University of Hawai'i

Face Recognition Analysis of Triplet Loss based Neural Networks

Author
Selim Karaoglu

Professor
Peter Sadowski

December 11, 2018

Abstract

Face recognition gained popularity in computer vision and the number of methods and techniques used in the process are increased. This paper provides experimental analysis on face recognition process with Triplet Loss function based network architectures. We implemented several different techniques to improve the overall computation time of the process - to be able to implement live detection on low computational resources - and we compared the results.

Introduction

This paper provides an analysis on Face Recognition based on FaceNet. After it's publication, FaceNet gained popularity and there are several open source libraries that are provided by publishers. Live face recognition with FaceNet is only possible on high-end graphics cards and CPUs but the majority of the libraries using FaceNet technique are not fast enough for live detection on devices with low computation power (Raspberry Pi, low-end pcs etc.). This paper provides experimental work on improving the Face Recognition computation time performance by applying different methods throughout the process. We split face recognition into pieces of subprocesses; preprocessing, encoding and classifying. Then we attempt to improve performance on each subprocess separately to improve overall computation time. In preprocessing, we applied several different methods for face detection and a few methods for landmark estimation, however kept transformation as it was, owing to the fact that function's computation is relatively too fast. For the encoding portion, we used weights from several pre-trained networks and trained Inception v2 model. For the classification portion, we planned to implement an SVM and a NN and compared them on performance and accuracy.

Background

In this section we will provide some background information for all aspects of the project. We start with theoretical background then we further explain the experimental background. This paper focuses on one a milestone paper in computer vision called FaceNet, mainly because it's one of the most accurate methods (with %99.63 accuracy) [1] and the popularity of the paper led to lots of sources we can work with.

FaceNet

Face recognition is a popular area in computer vision and there are numerous papers and methods published. One of these methods FaceNet, or with its full name; "FaceNet: A Unified Embedding for Face Recognition and Clustering", a paper published by Google in 2015 is particularly appealing for this project. The main reason we focused on FaceNet is its accuracy. With %99.63 accuracy on Lfw dataset FaceNet outperforms the other methods [2]. In addition to that, there are numerous open source libraries and different work on FaceNet method. In the paper, two different deep network archi-

tectures are explored, however we only focus on the Inception architecture published by Szegedy et al. FaceNet paper mainly focuses on comparing different neural networks on their accuracy (Inception Networks are only a part of it). We already know that FaceNet has shown a record accuracy, so instead focusing on the accuracy we focused on the processes that affect the computation time. Even though the subprocesses are not explained in details in the FaceNet paper, we implemented different techniques to compare against when applicable.

Libraries

This work heavily relies on open source libraries. A face recognition project with numerous different subprocesses requires a lot of code work, some of these open source libraries provided use especially on the performance improvement experiments. Although some of these libraries have dependencies. That's why we build different work environments throughout the process and applied the methods separately. Here we provide some information about the most important libraries.

dLib:

Dlib is a modern C++ toolkit containing machine learning algorithms and tools for creating complex software in C++ to solve real world problems [3]. Dlib is a library with hundreds of different tools, but in this project, we mainly used image processing and GUIs. This open source project is available in Linux only, however there are some binary files for use in Windows and MAC OSs. We first used these windows binaries for face detection and face landmark estimation, however GUI in Windows binaries are not supported, therefore getting any plot (or any visual output) was impossible. Therefore, we used dLib on windows for only face detection part and used other platforms (like UHHPC or docker) for any other subprocess.

OpenCV:

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms [4]. OpenCV library also does not support Windows OS. There are some workarounds like building it from

the source or building it from binaries, but these methods led us to lots of bugs and errors on the run. For the parts we used OpenCV, we used docker environments, which runs on Linux, so there were no problems on running.

openface:

OpenFace is a Python and Torch implementation of face recognition with deep neural networks and is based on the CVPR 2015 paper FaceNet... Torch allows the network to be executed on a CPU or with CUDA [5]. We used CPU mode with the openface library (CUDA is only available with an Nvidia graphics card). This library also provides two different pre-trained models of Inception v4 model used in this project.

facenet:

This is a TensorFlow implementation of the face recognizer described in the paper FaceNet [6]... This library provides Inception Resnet v2 pre-trained model.

facemark:

OpenCV's facial landmark API is called Facemark. It has three different implementations of landmark detection based on three different papers [7]. We used facemark API on face landmark estimation subprocess.

