns," the method suggests dividing a face image into a 7×7 grid of equally sized cells (Figure 12-3).

Figure 12-3. *Applying LBPH for face recognition starts by dividing the face image into a 7x7 grid of equally sized cells*

You then extract a local binary pattern histogram from each of the 49 cells. By dividing the image into cells, you introduce *locality* into the final feature vector. Furthermore, cells in the center have more weight such that they contribute *more* to the overall representation. Cells in the corners carry less identifying facial information compared to the cells in the center of the grid (which contain eyes, nose, and lip structures). Finally, you concatenate this weighted LBP histogram from the 49 cells to form your final feature vector.

Fisherfaces

The Principal Component Analysis (PCA), which is the core of the Eigenfaces method, finds a linear combination of features that maximizes the total variance in data. While this is clearly a powerful way to represent data, it doesn't consider any classes and so a lot of discriminative information may be lost when throwing components away. Imagine a situation where the variance in your data is generated by an external source, let it be the light. The components identified by a PCA do not necessarily contain any discriminative information at all, so the projected samples are smeared together and a classification becomes impossible.

The Linear Discriminant Analysis performs a class-specific dimensionality reduction and was invented by the great statistician Sir R. A. Fisher. The use of multiple measurements in taxonomic problems. In order to find the combination of features that separates best between classes the Linear Discriminant Analysis maximizes the ratio of between-classes to within-classes scatter, instead of maximizing the overall scatter. The idea is simple: same classes should cluster tightly together, while different classes are as far away as possible from each other in the lower-dimensional representation.

Detecting a Face

The first feature that you need for performing face recognition is to detect where in the current image a face is present. In Python you can use Haar cascade filters of the OpenCV library to do this efficiently.

For the implementation shown here, I used Anaconda with Python 3.5, OpenCV 3.1.0, and dlib 19.1.0. To use the following code, please make sure that you have these (or newer) versions.

To do the face detection, a couple of initializations must be done, as shown here:

```
# Import the OpenCV library
import cv2
# Initialize a face cascade using the frontal face haar cascade provided
# with the OpenCV2 library. This will be required for face detection in an
# image.
faceCascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
# The desired output width and height, can be modified according to the needs.
OUTPUT_SIZE_WIDTH = 700
OUTPUT_SIZE_HEIGHT = 600

# Open the first webcam device
capture = cv2.VideoCapture(0)

# Create two opencv named windows for showing the input, output images.
cv2.namedWindow("base-image", cv2.WINDOW_AUTOSIZE)
cv2.namedWindow("result-image", cv2.WINDOW_AUTOSIZE)

# Position the windows next to each other
cv2.moveWindow("base-image", 20, 200)
cv2.moveWindow("result-image", 640, 200)

# Start the window thread for the two windows we are using
cv2.startWindowThread()

rectangleColor = (0, 100, 255)
```

The rest of the code will be an infinite loop that keeps getting the latest image from the webcam, detects all faces in the image retrieved, draws a rectangle around the largest face detected, and then finally shows the input, output images in a window (Figure 12-4).

