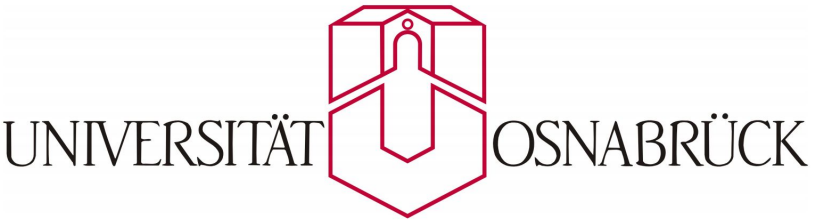

Using Convolutional Neural Network for Development of an Eye-Tracking Application, Capable of Running on the Mobile Devices

by

Sahar Niknam

Submitted to the University of Osnabrück

in Partial Fulfillment of the Requirements for the Degree

of

Master of Science in Cognitive Science

First Supervisor: Peter König, Prof. Dr.

Second Supervisor: Felix Weber, M.Sc.

September 30th, 2020

Abstract

Acknowledgement

Table of Contents

List of Figures

1. Introduction

2. "Eye Tracking for Everyone"

Khosla et al. (2016) tried to realize a software-based eye tracking method, using the ever-growing potentials of neural networks. The authors of the paper, "Eye Tracking for Everyone," (Khosla, 2016) calims that their eye tracking model, the iTracker, which they trained on a large-scale specialized dataset for eye tracking, has an equally satisfactory performance as the current hardware-based devices. The Author gathered and augmented the dataset especially for their experiment. However, they also tested their model on other available datasets and achieved state-of-the-art results. And considering the fact that all these devices work with sensitive equipment, and therefore their usage is costly and usually needs expert's attendance, one can consider the development of a software eye tracker with an equal performance, as a valuable achievement. Especially that the software, according to the authors, is capable of running on a regular smartphone, and in real time.

2.1. GazeCapture

For an efficient training of a neural network, the high variability of the dataset is essential. To meet this criterion, Khosla et al. decided to build their own dataset instead of using the available ones in the field which are all small, and not sufficiently diverse. One of the undesired, but inherent, characteristics of the lab-produced eye tracking datasets is uniformity. For collecting the data, researchers invite the subjects to their labs for the measurements and recording. Thus, the same environment with the same solid background, the same lightning conditions, and roughly the same head poses is coded in the entire dataset. In addition, bringing subjects to a lab, that is, the necessity of the physical presence in a specific place, will always reduce the diversity of the population, not to mention the size of it. To overcome these issues, the authors decided to collect their dataset using crowdsourcing methods. By doing so, they could recruit individuals across the globe, from a widely diverse population. And since the subjects collaborated in building the dataset from their own residence, the diversity of the

backgrounds, lightning conditions, and the head poses is tremendous (e.g. some subjects recorded themselves while lying in their beds).

Achieving this goal, the authors wrote an iOS application, called *GazeCapture* [*], and recruit subjects through the Amazon Mechanical Turk [*] which is a global crowdsourcing platform. The application displays black dots with pulsing red circles around (to draw the subject attention) at random locations on the device screen for 2 seconds, and takes pictures of the subject, using the front-facing camera, starting from 0.5 second after appearance of the dots. Furthermore, the authors instructed the subjects to change the orientation of the device, every 60 dots, to add to the variability of the dataset even more. They also encouraged the subjects to move their head during the time they are looking at one single dot.

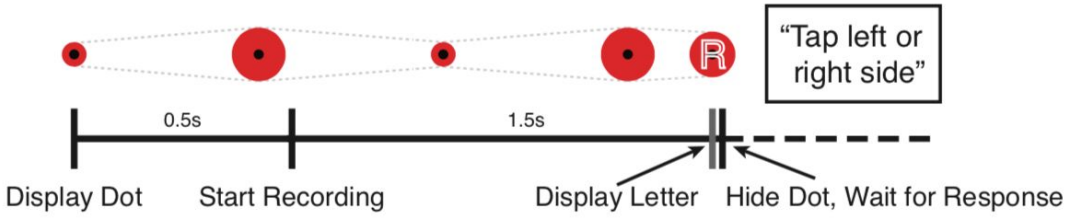

Figure *. "The timeline of the display of an individual dot. Dotted gray lines indicate how the dot changes size over time to keep attention" (Khosla et al., 2016).

