CSC420 Project Report on Ethnicity Recognition

Xinyan(Celina) Jiang
1001469289

Introduction

My project is on facial alignment, face detection and Ethnicity recognition. Basically, when given a photo with a group of people, my program is able to detect all the faces present in the photo, draw a bounding box around each detected face, adjust the facial landmarks, then determine and label the ethnicity of each person. Since there exists many ethnicity, to simplify things, I have included only five major ethnicity that I can gather enough data to train a CNN on, and they are: White, Black, Asian, Indian, and Others. Another reason that I decided on my ethnicity labels is that the online dataset in which I used to train my neural network on is pre-labeled with the five ethnicity. Hence, it will be easier to just use what was given. Furthermore, the reason that I picked this project is because I did gender classification using linear regression in CSC321, and I wanted to create a ethnicity recognition classifier to combine it with my gender classifier. Using these two classifiers, I can then have a caption generation program that can describe the ethnicity and gender of every person in a photo.

In the start of the project, I read numerous papers on how to approach ethnicity recognition. One of the first papers I read is called ”*Prediction of Human Ethnicity from Facial Images Using Neural Networks*” [(3)], which compared two methods of training a ethnicity classifier. It first talked about the extraction of skin colour as a feature to classify ethnicity. Since RGB colour space varies largely in intensity, this paper introduced a YCbCr color model where images were converted from RGB to YCbCr color space using two equations. Then it also introduced geometric feature calculations where different ratios of distances between facial areas were computed as features. For instance, the ratio of the distance from left eye to nose vs. the distance of right eye to nose is extracted as a feature. Lastly, the paper proceeds to introduce the normalized forehead area calculation where the ratio of forehead area to total face area was computed. Note that article mentioned that “the forehead to face ratio is found to be more than 25% for Negros and less than 25% for Mongolians. In case of Caucasians, this ratio varies largely.” Finally, they ran an Artificial Neural Network using multi-layered perceptrons on these extracted features. The alternative method mentioned in this paper was to use CNN, where the samples were tested on VGGNet. In the end, the paper concluded that the “performance of the CNN approach is far superior than ANN”, with the accuracy achieved in ANN be 82.4% while in CNN was 98.6%. Furthermore, in another paper titled “Learning ethnicity from Face: A Survey”[(**?**)], skin tone as a “”variable visual feature within any given ethnicity is actually one of the least important factors in distinguishing between ethnicity.” Since VGGNet outperformed the ANN approach and skin tone turned out to be a very insignificant feature, I decided to use CNN to train for the ethnicity classifier. However, I also decided to generate my own CNN instead of using the VGGNet. Since skin colour isn’t the most essential feature.

The hardest in this project would probably be the training of the images using CNN since there are many hyperparameters that require tuning. A limitation would be the under-representation of certain races in the training set, which could lead to a less robust classifier due to the lack of variations.

Methods

1. Data Processing

To begin, my partner found the online dataset, called UTKFace [(7)], that I used to train the CNN. It is a dataset containing over 20,000 images with annotations of age, gender, and ethnicity. The dataset covered an expansive range of ages from 0 to 116, and it contained images with a large variation in pose, facial expression, illumination, occlusion and resolution. Hence, training my CNN with these varied photos could generate a more accurate model during test time. The dataset after downloading and unzipping revealed a folder with aligned and cropped faces that has over 23,709 images in total. My partner and I wrote the code to first rename all the images with only the information we need, then split them into 60% training, 20% validation and 20% test sets. The code implementation can be found in the file called **data.py**, and the first function we used is called **process_data()**. Because we discovered that some images were labeled incorrectly with missing classes, in **process_data()**, we loop over every single image name and only split the ones that are labeled correctly. Note that we first split them by ethnicity, then by gender. The reason for doing so is that when splitting for training, validation and test sets, we wanted to have a fair proportional representation of each gender in each ethnicity, in case that the random split might split all males into training, and females in test sets. The resulting folder of images is named "dataset", and the general view of the split is shown in the right image of **Figure 1**.