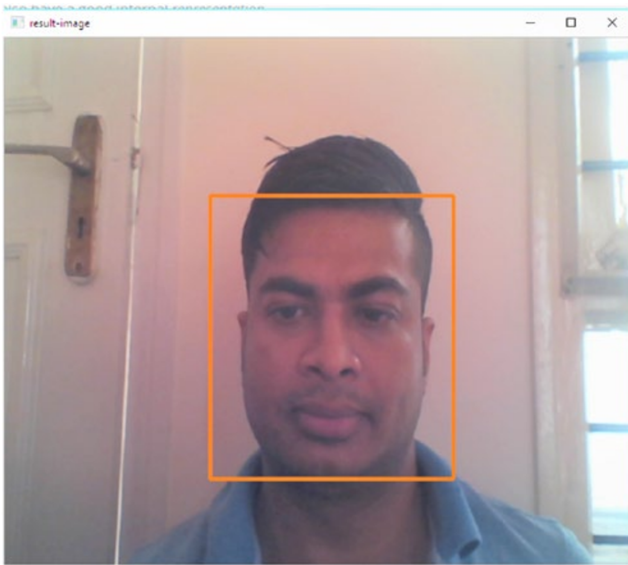

Figure 12-4. *A sample output showing detected face*

You can do this with the following code within an infinite loop:

```
# Retrieve the latest image from the webcam
rc,fullSizeBaseImage = capture.read()
# Resize the image to 520x420
baseImage= cv2.resize(fullSizeBaseImage, (520, 420))

# Check if a key was pressed and if it was Q or q, then destroy all
# opencv windows and exit the application, stopping the infinite loop.
pressedKey = cv2.waitKey(2)
if (pressedKey == ord('Q')) | (pressedKey == ord('q')):
cv2.destroyAllWindows()
exit(0)
# Result image is the image we will show the user, which is a
# combination of the original image captured from the webcam with the
# overlayed rectangle detecting the largest face
resultImage = baseImage.copy()

# We will be using gray colored image for face detection.
# So we need to convert the baseImage captured by webcam to a gray-based image
gray_image = cv2.cvtColor(baseImage, cv2.COLOR_BGR2GRAY)
# Now use the haar cascade detector to find all faces in the
# image
faces = faceCascade.detectMultiScale(gray_image, 1.3, 5)
```

```
# As we are only interested in the 'largest' face, we need to
# calculate the largest area of the found rectangle.
# For this, first initialize the required variables to 0.
maxArea = 0
x = 0
y = 0
w = 0
h = 0

# Loop over all faces found in the image and check if the area for this face is
# the largest so far
for(_x, _y, _w, _h) in faces:
if _w * _h > maxArea:
        x = _x
        y = _y
        w = _w
        h = _h
    maxArea = w * h

# If any face is found, draw a rectangle around the
#  largest face present in the picture
If maxArea > 0:
cv2.rectangle(resultImage, (x-10, y-20),
(x + w+10, y + h+20), rectangleColor, 2)
# Since we want to show something larger on the screen than the
# original 520x420, we resize the image again

# Note that it would also be possible to keep the large version
# of the baseimage and make the result image a copy of this large
# base image and use the scaling factor to draw the rectangle
# at the right coordinates.
largeResult = cv2.resize(resultImage,
(OUTPUT_SIZE_WIDTH, OUTPUT_SIZE_HEIGHT))
# Finally, we show the images on the screen
cv2.imshow("base-image", baseImage)
cv2.imshow("result-image", largeResult)
```

Tracking the Face

The previous code for face detection has some drawbacks.

- The code might be computationally expensive.
- If the detected person is turning their head slightly, the Haar cascade might not detect the face.
- It's difficult to keep track of a face between frames.

A better approach for this is to do the detection of the face once and then make use of the correlation tracker from the excellent dlib library to just keep track of the faces from frame to frame.

For this to work, you need to import another library and initialize additional variables.

```
import dlib

# Create the tracker we will use to recognize face in different frames
# we get from the webcam
tracker = dlib.correlation_tracker()

# The Boolean variable we use to keep track whether we are
# using dlib tracker, or not.
trackingFace = 0
```

Within the infinite `for` loop, you will now determine whether the dlib correlation tracker is currently tracking a region in the image. If this is *not* the case, you will use a similar code as before to find the largest face, but instead of drawing the rectangle, you use the found coordinates to initialize the correlation tracker.

```
# If we are not tracking a face, then try to detect one using the above code itself.
if not trackingFace:

# We will be using gray colored image for face detection.
# So we need to convert the baseImage captured by webcam to a gray-based image
gray = cv2.cvtColor(baseImage, cv2.COLOR_BGR2GRAY)
# Now use the haar cascade detector to find all faces
# in the image
faces=faceCascade.detectMultiScale(gray,1.3,5)

# In the console we can show our this case of using the
# detector for a face, when we are detecting it for first time.
print("Using the cascade detector to detect face")


# As we are only interested in the 'largest' face, we need to
# calculate the largest area of the found rectangle.
# For this, first initialize the required variables to 0.
maxArea = 0
x = 0
y = 0
w = 0
h =0
```

```
# Loop over all faces and check if the area for this
# face is the largest so far
# We need to convert it to int here because dlib tracker
# needs an int as its argument. If we omit the cast to
# int here, you will get cast errors since the detector
# returns numpy.int32 and the tracker requires an int
for(_x, _y, _w, _h) in faces:
if _w * _h > maxArea:
            x = int(_x)
            y = int(_y)
            w = int(_w)
            h = int(_h)
        maxArea = w * h

# If any face is found, draw a rectangle around the
# largest face present in the picture
if maxArea > 0:

#Initialize the tracker
tracker.start_track(baseImage,
dlib.rectangle(x-10, y-20, x+w+10, y+h+20))

# Set the indicator variable such that we know the
# tracker is tracking a face in the image
trackingFace = 1
```