The downside of remote crowdsourcing is the questionable reliability of the collected data in the absence of a supervisor. Regarding the specific task of this project, two main issues are imaginable: first, the subject not looking at the dot, and secondly, the subject's eyes not being in the captured frame. To solve the first problem, the authors exploited the real time, built-in face detector of the iOS to make sure the camera captured the face mostly. For the second problem, they displayed one of the two letters *L* and *R* to the pulsing dots for about 0.05 second, shortly before they disappeared. Based on the displayed letter, subjects had to tap on either the left or the right side of their devices' screen. And failing to do so, the subjects needed to repeat that dot recording.

2.2. Dataset

2.2.1. Images

The GazeCapture dataset consists of 1474 folders, and according to Khosla et al., each contains recorded frames of one individual. The dataset includes 2,445,504 frames, both in portrait and landscape orientations, with the dimensions of 640 × 480 pixels with 3 RGB color channels. That makes an average of 1659 frames per subject, with a range varying from 4 to 3590 frames in a folder. Every folder also contains 9 files of metadata on the recorded frames in *json* format.

Figure *. Sample frames from the GazeCapture dataset (Khosla et al., 2016).

2.2.2. Metadata

The metadata files include: *appleFace.json*, *appleLeftEye.json*, *appleRightEye.json*, *dotInfo.json*, *faceGrid.json*, *frames.json*, *info.json*, *motion.json*, *screen.json*.

The appleFace.json, appleLeftEye.json, and appleRightEye.json are dictionaries with 5 keys: *X*, *Y*, *W*, *H*, and *IsValid*; the values are lists of the X and Y coordinates of the top left corner, and the width and height of a square cut of the image that captures the face, left eye, and the right eye for each frame. The X and Y coordinates of the face are with respect to the top left corner of the frame, and the X and Y coordinates of the left and the right eyes are with respect to the top left corner of the face crop. The authors extracted these data using the iOS built-in eye and face detectors. The *IsValid* key takes either 1 or 0 as value, indicating whether the frame properly captured the face, left eye, or the right eye or not, respectively.

The faceGrid.json is also a dictionary with the same keys giving the lists of X and Y coordinates of the top left corner, and the width and the height of the face box within a canvas, representing the rescaled version of frames to a 25 × 25 square. The *IsValid* value is either 1, that is the face crop lies mostly within the frame, or 0 meaning that it does not.