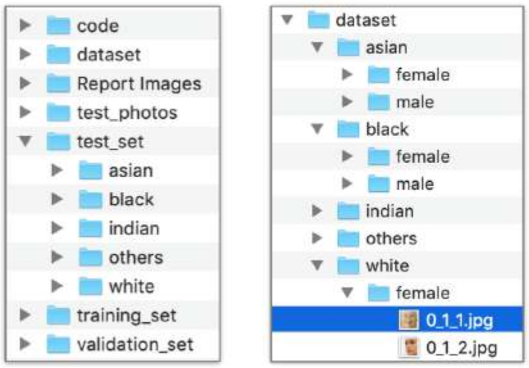
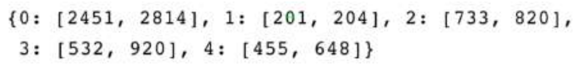


Figure 1: Folder paths

Note that a dictionary called count was used to store the total number of male and female images under each ethnicity, and it is shown in **Figure 2**. The keys in the dictionary are indices to the list called races:

races = ["white", "black", "asian", "indian", "others"]



```
{0: [2451, 2814], 1: [201, 204], 2: [733, 820],
 3: [532, 920], 4: [455, 648]}
```

Figure 2: Dictionary variable count

Next, my partner and I implemented the function called **split_data(total_count)** which takes in the count dictionary and split the gender in each ethnicity by 60% training, 20% validation and 20% test sets to ensure a fair representation. The resulting folders can be seen in **Figure 1** to the left. Note that the three sets only only split by ethnicity and no longer further split by gender, and [male, female] = [0, 1].

2. Build the Training and Validation Matrices

Before training, I wrote a function in **cnn_model.py** that generates the training matrix and its corresponding labels, called **build_data(data_type, size)**, when given the type of the data (i.e. training, validation or test), and the desired size of the sets (i.e. 20, 500, "all"). The program first uses the **data_type** of the matrix to determine the file path to the images, then it computes the actual data size for each ethnicity, represented by a variable named **ethnicity_size**, to ensure that no error would have occurred if we entered a number bigger than the number of images we have. This could be found in line 38-49. Next, we loop over the images in each ethnicity file and first extract labels, then read the image and resize to $(50, 50, 3)$ instead of the original $(200, 200, 3)$ in order to speed up the building of the matrix and training. This is found on lines 51-70. Also, since both my partner and I use a MacBook, we also have to make sure that ".DS_Store" is not appended to the matrix. After creating the matrices and labels for each ethnicity and storing the data in **ethnicity_dict** and **laels_dict**, we merged every ethnicity's data together into one variable named **data**. Likewise, we merged all the corresponding labels into the variable called **data_labels**. This implementation is found on lines 72-87.

Then, in **main.py**, I wrote the option to save the matrices into a txt file called **model_sets.txt** if they haven't been created before using Pickle, or load the matrices if it is already stored. This can be found on lines 28-55. Note that this file is stored under the path "code/colour_files".

3. Training a CNN for ethnicity Recognition

After building the validation and training matrices, I wrote the functions used for initializing and training for the ethnicity recognition CNN. In the file **cnn_model.py**, I first wrote **initialize_model()** in lines 94-119 where I specified the layers of my CNN. Note that I modified the CNN architecture from the paper called "ethnicity Recognition Based on Convolutional Neural Network." [(2)]. Originally, I wanted to use VGGNet. However, VGGNet has 16 convolutional layers that takes 2-3 weeks to train using 4 GPU's, which is impossible for me given the short time. In this paper, it mentioned an eight-layer CNN structure that executes convolution, max pooling, convolution, max pooling, convolution, max pooling, convolution, and convolution respectively. Then, after extracting all features, the paper applied the Softmax activation function on a fully connected layer. Since each of my training images were resized to 50-by-50 for space and speed purposes, I eliminated a max pooling layer and only used 3-by-3 filters in each convolution layers to prevent the input before entering each layer become too small. I also dropped out some random nodes in order to eliminate noise. Further, another convolutional layer was added before the softmax layer to imitate the 2 fully connected layers used in AlexNet at the very end. Before testing out the different network architectures, I wrote the **train_network()** function that takes in the training and validation matrix plus their labels, the CNN model I initialized, and the batch size for training. Note that this function can be found in lines 120-145. This code is taken from my CSC420 Assignment 5 where I used AdamOptimizer with a learning rate of 0.001 and the sparse categorical cross-entropy because the labels are indices to a ethnicity array. If I were to use a one-of-K encoding, I would have to use categorical cross-entropy instead. But since my assignment's result was very good, I decided to use the same parameters. I also generated the accuracy and loss graphs in this function, and the accuracy graphs can be found in the Discussion section. After **model.fit** has been called and my CNN is trained with the best possible hyperparameters, I returned the model to the model variable in **main.py**, line 81. Note that from lines 77-88, I rained a model and stored it to JSON, and I saved weights to HDF5 for faster access. Next time, if the model and wieghts have already been generated and stored, I could simply read them from file, and compile them quickly using lines 58-74. Note that I wrote the entire **main.py** and **cnn_model.py** on my own, and I performed all trainings on my own. My partner implemented the SVM method instead of a CNN. Lastly, the code for loading and saving the model and weights using JSON is taken from a website titled ”Save and Load Your Keras Deep Learning Models” [(Brownlee)].