Now the final bit within the infinite loop is to check again if the correlation tracker is actively tracking a face (i.e., did it just detect a face with the previous code, `trankingFace=1`?). If the tracker is actively tracking a face in the image, you will update the tracker. Depending on the quality of the update (i.e., how confident the tracker is about whether it is still tracking the same face), you either draw a rectangle around the region indicated by the tracker or indicate you are not tracking a face anymore.

```
# Check if the tracker is actively tracking a face in the image
if trackingFace:

# Update the tracker and request information about the
# quality of the tracking update
trackingQuality = tracker.update(baseImage)

# If the tracking quality is good enough, determine the
# updated position of the tracked region and draw the
# rectangle
If trackingQuality >= 9.0:
tracked_position = tracker.get_position()

t_x = int(tracked_position.left())
t_y = int(tracked_position.top())
t_w = int(tracked_position.width())
t_h = int(tracked_position.height())
cv2.rectangle(resultImage, (t_x,t_y),
(t_x+t_w, t_y+t_h),
rectangleColor, 2)

else:
# If the quality of the tracking update is not good enough
# for us  (e.g. the face being tracked moved out of the
# screen) we stop the tracking of the face and in the
# next loop we will find the largest face in the image
# again
trackingFace = 0
```

As you can see in the code, you print a message to the console every time you use the detector again. If you look at the output of the console while running this application, you will notice that even if you move quite a bit around on the screen, the tracker is quite good at following a face once it is detected.

Face Recognition

A face recognition system identifies the name of person present in the video frame by matching the face in each frame of video with the trained images and returns (and writes in a CSV file) the label if the face in the frame is successfully matched. You will now see how to create a face recognition system step-by-step.

First you import all the required libraries. face_recognition is the simple library built using dlib's state-of-the-art face recognition also built with deep learning.

```
import os
import re
import warnings
import scipy.misc
import cv2
import face_recognition
from PIL import Image
import argparse
import csv
import os
```

Argparse is a Python library that allows you to add your own arguments to a file; it can then be used to input any image directory or a file path at the time of execution.

```
parser = argparse.ArgumentParser()
parser.add_argument("-i", "--images-dir", help="image dir")
parser.add_argument("-v", "--video", help="video to recognize faces on")
parser.add_argument("-o", "--output-csv", help="Ouput csv file [Optional]")
parser.add_argument("-u", "--upsample-rate", help="How many times to upsample the image looking for faces. Higher numbers
                    find smaller faces. [Optional]")
args = vars(parser.parse_args())
```

In the previous code, while running this Python file, you have to specify the following: the training input image directory, video file which we will use as data set, and an output CSV file to write the output at each time frame.

```
#Check if argument values are valid
if args.get("images_dir", None) is None and os.path.exists(args.get("images_dir", None)):
    print("Please check the path to images folder")
    exit()
if args.get("video", None) is None and os.path.isfile(args.get("video", None)):
    print("Please check the path to video")
    exit()
if args.get("output_csv", None) is None:
    print("You haven't specified an output csv file. Nothing will be written.")
# By default upsample rate = 1
upsample_rate = args.get("upsample_rate", None)
if upsample_rate is None:
    upsample_rate = 1
```

```
# Helper functions
def image_files_in_folder(folder):
    return [os.path.join(folder, f) for f in os.listdir(folder) if re.match(r'.*\.(jpg|jpeg|png)', f, flags=re.I)]
```

By using the previous function, all image files from the specified folder can be read.