Mainly focusing on the experimental part in the project brings lots of coding, and considering there are compatibilities in the libraries led us to use multiple platforms to build the project. Mainly because the limitations, we still set the project on different platforms and transferred the output data by hand through different platforms. A major difficulty was to build the dLib library that almost every FaceNet application relies on. That led us to build the complex structure, however, with a Linux operating system, it's easy to build a smooth-running application. We provide detailed information about the environmental setup here.

Environments

We run code in different environments due to the limitations, but building a python 3 environment on Linux will be enough to run all the code provided

by this project. Since we start the experiments on a Windows OS running machine, we later used UHHPC because it's running on Linux OS. For the encoding part of the project, we used docker environments to run the code. Docker provides a pre-built environment that you can connect and use from command line. But another option is to build the libraries and using them locally which can be done in an environment running on Linux OS. To bring clarity, we provide details about each environment we used in this project.

Windows OS:

To built an environment in Windows OS machine, first step is to get anaconda and build a python 3.6 environment. Following packages and their dependencies need to be installed; dlib, matplotlib and scikit-image.

UHHPC:

We used UHHPC cluster, not to apply multi-threading, but to use the Linux environment it based on. To create an environment in UHHPC, first step is the same with Windows; to get anaconda and build a python 3.6 environment. Further, following libraries and their requirements should be installed; dlib, matplotlib, scikit-image, opencv (v2), cvlib.

Docker:

docker needs to be downloaded and installed. Running pre-built docker environments are easy, just need to download the environment and run the code. The code of docker environments are provided in appendix.

Data

As required by the nature of the project, the data we need is mainly face images. There are several different datasets available online. One of the popular face image datasets is Labeled Faces in the Wild (Lfw) dataset [16]. Another dataset we used is funneled version of Lfw (Lfw-a). These datasets are provided by University of Massachusetts. In addition to Lfw and Lfw-a, Olivetti Faces by scikit-image is another dataset we used [9]. We used all these datasets (separately and combined) for face detection, we fed this data to the classifier as positive patches. For negative patches we used extracted patches from the CIFAR-10 dataset [17]. For the embedding part of the

Figure 1: Examples from different datasets (Lfw, Lfw-a, Olivetti, Cifar-10)

project, we used some training images gathered by us, pictures of a known person under different lightning, angle and pose.

Considering all the different datasets we used, there are 13633 face images and 117000 patches from CIFAR-10 dataset. These datasets are sufficient to train a classifier. With only Lfw (13233) dataset and 50000 patches from CIFAR-10, our classifier shown 0.99 accuracy on training. With that on the side, we used Lfw again to train our Inception ResNet v2 model DNN.

Methodology

To reach the goals of the project, we applied step by step approach to face recognition process and split it into subprocesses, then we worked on each subprocess independently. On each subprocess applied several different methods and compared their performance and in final see if these possible improvements will affect the overall computation time of whole face recognition process. Obviously, we separate the training time for DNNs from this computation time, because training is done only once for the whole project. Training data shows a significant importance here. The majority of the face recognition projects uses Lfw dataset as training data. There are some experiments that used the funneled version of Lfw dataset [5], but since we applied preprocessing for each image (which is a necessity on face recognition as a whole process), Lfw is a better fit for our experiments.

In this section, we explained the methods used throughout the process in details. Necessary code and environment details are provided in appendix.