4. Face Detection

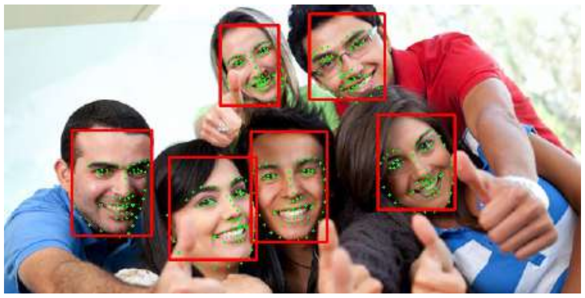


Figure 3: Output image after applying the functions **detect_face** and **find_landmarks**

After a good ethnicity recognition model has been generated and saved, I wrote a commandline input parser that takes the path of the input image, and the path of the output image after detection. This implementation is found on lines 91-96 in **main.py**. Then, my partner and I wrote the **detect_face(input_img_path)** function in **face.py** for detecting faces and returning the coordinates required to draw each bounding box. Note that this implementation is found online from a website titled “Face Detection with OpenCV and Deep Learning” [(5)]. Note that we used OpenCVs pre-trained deep neural network module with Caffe models for face detection. Essentially, the detector is based on the Single Shot Detector (SSD) framework using ResNet as the base network. The file **deploy_prototxt.txt** stored in the file folder contains the face dettection model's architecture, whereas the **res10_300x300_ssd_iter_140000.caffemodel** file stores the weights for the actual neural network layers. From line 13-20 in **face.py**, we first read the two aforementioned files, then from 24-27, we read the input image, then use the library function **cv2.dnn.blobFromImage** to perform mean subtraction and scaling of the input image. In our cause, our scale factor value is 1.0 (i.e., no scaling), our input image is resized to 300-by-300, and the 3-tuple of the RGB means, (104.0, 177.0, 123.0), is taken from the website. The function then returns a blob of our input image which can then be passed into the face detection neural network on lines 31-32. After the face detection are computed, we remove the ones with confidence lower than 0.5, and find the top left and bottom right corner of the bounding box. Then we append it to result and return the detection to **main.py**. **Figure 3** displays the image after bounding boxes are applied.

5. Facial Landmark Alignment

Continue from step 4, I began looping over the detected boxes on line 107, and perform facial alignment on each detected face in the main file. Note that I wrote the **find_landmarks** and **align_face** function found in **face.py** where some lines of the code were taken from the website called “Facial landmarks with dlib, OpenCV, and Python”[(6)]. To begin, after determine the location of each face in the image, in the function **find_landmarks**, I first take in the coordinates of the bounding box obtained from **detect_face()**, and the path to the input image. Then I load a facial landmark predictor from the file **shape_predictor_68_face_landmarks.dat** using the library function **dlib.shape_predictor**. From lines 55-62, I passed in the grayscaled input image and the coordinates of the bounding box into the predictor function in order to obtain the landmarks. Then from lines 66-71, I append the actual coordinates of the facial landmarks on the detected face into the variable called coords. After the coordinates of the landmarks are returned to the main file under the variable coords on line 109, I then pass it into the **align_face** function to perform actual facial alignment. Note that the facial landmarks I am using is from the library called dlib. **Figure 3** also shows the facial landmarks.