The following function tests the input frame with the known training images:

```
def test_image(image_to_check, known_names, known_face_encodings, number_of_times_to_upsample=1):
    """
    Test if any face is recognized in unknown image by checking known images
    :paramimage_to_check: Numpy array of the image
    :paramknown_names: List containing known labels
    :paramknown_face_encodings: List containing training image labels
    :paramnumber_of_times_to_upsample: How many times to upsample the image looking for
    faces. Higher numbers find smaller faces.
    :return: A list of labels of known names
    """
    # unknown_image = face_recognition.load_image_file(image_to_check)
    unknown_image = image_to_check
    # Scale down the image to make it run faster
    if unknown_image.shape[1] > 1600:
        scale_factor = 1600 / unknown_image.shape[1]
        with warnings.catch_warnings():
            warnings.simplefilter("ignore")
        unknown_image = scipy.misc.imresize(unknown_image, scale_factor)
    face_locations = face_recognition.face_locations(unknown_image,  number_of_times_to_upsample)
    unknown_encodings = face_recognition.face_encodings(unknown_image, face_locations)
    result = []
    for unknown_encoding in unknown_encodings:
        result = face_recognition.compare_faces(known_face_encodings, unknown_encoding)
    result_encoding = []
    for nameIndex, is_match in enumerate(result):
        if is_match:
            result_encoding.append(known_names[nameIndex])

    return result_encoding
```

Now you define the function to extract the label for matched, known images.

```
def map_file_pattern_to_label(labels_with_pattern, labels_list):
    """
    Map file name pattern to full label
    :paramlabels_with_pattern: dict : { "file_name_pattern": "full_label" }
    :paramlabels_list: list : list of labels of file names got from test_image()
    :return: list of full labels
    """
    result_list = []
    for key, label in labels_with_pattern.items():
        for img_labels in labels_list:
            if str(key).lower() in str(img_labels).lower():
                if str(label) not in result_list:
                    result_list.append(str(label))
                # continue
    # result_list = [label for key, label in labels_with_pattern if str(key).lower() in labels_list]
    return result_list
```

Read the input video to extract test frames.

```
cap = cv2.VideoCapture(args["video"])

#get the training images
training_encodings = []
training_labels = []
for file in image_files_in_folder(args['images_dir']):
    basename = os.path.splitext(os.path.basename(file))[0]
    img = face_recognition.load_image_file(file)
    encodings = face_recognition.face_encodings(img)

    if len(encodings) > 1:
        print("WARNING: More than one face found in {}. Only considering the first face.".format(file))

    if len(encodings) == 0:
        print("WARNING: No faces found in {}. Ignoring file.".format(file))
    if len(encodings):
        training_labels.append(basename)
        training_encodings.append(encodings[0])


csvfile = None
csvwriter = None
if args.get("output_csv", None) is not None:
    csvfile = open(args.get("output_csv"), 'w')
    csvwriter = csv.writer(csvfile, delimiter=',', quotechar='|', quoting=csv.QUOTE_MINIMAL)

ret, firstFrame = cap.read()
frameRate = cap.get(cv2.CAP_PROP_FPS)
```

Now define the labels of your training sets. Then match the extracted frame from the given input video to get the desired results.

```
# Labels with file pattern, edit this
label_pattern = {
    "shah": "Shahrukh Khan",
    "amir": "Amir Khan"
          }

# match each frame in video with our trained set of labeled images
while ret:
    curr_frame = cap.get(1)

    ret, frame = cap.read()

    result = test_image(frame, training_labels, training_encodings, upsample_rate)
    labels = map_file_pattern_to_label(label_pattern, result)
    curr_time = curr_frame / frameRate
    print("Time: {} faces: {}".format(curr_time, labels))
    if csvwriter:
        csvwriter.writerow([curr_time, labels])
    cv2.imshow('frame', frame)

    key = cv2.waitKey(1) & 0xFF
    if key == ord('q'):
        break
if csvfile:
    csvfile.close()
cap.release()
cv2.destroyAllWindows()
```

Deep Learning–Based Face Recognition

Import the necessary packages.

```
import cv2                # working with, mainly resizing, images
import numpy as np        # dealing with arrays
import os                 # dealing with directories
from random import shuffle # mixing up or currently ordered data that might lead our network astray in training.
from tqdm import tqdm
from scipy import misc
import tflearn
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.core import input_data, dropout, fully_connected
from tflearn.layers.estimator import regression
import tensorflow as tf
import glob
import matplotlib.pyplot as plt
import dlib
```