Preprocessing

This subprocess is the first step in the face recognition. In this step we take the input image, detect the faces in the image if there are any, find the

landmarks on the detected face and finally centralize the face for better encoding results. This subprocess can be applied in three steps; face detection, face landmark estimation and transformation. We will explain each step with different methods used for it.

Face Detection

Face detection is the step where we take the input image and detect the faces in it. To achieve this, we used several methods based on Histogram of Oriented Gradients (HOG). The Histogram of Oriented Gradient (HOG) feature descriptor is popular for object detection. Detecting humans in images is a challenging task owing to their variable appearance and the wide range of poses that they can adopt. The first need is a robust feature set that allows the human form to be discriminated cleanly, even in cluttered backgrounds under difficult illumination. We study the issue of feature sets for human detection, showing that locally normalized Histogram of Oriented Gradient (HOG) descriptors provide excellent performance [8]. We implemented the build in dlib function of face detection and wrote our own face detection function with scikit-image library's feature function [9]. There is another method for face detection provided by the dlib based on CNNs [10]. We implemented these three different methods and calculated the runtimes.

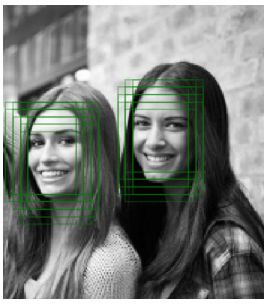

Figure 2: Output of our face detection method.

- **dLib built-in function:** Dlib library has a built-in function for face detection using HOG called $get\_frontal\_face\_detector()$. It's very easy to implement, very easy to use and computationally very fast.

- **Our function with skimage HOG:** We implemented our own linear SVM to use with feature.hog function provided by scikit-image library to extract HOG features of the image. This function first needs two group of images; positive patches and negative patches. We used HOG's of Lfw (also tried Lfw-a and Olivetti Faces and combinations) for positive patches and HOG's of patches extracted from CIFAR-10 dataset for negative patches. We cross validated the classifier and the accuracy was 0.99035 on best parameters. Since this training will not applied repeatedly, we didn't include it to computation time.

- **dlib CNN based face detector:** Dlib library also provides a CNN

Figure 4: Faces from same image with preprocessing applied

based face detection function called $cnn\_face\_detection\_model\_v1()$. This function has the same implementation with dlib's HOG function, however this rather new [11] technique does not have a good performance comparing the others.

Face Landmark Estimation

Face Landmark Estimation is where we detect the important features of the face and second step of preprocessing. This step takes detected (and extracted) faces (from previous step) and extracts the features [12] from it. We compared two different methods and compared computation times.

- **dLib 68 point landmark function:** Dlib library provides this pre-trained shape predictor. To be able to use this function, the model provided with the .dat format should be downloaded. After the model is defined, one line dlib function returns the face landmarks.
- **OpenCV - facemark function:** OpenCV provides a landmark estimation function based on three different algorithms [4]. We used the landmark estimation implemented from local binary features by Ren S. [13].

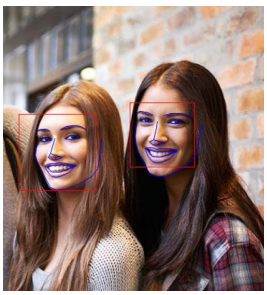

Figure 3: Output of dLib shape predict method.

Transformation

We applied face landmark estimation for only one reason, to be able to centralize the faces that are not captured from the front. Basically we are centralizing the images before we get the embeddings to increase the accuracy of recognition [14]. To achieve this goal we used OpenCV's affine transformation function [4].

Encoding

So far, we have built a system that takes the input image, detect the faces, detect landmarks of faces and centralize the faces. Now we need to turn those images to encodings. We use these encodings to calculate Euclidean distance [1] and classify the embeddings, in other words identify the person in the images. FaceNet paper emphasizes two different network architectures, Zeiler&Fergus model [15] and Inception model by Szegedy et al. and apllied these with Triplet Loss function [1]. For this paper we used two different Inception v4 DNN pre-trained weights and trained an Inception ResNet v2 model [2] with small number of epochs.