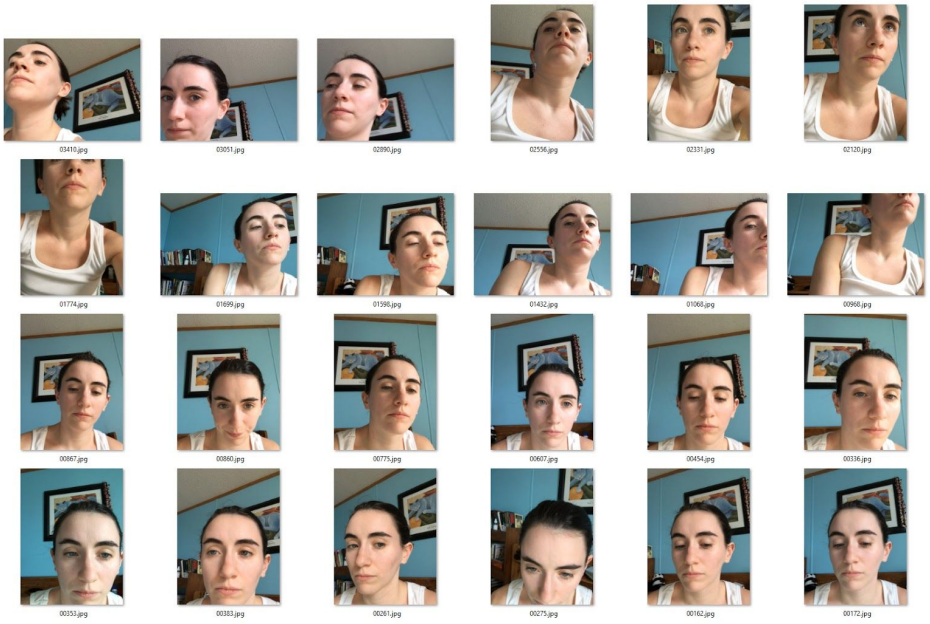

Figure *. Sample frames from one subject.

The dotInfo.json is another dictionary with 6 keys: *DotNum*, *XPts*, *YPts*, *XCam*, *YCam*, and *Time*. According to the authors, the DotNum value is the counting label of the displayed dot for each frame, starting with 0. For example, if the first 14 values in the DotNum list are 0, and the next 15 ones are 1, that means in the first 14 frames, the person was looking at one dot, and in the next 15 frames they were looking at another dot with a different location. And the XPts and YPts are the location of the center of the dots, in points [*], with respect to the top left corner of the screen. While XCam and YCam are supposedly giving the same information, except in centimeters and with respect to the center of the front facing camera of the device. The Time is a list of time spans in second, between the display of a dot and the capture of the corresponding frame.

The frames.json listed the names of all the frames in the folder.

The motion.json is another list of the device motion data recorded at 60 Hz frequency. Each element of the list is a dictionary with 10 keys: *GravityX*, *UserAcceleration*,

AttitudeRotationMatrix, *AttitudePitch*, *AttitudeQuaternion*, *AttitudeRoll*, *RotationRate*, *AttitudeYaw*, *DotNum*, *Time*. The first 8 keys report the measurements of the attitude, rotation rate, and acceleration of the device, retrieved using Apple's *CMDeviceMotion* class. The DotNum is the same value as in the dotInfo.json file. And the Time is the temporal distance between the first appearance of that dot and the exact time of recording the motion parameters.

The info.json, a small dictionary, gives 5 pieces of information: the number of *TotalFrames* in the folder, the *NumFaceDetections* and the *NumEyeyDetections* that indicate how many frames captured the face and the eyes adequately, the *Dataset* that says the subject belongs to which one the train, validation, and test sets that the authors used to train their neural network for reporting the results in their paper, and finally the *DeviceName* that includes 8 models of the Apple's smartphones (iPhone) summing up to 85% of the dataset and 7 models of the Apple's tablets (iPad) that makes the remaining 15% of the GazeCapture dataset.

The screen.json is another dictionary with 3 keys: *H*, *W*, and *Orientation*. The H and W values recorded the height and width of the device screen. The *Orientation* that takes 4 values: 1 for the portrait position with camera on top of the screen, 2 for the portrait position with camera below the screen, 3 for the landscape position with the camera on the left, and 4 for the landscape position with the camera on the right side of the screen.