Initialize the variables.

```
from skimage import io
tf.reset_default_graph()
TRAIN_DIR ='resize_a/train'
TEST_DIR ='resize_a/test'
IMG_SIZE = 200
boxScale=1
LR = 1e-3
MODEL_NAME = 'quickest.model'.format(LR, '2conv-basic')
```

The `label_img()` function is used to create the label array, and the `detect_faces()` function detects the face portion in the image.

```
def label_img(img):
    word = img.split('(')[-2]
    word_label = word[0]
    if word_label == 'R': return [1,0]

    elif word_label == 'A': return [0,1]

def detect_faces(image):

    # Create a face detector
    face_detector = dlib.get_frontal_face_detector()

    # Run detector and get bounding boxes of the faces on image.
    detected_faces = face_detector(image, 1)
    face_frames = [(x.left(), x.top(),
                    x.right(), x.bottom()) for x in detected_faces]

    return face_frames
```

The `create_train_data()` function is used for preprocessing the training data.

```
def create_train_data():
    training_data = []
    for img in tqdm(os.listdir(TRAIN_DIR)):
        label = label_img(img)
        path = os.path.join(TRAIN_DIR,img)
        img= misc.imread(path)
        img = cv2.imread(path,cv2.IMREAD_GRAYSCALE)
        img = cv2.resize(img, (IMG_SIZE,IMG_SIZE))
        detected_faces = detect_faces(img)
        for n, face_rect in enumerate(detected_faces):
            img = Image.fromarray(img).crop(face_rect)
            img = np.array(img)
        img = cv2.resize(img, (IMG_SIZE,IMG_SIZE))
# If any face is found, draw a rectangle around the
#  largest face present in the picture

        training_data.append([np.array(img),np.array(label)])
    shuffle(training_data)
    np.save('train_data.npy', training_data)
    return training_data
```

The `process_test_data()` function is used to preprocess the testing data.

```
def process_test_data():
    testing_data = []
    for img in tqdm(os.listdir(TEST_DIR)):
        path = os.path.join(TEST_DIR,img)
        imgnum = img.split('.')[-2]
        img_num=get_num(imgnum)
        img= misc.imread(path)
        img = cv2.imread(path,cv2.IMREAD_GRAYSCALE)
        img = cv2.resize(img, (IMG_SIZE,IMG_SIZE))
        detected_faces = detect_faces(img)
        for n, face_rect in enumerate(detected_faces):
            img = Image.fromarray(img).crop(face_rect)
            img = np.array(img)
        img = cv2.resize(img, (IMG_SIZE,IMG_SIZE))
# If any face is found, draw a rectangle around the
#  largest face present in the picture

        testing_data.append([np.array(img), img_num])
```

Then you create the model and fit the training data in the model.

```
train_data= create_train_data()
train = train_data[:-2]
test = train_data[-2:]
X = np.array([i[0] for i in train]).reshape(-1,200,200,1)
Y = [i[1] for i in train]
test_x = np.array([i[0] for i in test]).reshape(-1,200,200,1)
test_y = [i[1] for i in test]
convnet = input_data(shape=[None, 200, 200, 1], name='input')

convnet = conv_2d(convnet, 4, 5, activation='relu')
convnet = max_pool_2d(convnet, 5)

convnet = conv_2d(convnet, 5, 5, activation='relu')
convnet = max_pool_2d(convnet, 5)

convnet = conv_2d(convnet, 8, 5, activation='relu')
convnet = max_pool_2d(convnet, 5)

convnet = fully_connected(convnet, 8, activation='relu')
convnet = dropout(convnet, 0.2)

convnet = fully_connected(convnet, 2, activation='softmax')
convnet = regression(convnet, optimizer='adam', learning_rate=LR, loss='categorical_crossentropy', name='targets')
model.fit({'input': X}, {'targets': Y}, n_epoch=1, validation_set=({'input': test_x}, {'targets': test_y}),
    snapshot_step=500, show_metric=True, run_id=MODEL_NAME)
```