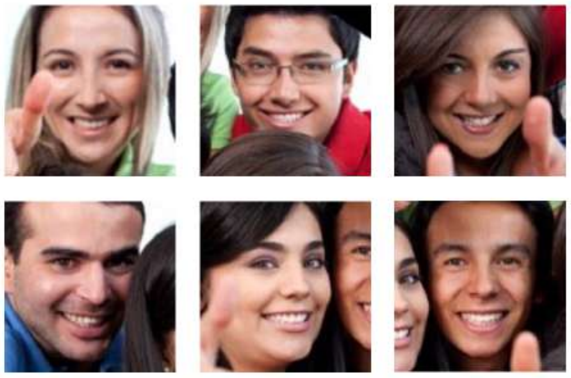


Figure 4: The aligend faces from **Figure 5**

To align the face, I first initialized the ratio of the desired eyes position and the desired face size, the I extract the actual coordinates of the left eye and right eye in lines 93-96 to compute the center of mass between the eyes. Then I used the arctan function to compute the angle of rotation between eyes so that I can rotate the eyes later to make them lie on the same horizontal line. Next, I calculated the scale between the desired distance and the actual distance of the eyes from lines 108-110 so that I can use them to compute the affine transformation matrix M. Next I computed the coordinates of the midpoint between the two eyes and pass it into the library function **cv2.getRotationMatrix2D** along with the angle and scale I computed in order to obtain the 2D affine transformation matrix M. Next, I updated the translation component of the matrix on lines 120-121 to center the image later by subtracting the midpoint. Afterwards, I applied the affine transformation on input image using M and returned the aligned face to the main function using **cv2.warpAffine()**. Note that some of the calculations are taken from the website "Face Alignment with OpenCV and Python"[(4)], and the aligned faces from **Figure 5** are displayed in **Figure 5**. Notice that the slanted faces have been centered and adjusted straight.

6. Ethnicity Prediction

Finally, the ethnicity of each detected/aligned face were predicted by passing each face into the function **model.predict()** as an argument. The result is an array of 5 elements with values between [0, 1] where the index with the highest value determines the ethnicity of the face. For instance, if the resulting array is [0.1, 0, 0.2, 0.8, 0.15], then the likely ethnicity would be "indian" (refer back to page 2, left column above **Figure 2** for the race array).

Results and Discussions

In conclusion, my CNN model achieves around the same accuracy as my partner's SVM. However there are some cases where one of our method works better than the other's. For instance, **Figure 5** shows that SVM performs better, whereas **Figure 6** shows that CNN performs more accurately.

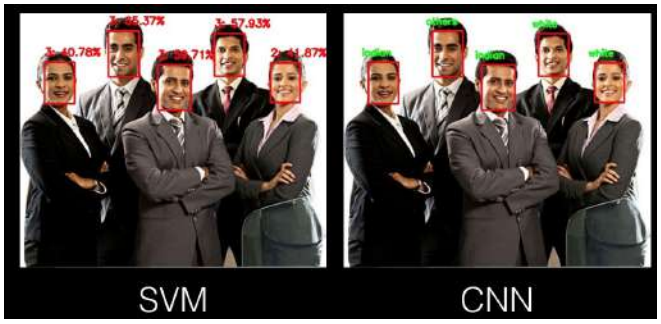


Figure 5: SVM makes better predictions than CNN

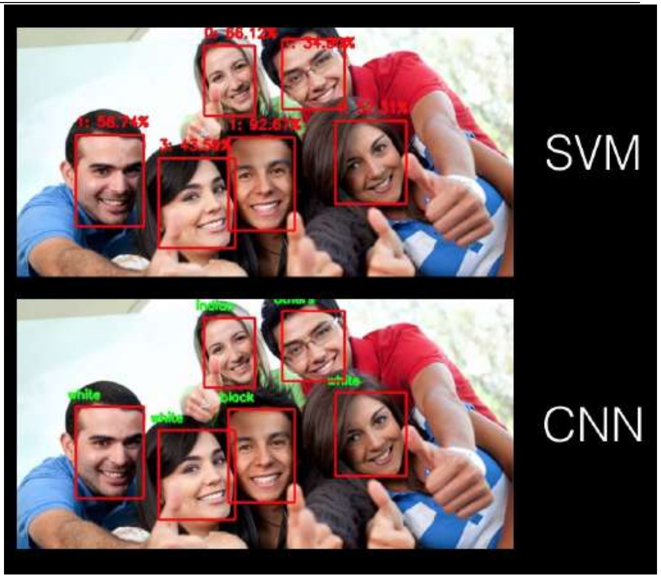
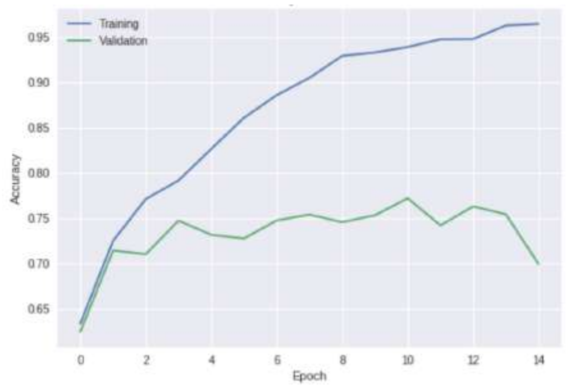