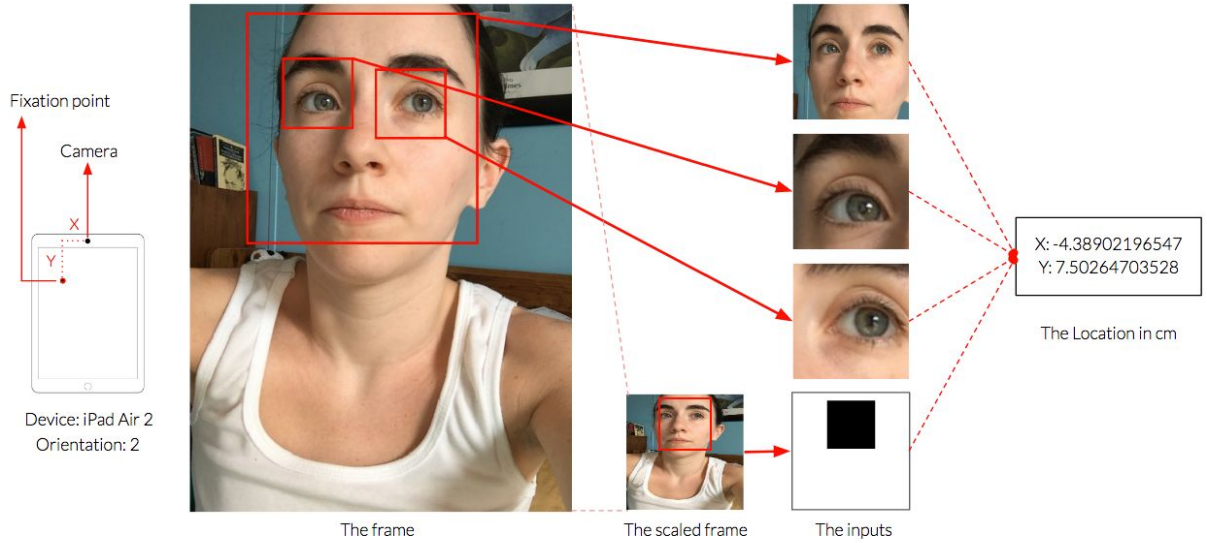

Figure *. A sample frame metadata.

2.2.3. Dataset Analysis

Khosla et al. has created a precious large-scale dataset with high variability and incomparable with the previously generated eye tracking datasets. However, a closer inspection of the GazeCapture dataset reveals a few issues and some inconsistencies with the dataset description in the paper and the iTracker Github repository [*] that could question the reliability of the dataset for eye tracking training.

Inaccurate count of the subjects. The authors reported that they have collected the GazeCapture dataset from 1474 subjects, each of which has a folder containing their recorded frames in the dataset. While it may not be a significant difference, a review of about 5,000 of the folders revealed that, at least, 50 pairs of the folders share the images of the same individuals (e.g. folder pairs of *00533-00534* and *01421-01423*). Even though the frames in different folders are not identical, the number of subjects falls short of 1474.

Numerous invalid frames. The authors reported the size of the GazeCapture dataset to be as large as almost 2.5 million frames. But the number of frames, they tagged as invalid, is as high as 954,545. That makes about 40% of the dataset unusable. While even cutting the GazeCapture dataset in half would leave a dataset 6 times as large as the next largest eye tracking dataset, but the difference between the number of the valid and the invalid frames is still significant and the author failed to address the actual size of the dataset properly in the paper, especially when comparing it with the other dataset in the table 1 [*].

Valid frames that are invalid. Though Khosla et al. did not give a detailed description of the GazeCapture image recording, delving into the GazeCapture metadata, we can find out that they recorded about 14 images of a subject while the person was fixating one single location. According to the authors, the duration of the fixation for each single location was 2 seconds and the recording started 0.5 second after the display of the dot. That means they captured about 14 images during a period of 1.5 second, which is a relatively narrow window. And considering the *Tap left*/*Tap right* arrangement in the last 0.05 second of this window, one may expect perfect capture of subjects attention. However, reviewing about 1% of the GazeCapture dataset proved otherwise. Many frames in the dataset capture subjects with closed eyes or while they are looking away from the screen, nonetheless these frames are tagged as valid and have coordinates

as corresponding labels. Apart from closed-eyes and looked-away frames, the chances of iOS detecting eyes' locations erroneously are not eliminated.

Figure *. Samples of closed-eyes and looked-away frames in the GazeCapture dataset.