Finally, you prepare the test data and predict the output.

```
test_data = process_test_data()

fig=plt.figure()

for num,data in enumerate(test_data[:12]):


    img_num = data[1]
    img_data = data[0]
    y = fig.add_subplot(3,4,num+1)
    orig = img_data
    data = img_data.reshape(IMG_SIZE,IMG_SIZE,1)
    #model_out = model.predict([data])[0]
    model_out = model.predict([data])[0]

    if np.argmax(model_out) == 0: str_label='Ronaldo'
    elif np.argmax(model_out) == 1: str_label='amitabh'

    y.imshow(orig,cmap='gray')
    plt.title(str_label)
    y.axes.get_xaxis().set_visible(False)
    y.axes.get_yaxis().set_visible(False)
plt.show()
```

Transfer Learning

Transfer learning makes use of the knowledge gained while solving one problem and applying it to a different but related problem.

Here you will see how you can use a pretrained deep neural network called the Inception v3 model for classifying images.

The Inception model is quite capable of extracting useful information from an image.

Why Transfer Learning?

It's well known that convolutional networks require significant amounts of data and resources to train.

It has become the norm for researchers and practitioners alike to use transfer learning and fine-tuning (that is, transferring the network weights trained on a previous project such as ImageNet to a new task).

You can take two approaches.

- *Transfer learning*: You can take a CNN that has been pretrained on ImageNet, remove the last fully connected layer, and then treat the rest of the CNN as a feature extractor for the new data set. Once you extract the features for all images, you train a classifier for the new data set.
- *Fine-tuning*: You can replace and retrain the classifier on top of the CNN and also fine-tune the weights of the pretrained network via backpropagation.

Transfer Learning Example

In this example, first you will try to classify images by directly loading the Inception v3 model.

Import all the required libraries.

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import os

# Functions and classes for loading and using the Inception model.
import inception
```

Now define the storage directory for the model and then download the Inception v3 model.

```
inception.data_dir = 'D:/'
```

```
inception.maybe_download()
```

Load the pretrained model and define the function to classify any given image.

```
model = inception.Inception()
```

```
def classify(image_path):
    # Display the image.
    p = Image.open(image_path)
    p.show()

    # Use the Inception model to classify the image.
    pred = model.classify(image_path=image_path)

    # Print the scores and names for the top-10 predictions.
    model.print_scores(pred=pred, k=10, only_first_name=True)
```

Now that the model is defined, let's check it for some images.

This gives a 91.11 percent correct result, but now if you check for some person, this is what you get:

It's 48.50 percent tennis ball!

Unfortunately, the Inception model seemed unable to classify images of people. The reason for this was the data set used for training the Inception model, which had some confusing text labels for classes.

You can instead reuse the pretrained Inception model and merely replace the layer that does the final classification. This is called *transfer learning*.

First you input and process an image with the Inception model. Just prior to the final classification layer of the Inception model, you save the so-called transfer values to a cache file.

The reason for using a cache file is that it takes a long time to process an image with the Inception model. When all the images in the new data set have been processed through the Inception model and the resulting transfer values are saved to a cache file, then you can use those transfer values as the input to another neural network. You will then train the second neural network using the classes from the new data set, so the network learns how to classify images based on the transfer values from the Inception model.

In this way, the Inception model is used to extract useful information from the images, and another neural network is then used for the actual classification.

Calculate the Transfer Value

Import the `transfer_value_cache` function from the Inception file.

```
from inception import transfer_values_cache
```

```
file_path_cache_train = os.path.join(cifar10.data_path, 'inception_cifar10_train.pkl')
file_path_cache_test = os.path.join(cifar10.data_path, 'inception_cifar10_test.pkl')
```

```
print("Processing Inception transfer-values for training-images ...")

# Scale images because Inception needs pixels to be between 0 and 255,
# while the CIFAR-10 functions return pixels between 0.0 and 1.0
images_scaled = images_train * 255.0

# If transfer-values have already been calculated then reload them,
# otherwise calculate them and save them to a cache-file.
transfer_values_train = transfer_values_cache(cache_path=file_path_cache_train,
                                              images=images_scaled,
                                              model=model)
```

```
Processing Inception transfer-values for training-images ...
- Processing image:   1021 / 50000
```

```
print("Processing Inception transfer-values for test-images ...")