Even though all network architectures experimented in FaceNet paper is based on triple loss function, Inception network architectures outperform other network architectures. Comparing mean validation rates on the test data, two different Inception models (v2 and v3) perform with %89 validation rate. Zeiler&Fergus model follows up with a performance of %87 validation rate [1]. As we can see from the results published, there are not any significant difference on their performances.

This subprocess takes the centralized face images as input, and return 128 dimensional array for each image. This 128-D array is embeddings for each image. We run the models on our training images to get the embedding for each image. Even though we trained the Inception Resnet v2 model with small number of epochs (25 epochs took 9 hours) we also get the results from our trained network model.

Classification

This is the last step of our face recognition project. In this subprocess, we compare the face embedding from the input image with the images with known identities. This is a simple classification problem and there are several ways to achieve this goal. A linear SVM can do the classification with good accuracies in short period of time. This method is also applied in the FaceNet paper. In addition to SVM, we designed a simple Neural Network for this classification problem. First we train both models with training images. After we trained the models, we fed new images and evaluate the results. We compared the computational time for both methods.

Results

Here we compare the results we achieved for each experiment. After comparisons, we tried to improve the overall performance of FaceNet based recognition process to make live detection applicable on low computational sources. To be able to make a flawless live recognition, computation for the whole process should be executed in a shorter time than 0.042 seconds [5]. With low computation powers, this number is hard to achieve. We set docker environments with certain computation limits and run our applications within the environment. We picked a method from our best performing subprocesses and calculated the overall computation time. First we provide information on subprocesses.

Face Detection

We applied three different methods for face detection; dlib HOG, our own function based on skimage HOG and dlib CNN. The fastest calculation is done by dlib HOG, followed by our HOG function and dlib CNN is extremely slow comparing to these two methods.

On a docker environment with 0.9 Ghz processor and 4 GB of RAM, dlib HOG executed in 0.122 seconds, our HOG implementation executed in 0.148 seconds and dlib CNN executed in 4 seconds. If we execute the same code on a docker environment with 1 GB of RAM, the execution time results are 0.31, 0.45 and 6 seconds respectively. These results show us that even just the detection part is takes too long to execute on low resources. but with the implementation of multi-threading this problem might be solved.

Face Landmark Estimation

For face landmark estimation, we compared two different methods; dlib's shape predictor and opencv's facemark. Both methods resulted with close performances and both are very fast. On a docker with 4 GB RAM environment dlib executed in 0.011 seconds, facemark followed with 0.013 seconds. On a docker with 1 GB RAM dlib performed in 0.037 and facemark performed in 0.041 seconds.

Transformation

To centralize the images, we used simple 2 dimensional transformation. This transformation is computationally more economic than 3-D transforma-

tions [5]. Like landmark estimation, this subprocess is also fast to execute. On 4 GB docker environment transformation executed in 0.0048 seconds and on 1 GB docker environment execution finished in 0.011 seconds.

Encoding

Using different DNN's to get the face embeddings will not affect the computation time, therefore we did not compare the encoding performance.

Classification

The last subprocess we experimented in this paper is the classification of the embeddings and getting the results. Since we already trained our network, we can feed the images and get the encodings for the image, but to tell if the encodings match with a known person is provided by classification. FaceNet paper uses a linear SVM for classification. We implemented a neural network for this task and compared both methods.

Both SVM and NN works with the same flow; take the embedding as an input than compare the embedding to known embeddings. Both methods needs to be trained on training images. After the training classification is fast (related to training) but still this performance is a bottleneck for the live detection. SVM completes the classification task in 0.56 seconds. Our neural network slightly performs better with 0.49 seconds. However, this performance improvement is still not enough to implement the whole process for live detection (For this last part, we used Linux OS environment with 0.9 Ghz CPU and 4 GB RAM).