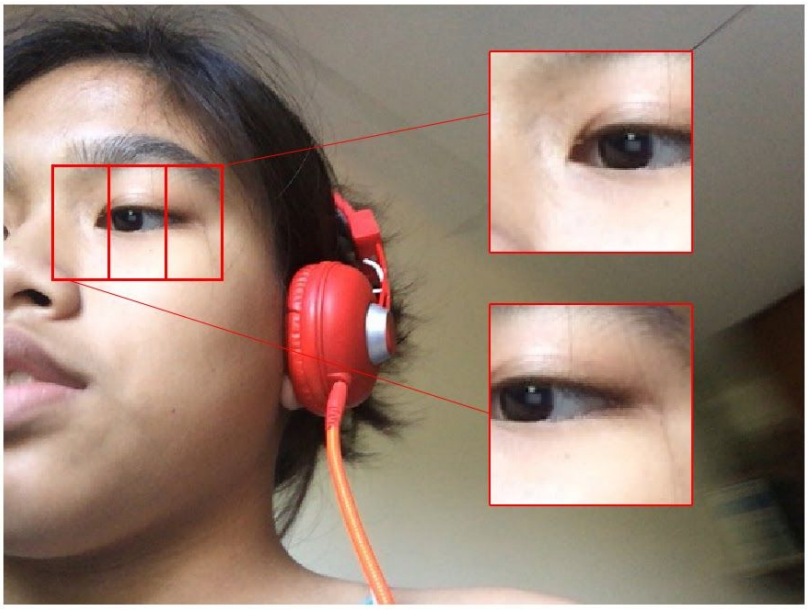

Figure *. iOS detected the left eye for both left and right eye crops and tagged the frame *valid*.

Fixation location. Unfortunately, Khosla et al. are not clear about the locations, the number, and the distribution of the displayed dots for the subjects. However, they have mentioned that not all the subjects have frames captured of them looking at the full set of the dots. But even the subjects with a full set of fixation locations do not have each location in 4 possible orientations.

The fixation locations are the labels in the GazeCapture dataset. The metadata gives the labels data in two formats: points (XPts, YPts) [*] with the origin being the top left corner of

the screen, and also in centimeters (XCam, YCam) with the origin being the center of the front facing camera. The two origins have a constant distance on one single device, that means the location in points and the location in centimeters, regardless of the orientation of the device, should be a one-to-one or a reversible transformation. Based on the definition of the point unit, the only exception to this rule happens when the Display Zoom of the device changes (and the screen size in the screen.json file stored the information about the Display Zoom). Reviewing the authors' codes for converting (XPts, YPts) to (XCam, YCam) and vice versa [*, *, *] agree with these assumptions. However, inspecting the dotInfo.json files in metadata opens up that only 98 subjects, out of 1474, have the same number of unique coordinates in points and centimeters. While the majority of the subjects in the GazeCapture dataset either have the 76 unique coordinates in points and 90 unique coordinates in centimeters, or 93 and 120 unique coordinates in points and centimeters, respectively. That is, one single location in centimeters has two different coordinates in points when the orientations change from 1 to 2, or from 3 to 4, which based on [*] should not be the case. For example, frames *00742.jpg* and *01763.jpg* in the folder *02156* have the same screen size and the same fixation location, while they have two different, and not even symmetric, coordinations in centimeters, apparently only because they differ in orientation (their orientations are 4 and 3, respectively).

2.3. iTracker

Khosla et al. wrote their model, iTracker, in Caffe [*] that is a popular language in computer vision, for coding convolutional neural networks. But in the paper's Github repository, the codes are also available in Pytorch [*].

As a side note, the iTracker model has been never, commercially or experimentally, developed as an application for using on mobile devices; except as a part of one of the authors' doctoral dissertation [*].