# Scale images because Inception needs pixels to be between 0 and 255,
# while the CIFAR-10 functions return pixels between 0.0 and 1.0
images_scaled = images_test * 255.0

# If transfer-values have already been calculated then reload them,
# otherwise calculate them and save them to a cache-file.
transfer_values_test = transfer_values_cache(cache_path=file_path_cache_test,
                                             images=images_scaled,
                                             model=model)
```

As of now, the transfer values are stored in the cache file. Now you will create a new neural network.

Define the networks.

```
# Wrap the transfer-values as a Pretty Tensor object.
x_pretty = pt.wrap(x)

with pt.defaults_scope(activation_fn=tf.nn.relu):
    y_pred, loss = x_pretty.\
        fully_connected(size=1024, name='layer_fc1').\
        softmax_classifier(num_classes=num_classes, labels=y_true)
```

Here is the optimization method:

```
global_step = tf.Variable(initial_value=0,
                          name='global_step', trainable=False)
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(loss, global_step)
```

Here is the classification accuracy:

```
y_pred_cls = tf.argmax(y_pred, dimension=1)
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Here is the TensorFlow run:

```
session = tf.Session()
session.run(tf.global_variables_initializer())
```

Here is the helper function to perform batch training:

```
def random_batch():
    # Number of images (transfer-values) in the training-set.
    num_images = len(transfer_values_train)

    # Create a random index.
    idx = np.random.choice(num_images,
                           size=train_batch_size,
                           replace=False)

    # Use the random index to select random x and y-values.
    # We use the transfer-values instead of images as x-values.
    x_batch = transfer_values_train[idx]
    y_batch = labels_train[idx]

    return x_batch, y_batch
```

For optimizing, here is the code:

```
def optimize(num_iterations):
    # Number of images (transfer-values) in the training-set.


    start_time = time.time()

    for i in range(num_iterations):
        # Get a batch of training examples.
        # x_batch now holds a batch of images (transfer-values) and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = random_batch()

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimizer.
        # We also want to retrieve the global_step counter.
        i_global, _ = session.run([global_step, optimizer],
                                  feed_dict=feed_dict_train)

        # Print status to screen every 100 iterations (and last).
        if (i_global % 100 == 0) or (i == num_iterations - 1):
            # Calculate the accuracy on the training-batch.
            batch_acc = session.run(accuracy,
                                    feed_dict=feed_dict_train)
```

```
            # Print status.
            msg = "Global Step: {0:>6}, Training Batch Accuracy: {1:>6.1%}"
            print(msg.format(i_global, batch_acc))

    # Ending time.
    end_time = time.time()

    # Difference between start and end-times.
    time_dif = end_time - start_time

    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_dif)))))

    # Use the random index to select random x and y-values.
    # We use the transfer-values instead of images as x-values.
    x_batch = transfer_values_train[idx]
    y_batch = labels_train[idx]

    return x_batch, y_batch
```

For plotting the confusion matrix, here is the code:

```
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(cls_pred):
    # This is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_test,  # True class for test-set.
                          y_pred=cls_pred)  # Predicted class.

    # Print the confusion matrix as text.
    for i in range(num_classes):
        # Append the class-name to each line.
        class_name = "({}) {}".format(i, class_names[i])
        print(cm[i, :], class_name)

    # Print the class-numbers for easy reference.
    class_numbers = [" ({0})".format(i) for i in range(num_classes)]
    print("".join(class_numbers))
```

Here is the helper function for calculating the classifications:

```
# Split the data-set in batches of this size to limit RAM usage.
batch_size = 256

def predict_cls(transfer_values, labels, cls_true):
    # Number of images.
    num_images = len(transfer_values)

    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array.
    cls_pred = np.zeros(shape=num_images, dtype=np.int)

    # Now calculate the predicted classes for the batches.
    # We will just iterate through all the batches.
    # There might be a more clever and Pythonic way of doing this.

    # The starting index for the next batch is denoted i.
    i = 0

    while i < num_images:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_images)

        # Create a feed-dict with the images and labels
        # between index i and j.
        feed_dict = {x: transfer_values[i:j],
                     y_true: labels[i:j]}
```

```
        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_dict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Create a boolean array whether each image is correctly classified.
    correct = (cls_true == cls_pred)

    return correct, cls_pred
```

```
def classification_accuracy(correct):
    # When averaging a boolean array, False means 0 and True means 1.
    # So we are calculating: number of True / len(correct) which is
    # the same as the classification accuracy.

    # Return the classification accuracy
    # and the number of correct classifications.
    return correct.mean(), correct.sum()
```

```
def predict_cls_test():
    return predict_cls(transfer_values = transfer_values_test,
                       labels = labels_test,
                       cls_true = cls_test)
```

```
def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # For all the images in the test-set,
    # calculate the predicted classes and whether they are correct.
    correct, cls_pred = predict_cls_test()

    # Classification accuracy and the number of correct classifications.
    acc, num_correct = classification_accuracy(correct)

    # Number of images being classified.
    num_images = len(correct)

    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, num_correct, num_images))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)
```

Now let's run it.

```
from datetime import timedelta

optimize(num_iterations=1000)

Global Step:  13100, Training Batch Accuracy: 100.0%
Global Step:  13200, Training Batch Accuracy: 100.0%
Global Step:  13300, Training Batch Accuracy: 100.0%
Global Step:  13400, Training Batch Accuracy: 100.0%
Global Step:  13500, Training Batch Accuracy: 100.0%
Global Step:  13600, Training Batch Accuracy: 100.0%
Global Step:  13700, Training Batch Accuracy: 100.0%
Global Step:  13800, Training Batch Accuracy: 100.0%
Global Step:  13900, Training Batch Accuracy: 100.0%
Global Step:  14000, Training Batch Accuracy: 100.0%
Time usage: 0:00:36

print_test_accuracy(show_example_errors=True,
show_confusion_matrix=True)
```

```
Accuracy on Test-Set: 83.2% (277 / 333)
Example errors:
Confusion Matrix:
[108 3 5] (0) Aamir Khan
[0 83 22] (1) Salman Khan
[4 22 86] (2) Shahrukh Khan
 (0) (1) (2)
```

APIs

Many easy-to-use APIs are also available for the tasks of face detection and face recognition.

Here are some examples of face detection APIs:

- PixLab
- Trueface.ai
- Kairos
- Microsoft Computer Vision

Here are some examples of face recognition APIs:

- Face++
- LambdaLabs
- KeyLemon
- PixLab

If you want face detection, face recognition, and face analysis from one provider, currently there are three major giants that are leading here.

- Amazon's Amazon Recognition API
- Microsoft Azure's Face API
- IBM Watson's Visual Recognition API

Amazon's Amazon Recognition API can do four types of recognition.

- *Object and scene detection*: Recognition identifies various interesting objects such as vehicles, pets, or furniture, and it provides a confidence score.
- *Facial analysis*: You can locate faces within images and analyze face attributes, such as whether the face is smiling or the eyes are open, with certain confidence scores.
- *Face comparison*: Amazon's Amazon Recognition API lets you measure the likelihood that faces in two images are of the same person. Unfortunately, the similarity measure of two faces of the same person depends on the age at the time of the photos. Also, a localized increase in the illumination of a face alters the results of the face comparison.
- *Facial recognition*: The API identifies the person in a given image using a private repository. It is fast and accurate.

Microsoft Azure's Face API will return a confidence score for how likely it is that the two faces belong to one person. Microsoft also has other APIs such as the following:

- *Computer Vision API*: This feature returns information about visual content found in an image. It can use tagging, descriptions, and domain-specific models to identify content and label it with confidence.
- *Content Moderation API*: This detects potentially offensive or unwanted images, text in various languages, and video content.

- *Emotion API*: This analyzes faces to detect a range of feelings and personalize your app's responses.
- *Video API*: This produces stable video output, detects motion, creates intelligent thumbnails, and detects and tracks faces.
- *Video Indexer*: This finds insights in video such as entities of speech, sentiment polarity of speech, and audio timeline.
- *Custom Vision Service*: This tags a new image based on the built-in models or the models built through training data sets provided by you.

IBM Watson's Visual Recognition API can do some specific detection such as the following:

- It can determine the age of the person.
- It can determine the gender of the person.
- It can determine the location of the bounding box around a face.
- It can return information about a celebrity who is detected in the image. (This is not returned when a celebrity is not detected.)