Conclusion

Face recognition process needs a decent computation power to be adapted for live detection. As authors stated in the openface documentary, it is possible to do live detection with high-end graphics cards (using CUDA) or with whole process run with multi-threading. We tried to apply different methods for subprocesses and decrease the computation time. Some of the methods we applied can decrease the computational time, however this change won't be sufficient to do live detection on devices run on low computational specifications.

For preprocessing part; using dLib's HOG for face detection, dLib's shape predictor for landmark estimation and applying affine transformations with OpenCV seems the smoothest method. Encoding experiments don't affect the overall performance. Classification is slightly faster with a neural network implementation. With these techniques applied together, we achieved faster computations, but due to time limitations (and complex environment setup and varieties of network architectures) we still couldn't build a single application setup.

Future Work

There are some improvements can be achieved and there are improvements can be applicable without current limitations;

- Face detection function build for this project can be improved with using local search in the sliding window function. This function uses naive algorithm to extract patches in the image.
- With more computational power, network architectures can be trained on bigger dataset with a larger number of epochs.
- This project can be compiled with the fastest working methods and presented as a stand-alone face recognition project.
- FaceNet paper suggests they used 260M images to train the network, if possible larger datasets can provide higher accuracies (especially for the first and the last classification steps).

References

[1] Florian Schroff, Dmitry Kalenichenko, and James Philbin, *FaceNet: A Unified Embedding for Face Recognition and Clustering* DOI:10.1109CVPR.2015.7298682. 2015

[2] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, Alexander A. Alemi, *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning* 2016.

[3] Davis King, *Dlib C++ library* Url: http://dlib.net/. 2008.

[4] Gary Bradski, Adrian Kaehler *OpenCV* Dr. Dobb's journal of software tools, 2008.

[5] B. Amos, B. Ludwiczuk, M. Satyanarayanan, *Openface: A general-purpose face recognition library with mobile applications* MU-CS-16-118, CMU School of Computer Science, Tech. Rep., 2016.

[6] David Sandberg, *facenet: Face recognition using Tensorflow* Url: https://github.com/davidsandberg/facenet, 2017.

[7] Satya Mallick, *Facemark : Facial Landmark Detection using OpenCV* https://www.learnopencv.com/facemark-facial-landmark-detection-using-opencv/, 2018

[8] Navneet Dalal, Bill Triggs, *Histograms of Oriented Gradients for Human Detection* 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), 2005.

[9] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors, *scikit-image: Image processing in Python* PeerJ 2:e453,Url: http://dx.doi.org/10.7717/peerj.453, 2014.

[10] Arun Ponnusamy, *CNN based face detector from dlib* Url: https://www.arunponnusamy.com/cnn-face-detector-dlib.html, 2018.

[11] Davis King, *Face detection tutorial* Url: http://dlib.net/cnn_face_detector.py.html, 2017.

[12] David Kazemi, Josephine Sullivan *Face Alignment with Part-Based Modeling* CVAP, KTH Institute of Technology, Stockholm, Sweden. 2011.

[13] Shaoqing Ren, Xudong Cao, Yichen Wei, Jian Sun *Shaoqing Ren, Xudong Cao, Yichen Wei, Jian Sun* The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1685-1692, 2014.

[14] Xiangxin Zhu ; Deva Ramanan *Face detection, pose estimation, and landmark localization in the wild* 2012 IEEE Conference on Computer Vision and Pattern Recognition, 2012.

[15] Matthew D Zeiler, Rob Fergus *Visualizing and Understanding Convolutional Networks* ECCV 2014. Lecture Notes in Computer Science, vol 8689. Springer, Cham, 2014.

[16] Erik Learned-miller, Gary Huang, Aruni Roychowdhury, Haoxiang Li, Gang Hua, Erik Learned-miller, Gary B. Huang, Aruni Roychowdhury, Haoxiang Li *Labeled Faces in the Wild: A Survey* doi=10.1.1.696.1377, 2007.

[17] Krizhevsky, Alex *Learning Multiple Layers of Features from Tiny Images* 2009.

Appendix

In this section we provide detailed information about the environments and the code we used for the project.

Environments

- **Windows OS Environment:** We started the project on a Windows OS, therefore the code we provided is designed for such environment. First we need an anaconda environment running python 3.6. Then we need some packages and their requirement installed. These packages are; matplotlib, numpy, pickle, scikit-image, scikit-learn, pillow and dlib.

- **UHHPC Environment (Linux based):** We used UHHPC servers mainly because we needed a Linux OS environment. The requirements are the same with Windows OS environment with an extra package; opencv (OpenCV library is not fully compatible with Windows OS).

- **docker:** We set up two different docker environments for this project with only difference, allocated RAM size. First environment runs with 4 GB RAM and the second runs with 1 GB RAM. This implementation is purely for experimenting the computational performance. On docker environment we need anaconda with python 3.6 with these packages and their requirements installed; matplotlib, numpy, pickle, scikit-image, scikit-learn, pillow, dlib, opencv, tensorflow, pytorch, openface, facenet. It is possible to download and build the libraries openface and facenet, we still find it useful to use the pre-built docker environments provided by the publishers.

Code

Programming work required for this project was heavy, for a lot of coding part of the project we used openface, facenet and face_recognition libraries and their work to derive our code [5][6][7].

Face Detection and Landmark Estimation methods use these libraries:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

```
import pickle as pkl
import skimage.data
from PIL import Image
from skimage import data, color, feature, transform
from sklearn.datasets import fetch_lfw_people,
    fetch_olivetti_faces
from sklearn.feature_extraction.image import PatchExtractor
from itertools import chain
```

Face Detection with skimage HOG

```
f2 = fetch_lfw_people()
p2 = f2.images
pos_patches = np.asarray(p2)
pos_patches.shape

imgs = []
for i in range(1,13234):
    name = np.asarray(Image.open("data/lfwa/ ("+str(i)+").
    jpg"))
    n = color.rgb2gray(name)
    imgs.append(n)

pos_patches = np.asarray(imgs)
pos_patches = np.concatenate((pos_patches, pos))
with open('pos_p.pkl', 'wb') as f:
    pkl.dump(pos_patches, f)

def patch_extraction(img, N, scale=1.0, patch_size=
    pos_patches[0].shape):
    extracted_patch_size = tuple((scale * np.array(
    patch_size)).astype(int))
    extractor = PatchExtractor(patch_size=
    extracted_patch_size, max_patches=N, random_state=0)
    patches = extractor.transform(img[np.newaxis])
    if scale != 1:
        patches = np.array([transform.resize(patch,
    patch_size, mode='constant' )
                            for patch in patches])
    return patches

negative_patches = np.vstack([patch_extraction(im, 1000,
    scale)
                              for im in images for scale in
    [0.5, 1.0, 2.0]])
imgs = []
for i in range(0,5000):
```

```
30      n = np.asarray(Image.open("data/img/"+str(i)+".png").
        crop((12,9,299,296)))
31      n = color.rgb2gray(n)
32      imgs.append(n)
33
34  neg = np.asarray(imgs)
35  neg_patches = np.vstack([patch_extraction(im, 1, scale)
36                                  for im in neg for scale in
        [0.5, 1.0]])
37  neg_patches.shape
38  neg_patches = np.concatenate((neg_patches, negative_patches)
        )
39
40  with open('neg_p.pkl', 'wb') as f:
41      pkl.dump(neg_patches, f)
42
43  X_train = np.array([feature.hog(im, block_norm="L1") for im
        in chain(pos_patches, negative_patches)])
44  y_train = np.zeros(X_train.shape[0])
45  y_train[:pos_patches.shape[0]] = 1
46  from sklearn.svm import LinearSVC
47  from sklearn.model_selection import GridSearchCV
48  params = [{'C':[0.01, 0.05, 0.1, 0.5, 1, 2, 5, 10]}]
49  grid = GridSearchCV(LinearSVC(), params, cv=5)
50  grid.fit(X_train, y_train)
51
52  model = grid.best_estimator_
53  model.fit(X_train, y_train)
54
55  test_image = skimage.data.astronaut()
56  test_image = skimage.color.rgb2gray(test_image)
57  test_image = skimage.transform.rescale(test_image, 0.5,
        multichannel=False)
58  test_image = test_image[:160, 40:180]
59
60  def sliding_window(img, patch_size=pos_patches[0].shape,
61                     istep=5, jstep=5, scale=1):
62      Ni, Nj = (int(scale * s) for s in patch_size)
63      for i in range(0, img.shape[0] - Ni, istep):
64          for j in range(0, img.shape[1] - Ni, jstep):
65              patch = img[i:i + Ni, j:j + Nj]
66              if scale != 1:
67                  patch = transform.resize(patch, patch_size)
68              yield (i, j), patch
69
70  indices, patches = zip(*sliding_window(t))
71  patches_hog = np.array([feature.hog(patch, block_norm="L1")
        for patch in patches])
72  labels = model.predict(patches_hog)
```