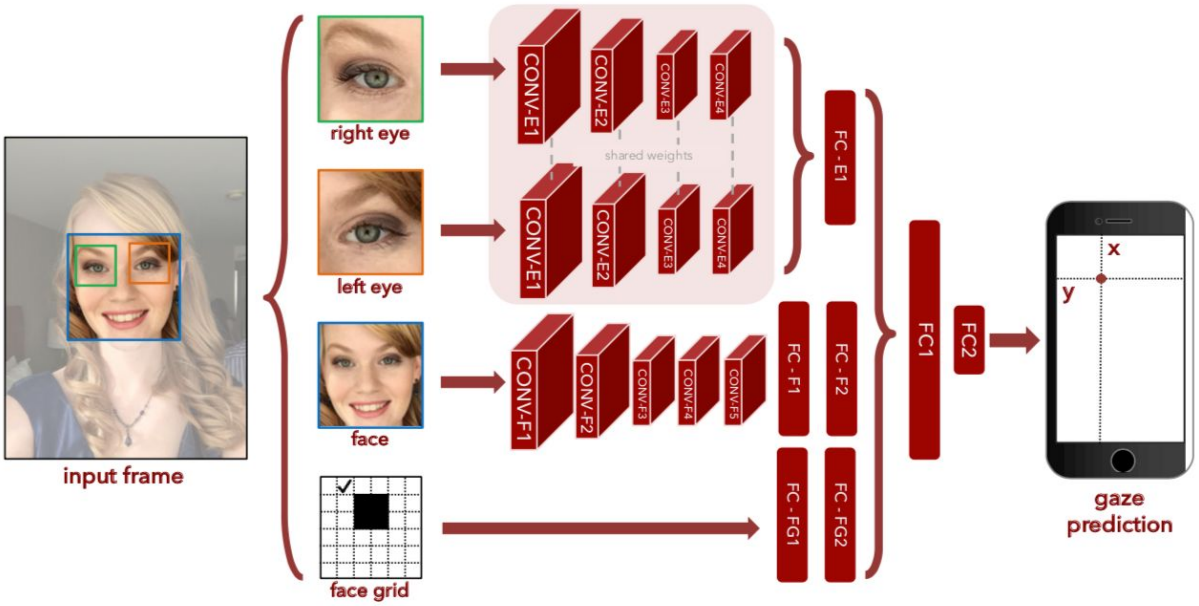

Figure *. The architecture of the iTracker Model (Khosla et al. 2016)

2.3.1. Architecture

The iTracker model has a complex, parallel architecture. The model takes four inputs: the left eye crop, the right eye crop, the face crop, and the face grid. The eyes and the face crops are cut out of the original frame based on the appleLeftEye.json, the appleRightEye.json, and the appleFace.json metadata, and then rescaled to a square patch of dimensions 224 x 224 pixels. The face grid crop is a 25 x 25 pixels square patch, created with the faceGrid.json metadata. Each input is fed to a small network, which its output is concatenated with the others', on different levels.

The left and the right eye crops are fed into two identical CNNs. The eyes' CNN has 4 convolutional units with 96, 256, 384, and 64 kernels, and 11, 5, 3, and 1 kernel sizes, respectively. All the strides sizes are 1, except for the first unit in which the kernels jump over 4 pixels in every move. Following the first two convolutional layers first comes a max pooling layer with a pool size of 3 and stride size of 2, and then a *local response normalization* layer. All the activation functions are ReLU. Then the output of the eyes' last convolutional layer is flattened, concatenated, and fed into a single dense layer with 128 units and ReLU activation function. The output of the dense layer is the output of the process of both eye crops.

The face crop is fed into an identical CNN with the eyes' CNNs. Except that after flattening the output of the final convolutional layer, the output is fed into two dense layers, sequentially; first one with 128 and the next one with 64 units, and both with ReLU activation function. The output of the second dense layer is the face process output.

The face grid is simply flattened and then fed into two sequential dense layers, with 256 and 128 units, both with ReLU activation function. The output of the second dense layer is the face grid process output.

And finally, the eyes, the face, and the grid outputs are concatenated to a 1 × 320 vector and fed into a dense layer with 128 units and ReLU activation function. The final and output layer is another dense layer only with 2 units (for the coordinates regression) with ReLU as its activation function.

Table *. The summary of the iTracker model.

Layer (type)	Output Shape	Param #	Connected to
input_6 (InputLayer)	[(None, 224, 224, 3)	0	
conv2d_8 (Conv2D)	(None, 54, 54, 96)	34944	input_6[0][0]
max_pooling2d_4 (MaxPooling2D)	(None, 26, 26, 96)	0	conv2d_8[0][0]
lr_n2d_4 (LRN2D)	(None, 26, 26, 96)	0	max_pooling2d_4[0][0]
conv2d_9 (Conv2D)	(None, 26, 26, 256)	614656	lr_n2d_4[0][0]
max_pooling2d_5 (MaxPooling2D)	(None, 12, 12, 256)	0	conv2d_9[0][0]
lr_n2d_5 (LRN2D)	(None, 12, 12, 256)	0	max_pooling2d_5[0][0]
conv2d_10 (Conv2D)	(None, 12, 12, 384)	885120	lr_n2d_5[0][0]
input_8 (InputLayer)	[(None, 224, 224, 3)	0	
input_9 (InputLayer)	[(None, 224, 224, 3)	0	
conv2d_11 (Conv2D)	(None, 12, 12, 64)	24640	conv2d_10[0][0]
input_7 (InputLayer)	[(None, 25, 25)]	0	
model (Model)	(None, 9216)	1559360	input_8[0][0] input_9[0][0]
flatten_3 (Flatten)	(None, 9216)	0	conv2d_11[0][0]
flatten_4 (Flatten)	(None, 652)	0	input_7[0][0]
concatenate_2 (Concatenate)	(None, 18432)	0	model[3][0] model[4][0]
dense_7 (Dense)	(None, 128)	1179776	flatten_3[0][0]
dense_9 (Dense)	(None, 256)	160256	flatten_4[0][0]

dense_11 (Dense)	(None, 128)	2359424	concatenate_2[0][0]
dense_8 (Dense)	(None, 64)	8256	dense_7[0][0]
dense_10 (Dense)	(None, 128)	32896	dense_9[0][0]
concatenate_3 (Concatenate)	(None, 320)	0	dense_11[0][0] dense_8[0][0] dense_10[0][0]
dense_12 (Dense)	(None, 128)	41088	concatenate_3[0][0]
dense_13 (Dense)	(None, 2)	258	dense_12[0][0]
Total params: 6,900,674 Trainable params: 6,900,674 Non-trainable params: 0			

2.3.2. Training

Khosla et al. divided the GazeCapture dataset into three subsets of train, validation, and test splits. Instead of randomly distributing the frames, they picked 1271, 50, and 150 subjects for the train, the validation, and the test subsets, respectively. Studying the metadata shows that their train, validation, and test data subsets contain 1,251,983, 59,480, and 179,496 frames. They made sure that all the subjects in the validation and the test sets have frames for the full set of fixation locations. They also augmented both train and test datasets, 25-fold and by shifting the eyes and the face and the face grid accordingly, during the training that led to their best reported results in the paper.

In the paper "Eye Tracking for Everyone" (Khosla et al., 2016), the authors did not provide the reader with a detailed outline of the training. And available information in the GazeCapture Github repository [*] conflicts the paper's, which could be due to the different rounds of training. In the paper, the authors mentioned that they have achieved their best network

3. Improvements

3.1. Dataset

3.2. Model

3.2.1. Architecture

3.2.2. Training

3.3. Experiment 1: Grayscale Images

In this scenario, each frame was converted to a grayscale image. The motivation behind this scenario was to reduce the computational workload, since dropping the color channels will downsize the dataset to one third. The conversion was done using a method of the *OpenCV* Python library: `cv2.COLOR_BGR2GRAY`.