Figure 6: CNN makes better predictions than SVM

Initially, the paper "ethnicity Recognition Based on Convolutional Neural Network." [(2)] also mentioned to apply two batch normalization layers in the CNN in order to accelerate training and improve accuracy. However, when I tried its implementation, the validation accuracy fluctuates and dips after epoch 14, while the training accuracy still increases (refer to **Figure 7**). This could mean an overfitting on the training set. If I take out the batch normalization layers, the accuracy becomes slightly better (refer to **Figure 8**). Hence, I decided to implement without the normalization layers. Further, despite playing around with the layers, for instance adding more convolution layers or applying less max pooling or dropout, the highest validation accuracy achieved still revolves around 78%.



fa

Figure 7: Accuracy of ethnicity Recognition model when 2 batch normalization layers are applied

Lastly, my method fails due to the below reasons:

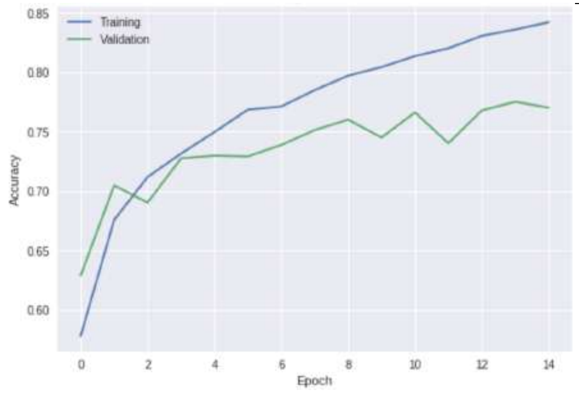


Figure 8: Accuracy of ethnicity Recognition model when 0 batch normalization layers are applied

1) **Size and distribution of the dataset**: Examining **Figure 2**, some ethnicity, such as Black, has only around 400 images, whereas the white people dataset contains more than 5000 images. Because the dataset for Black people is too small, my CNN model is less robust to variations, which explains why some Black people were misclassified to be White in my input images, but mostly all White people are correctly classified. (refer to **9**).

2) **Lack of variations in our training images**: Some side faces can be detected but not recognized correctly due to the lack of side face data in our training set. This is shown in **Figure 10**. Contrarily, all of the fontal face were detected and mostly correctly identified because my trianing images consisted of frontal faces only.

3) **Face detector fails to identify a face**: When some faces are too small in size, the face detector I used might be unable to even detect a face. However, if faces are big enough, the face detector is able to draw a box around them for later recognition. This is shown in **Figure 11**.

4) **My CNN wasn't good enough**: Other CNN models such as ALexNet and VGGNet has more layers using bigger filters. However, I could not implement them because my training images have to be reduced to (50, 50, 3) in order to save space and time. This means I can hardly use other more acurate but complicated CNN models.

To summarize, problem 3 4 and has to do with my own algorithm, whereas problems 1 2 has to do with my implementation and usage of training sets.

Main challenges

The main challenges I faced are face detection and training my CNN, Initially, I used the Haar Cascade classifier for face detection. However, it fails to identify most of the angled faces. After reading that facial alignment improves