Face detection with dLib HOG

```
def face_detection_dLib(file_name):
    detected_faces = dlib.get_frontal_face_detector()(image, 1)
    return detected_faces

file = "<img path>"
image = io.imread(file)
detected = face_detection_dLib(file)
```

Face detection with dLib CNN

```
weights = "mmod_human_face_detector.dat"
file = "<img path>"
image = io.imread(file)
def face_detection_cnn(file_name):
  cnn_face_detector = dlib.cnn_face_detection_model_v1(weights)
    detected_faces = cnn_face_detector(image, 1)
    return detected_faces

file = "<img path>"
image = io.imread(file)
detected = face_detection_cnn(file)
```

Landmark Estimation with dLib

```
predictor_model = "shape_predictor_68_face_landmarks.dat"
weights = "mmod_human_face_detector.dat"
image = io.imread(file)
detected_faces = dlib.get_frontal_face_detector()(image, 1)
#detected_faces2 = dlib.cnn_face_detection_model_v1(weights)

def face_landmarks(file):
    face_pose_predictor = dlib.shape_predictor(predictor_model)
    win.set_image(image)
    for i,face_rect in enumerate(detected_faces):
        win.add_overlay(face_rect)
        pose_landmarks = face_pose_predictor(image, face_rect)
        win.add_overlay(pose_landmarks)
    return pose_landmarks

image_landmarks = face_landmarks("<img path>")
```

Landmark Estimation with facemark

```
obj = cv2.face.createFacemarkLBF()
model = "lbfmodel.yaml"
obj.loadModel(model)
landmarks = obj.fit(img, faces)
```

Transformation

For this part we used aligner.py file that is designed just for this task.

```
import cv2
import dlib
import numpy as np

#Load template from pickle file
with open('template1.pkl', 'rb') as f:
    TEMPLATE = pkl.load(f)
TPL_MIN, TPL_MAX = np.min(TEMPLATE, axis=0), np.max(TEMPLATE
    , axis=0)
MINMAX_TEMPLATE = (TEMPLATE - TPL_MIN) / (TPL_MAX - TPL_MIN)

class Dlib_Align:
    INNER_EYES_AND_BOTTOM_LIP = [39, 42, 57]
    OUTER_EYES_AND_NOSE = [36, 45, 33]

    def __init__(self, facePredictor):
        self.detector = dlib.get_frontal_face_detector()
        self.predictor = dlib.shape_predictor(facePredictor)

    def getAllFaceBoundingBoxes(self, rgbImg):
        return self.detector(rgbImg, 1)

    def getLargestFaceBoundingBox(self, rgbImg, skipMulti=
    False):
        faces = self.getAllFaceBoundingBoxes(rgbImg)
        if (not skipMulti and len(faces) > 0) or len(faces)
    == 1:
            return max(faces, key=lambda rect: rect.width()
    * rect.height())
        else:
            return None

    def findLandmarks(self, rgbImg, bb):
        points = self.predictor(rgbImg, bb)
        return list(map(lambda p: (p.x, p.y), points.parts()
    ))

    def align(self, imgDim, rgbImg, bb=None,
              landmarks=None, landmarkIndices=
    INNER_EYES_AND_BOTTOM_LIP,
```

```
35                 skipMulti=False):
36             if bb is None:
37                 bb = self.getLargestFaceBoundingBox(rgbImg,
    skipMulti)
38                 if bb is None:
39                     return
40             if landmarks is None:
41                 landmarks = self.findLandmarks(rgbImg, bb)
42             npLandmarks = np.float32(landmarks)
43             npLandmarkIndices = np.array(landmarkIndices)
44             H = cv2.getAffineTransform(npLandmarks[
    npLandmarkIndices], imgDim * MINMAX_TEMPLATE[
    npLandmarkIndices])
45             thumbnail = cv2.warpAffine(rgbImg, H, (imgDim,
    imgDim))
46
47             return thumbnail
```

Encoding with pre-trained models and Classification

To train the pre-trained models on our own data, we used the pre-built docker environments provided by the openface library [5]. To use to automated docker build;

```
1 docker pull bamos/openface
2 docker run -p 9000:9000 -p 8000:8000 -t -i bamos/openface /
    bin/bash
3 cd /root/openface
```

The project has a large context so being acknowledged about the environment is important. This library provides different pre-trained models in openface/models directory. We worked on NN4 (Inception v4a) and NN2 (Inception v4b) network architectures from openface [5]. In addition to these two Inception v4 models, we implemented a Inception Resnet v2 based network architecture model provided by facenet[6]. We built the facenet library on a Linux environment;

```
1 python ./src/classifier.py TRAIN ./datasets/lfw/
    lfw_mtcnnalign_160 ./models/model-20170216-091149.pb ./
    models/lfw_classifier.pkl --batch_size 128 --
    min_nrof_images_per_class 10 --
    nrof_train_images_per_class 35 --use_split_dataset
```

This code outputs the trained weights in pickle format;

```
1 Number of classes: 19
2 Number of images: 665
```

```
Loading feature extraction model
Model filename: ./models/model-20170216-091149.pb
Calculating features for images
Training classifier
Saved classifier model to file "./models/lfw_classifier.pkl"
```

We can use this file for classification on test set;

```
python ./src/classifier.py CLASSIFY ./datasets/lfw/
    lfw_mtcnnalign_160 ./models/model-20170216-091149.pb ./
    models/lfw_classifier.pkl --batch_size 128 --
    min_nrof_images_per_class 10 --
    nrof_train_images_per_class 35 --use_split_dataset
```

Here's the results for the test portion of the Lfw dataset;

```
Number of classes: 19
Number of images: 1202
Loading feature extraction model
Model filename: ./models/export/model-20170216-091149.pb
Calculating features for images
Testing classifier
Loaded classifier model from file "./lfw_classifier.pkl"
   0  Abdullah Gul: 0.583
   1  Abdullah Gul: 0.611
   2  Abdullah Gul: 0.670
...
...
...
1198  Zhu Rongji: 0.688
1199  Zhu Rongji: 0.723
1200  Zhu Rongji: 0.656
1201  Zhu Rongji: 0.751
Accuracy: 0.981
```

NN for Classification

Our implementation of a basic neural network for the classification part.

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

train = "<train data embeddings>"
train_l = "<train data labels>"
test = "<test data embeddings>"
test_l = "<test data labels>"

```

```
_in = train[0].shape
model = keras.Sequential([
    keras.layers.Flatten(input_shape=_in),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(4, activation=tf.nn.softmax)
])
model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train, train_l, epochs=25)
test_loss, test_acc = model.evaluate(test, test_l)

print('Test accuracy:', test_acc)
predictions = model.predict(test)
print(predictions[0])
[ 0.07214264  0.4383606   0.23599184  0.25350484]
```