3.4. Experiment 2: Eyes' Networks

3.5. Experiment 3: MTCNN

Multi-Task Cascaded Convolutional Network (MTCNN) is a lightweight CNN for face detection and alignment in real time which is claimed to have superior accuracy over the state-of-the-arts methods and algorithms. The core idea of MTCNN is to run a CNN on a cascade of scaled versions of the original image, and it was introduced in a paper by Zhang, Zhang, Li, and Qiao (2016). In 2017, the Github user, *ipacz* [*], wrote a Python (pip) library based on the MTCNN idea which is now available to install and to use for free.

MTCNN takes an image and returns a list of detected faces in the image, each of which is a dictionary with 3 keys: *box*, *confidence*, and *keypoints*. The *box* key's value is a list of four values of the X and Y coordinates of the top left corner, and the width and the height of the face box (in pixels). The *confidence* is the reliability of the detection (in percent). And the *keypoints* is a nested dictionary of 5 facial landmarks coordinates, including the approximate center of the left and the right eyes, the nasal tip, and the left and the right corners of the mouth.

```
1 !pip install mtcnn
2 from mtcnn.mtcnn import MTCNN
3 faces = MTCNN().detect_faces(img)
4 faces
```

```
>>> [{'box': [90, -39, 277, 381],
  'confidence': 0.9999997615814209,
  'keypoints': {'left_eye': (155, 111),
   'mouth_left': (166, 259),
   'mouth_right': (263, 264),
   'nose': (207, 186),
   'right eye': (285, 118)}}]
```

Listing *. Install, import, and running MTCNN

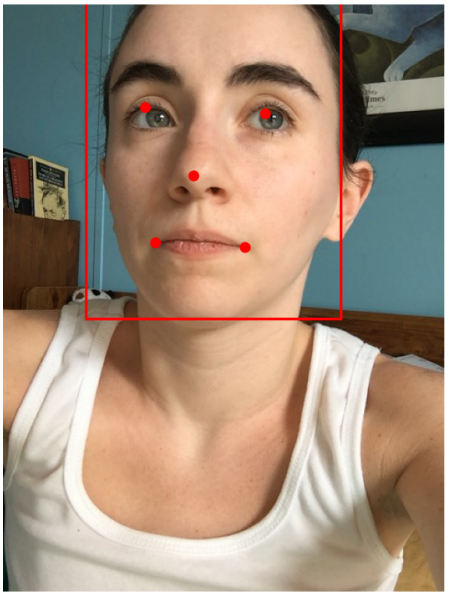

Figure *. MTCNN results are plotted with a red box and red dots.

In MTCNN experiment ideand 5 facial detection which was released in 2017. Unlike the regular CNNs, an MTCNN recognizes face in an image

3.5.1. Grid

3.5.2. Vector

3.6. Experiment 4: Nose crop

4. Results

5. Conclusion

Bibliography

Zhang, K., Zhang, Z., Li, Z., & Qiao, Y. (2016). Joint face detection and alignment using multitask cascaded convolutional networks. IEEE Signal Processing Letters, 23(10), 1499-1503.

Xu, P., Ehinger, K. A., Zhang, Y., Finkelstein, A., Kulkarni, S. R., & Xiao, J. (2015). Turkergaze: Crowdsourcing saliency with webcam based eye tracking. arXiv preprint arXiv:1504.06755.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093, 2014.

Kannan, H. D. (2017). Eye tracking for the iPhone using deep learning (Doctoral dissertation, Massachusetts Institute of Technology).

https://www.mturk.com/

https://apps.apple.com/us/app/gazecapture/id1025021075

https://github.com/CSAILVision/GazeCapture

https://www.paintcodeapp.com/news/ultimate-guide-to-iphone-resolutions

https://github.com/CSAILVision/GazeCapture/blob/master/code/screen2cam.m

https://github.com/CSAILVision/GazeCapture/blob/master/code/cm2pts.m

https://github.com/CSAILVision/GazeCapture/blob/master/code/pts2cm.m