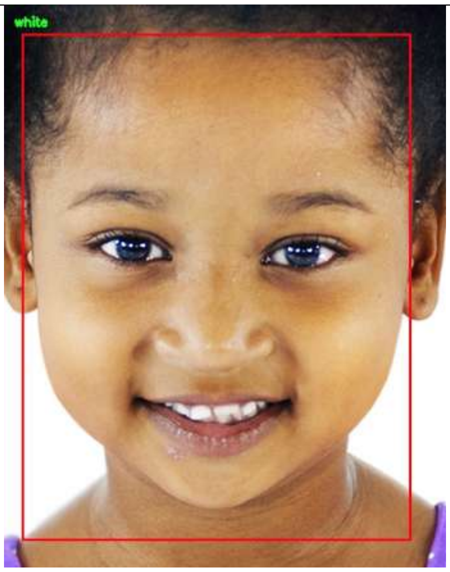


Figure 9: MBlack misclassified as White due to under representation

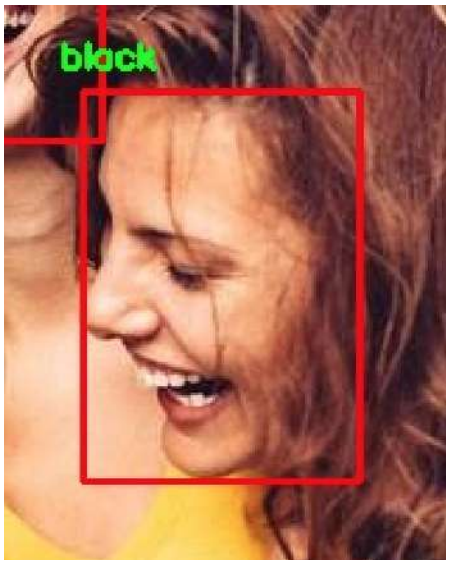


Figure 10: Side face misclassified due to lack of training variations

detection accuracy, I implemented it, but the result is still not as good. I solved this problem by using a deep learning classifier for face detection instead. The other challenge I face is debugging for my CNN. I first implemented the CNN using 20-by-20 images and kept getting receiving errors regarding my neural net. I later discovered that my image is too small to be passed down a 9-layered CNN. I then increased my image sizes to 50-by-50, and changed all the filter sizes in the CNN to size 3 only. Lastly, I first implemented my CNN using grayscaled images only. Because the accuracy

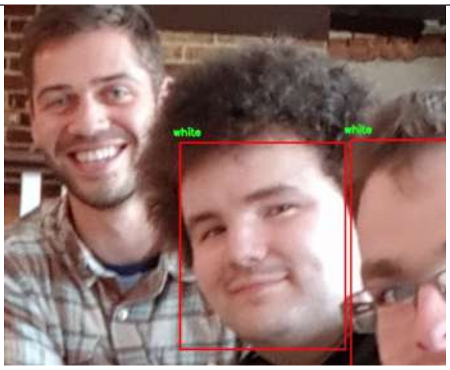


Figure 11: Left face is undetected because it is too small

was much lower than 75%, I decided to implement CNN using coloured images. However this poses a problem because the images now have three channels and is hard to reshape. I overcame this problem by finding out another function in scipy called **imresize** that resizes 3D images. Another problem the Batch Normalization layer that I thought would increase my accuracy but instead lowered it. This problem is mentioned in detail under Reuslts and Discussions.

Conclusion and Future Work

In conclusion, my project used affine transformation for facial alignment, face detection and a CNN using filters for ethnicity recognition. My partner on the other hand impleneted the SVM method for recogniton. And in general, when given a photo with a group of people, my program is able to detect all the faces present in the photo, draw a bounding box around each detected face, adjust the facial landmarks, then determine and label the ethnicity of each person. In the future, I would use a bigger and more varied dataset with equal race distributions to eliminate bias. I would also use a better computer to train other CNN's with bigger sized images so tha more details can be captured by the network.

References

Brownlee, J. Save and load your keras deep learning models. https://machinelearningmastery.com/save-load-keras-deep-learning-models/. Accessed: 2018-12-1.

Coskun, M., Ucar, A., Yildirim, O., and Demir, Y. (2017). Face recognition based on convolutional neural network. *2017 International Conference on Modern Electrical and Energy Systems (MEES)*, pages 376–379.

Masood, S., Gupta2, S., Wajid, A., and Gupta, S. (2018). Prediction of human ethnicity from facial images using neural networks. *Advances in Intelligent Systems and Computing*, (June):217–226.

Rosebrock, A. Face alignment with opencv and python. https://www.pyimagesearch.com/2017/05/22/face-alignment-with-opencv-and-python/. Accessed: 2018-11-29.

Rosebrock, A. Face detection with opencv and deep learning. https://www.pyimagesearch.com/2018/02/26/face-detection-with-opencv-and-deep-learning/post_downloads. Accessed: 2018-11-28.

Rosebrock, A. Facial landmarks with dlib, opencv, and python. https://www.pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/. Accessed: 2018-11-28.

Zhang, Zhifei, S. Y. and Qi, H. (2017). Age progression/regression by conditional adversarial autoencoder. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE.