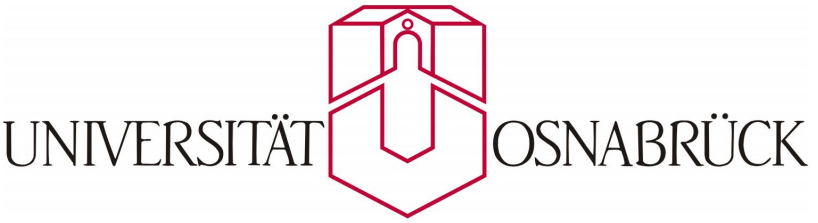


Using Convolutional Neural Network for Development of an Eye-Tracking Application, Capable of Running on the Mobile Devices

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

Chapter 1

Introduction

Chapter 2

"Eye Tracking for Everyone"

Khosla et al. [*] tried to realize a software-based gaze tracking method, using the ever-growing potentials of neural networks. The authors of the paper, "Eye Tracking for Everyone," [*] claim that their gaze tracking model, the iTracker, which they trained on a specialized dataset, has an equally satisfactory performance as the current methods and devices. Additionally, they tested their trained model on other available datasets and achieved state-of-the-art results. Therefore, considering the fact that all the commonly-used gaze tracker devices work with sensitive equipment, and consequently their usage is costly and usually needs expert's attendance, one can consider the development of an entirely software-based gaze tracker with a competing 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.

To this end, the authors gathered and augmented a large scale dataset for their experiment. Khosla et al. especially designed the dataset, known as the GazeCapture dataset, for the purpose of gaze tracking, rather than simply tracking the iris or the pupil as in an eye tracking task.

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 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.

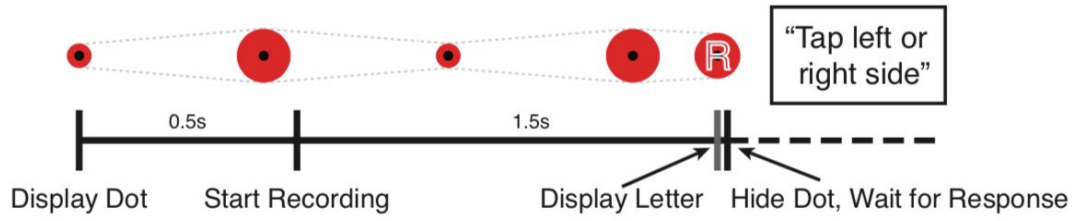


Figure *. "The timeline of the display of an individual dot. Dotted gray lines indicate how the dot changes size over time to keep attention" (Khosla et al., 2016).

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

The GazeCapture dataset consists of 1474 folders, each of which, according to Khosla et al., contains recorded frames of one individual. The dataset includes 2,445,504 frames in total. That makes an average of 1659 frames per subject, with a range varying from 4 to 3590 frames in a folder. Each frame is a color image with 3 RGB color channels, in *jpg* format, and with the dimensions of 640 × 480 pixels, 96 dpi, and a bit depth of 24. The average size of a frame is nearly 50 KB, which led to a the total size of 144 GB for the GazeCapture dataset. Every folder also contains 9 files of metadata on the recorded frames in *json* format, including either a dictionary or a list readable in Python.

2.2.1. Images

The images in the GazeCapture dataset are closed-up frames of individuals, divers in race and almost divers in (legal working) age, looking at a fixation point on a digital screen in front of them. The frames are captured under a wide range of different environments, backgrounds, lightning conditions, and head poses. The images all have the same dimensions (640, 480) but they are in two, portrait and landscape, orientations.

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.

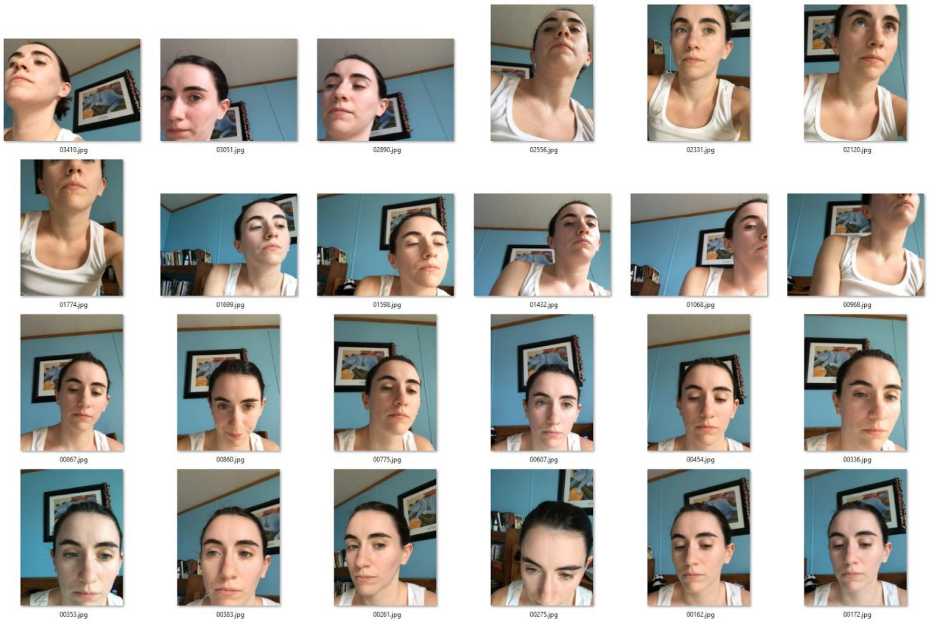


Figure *. Sample frames from one subject.

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.

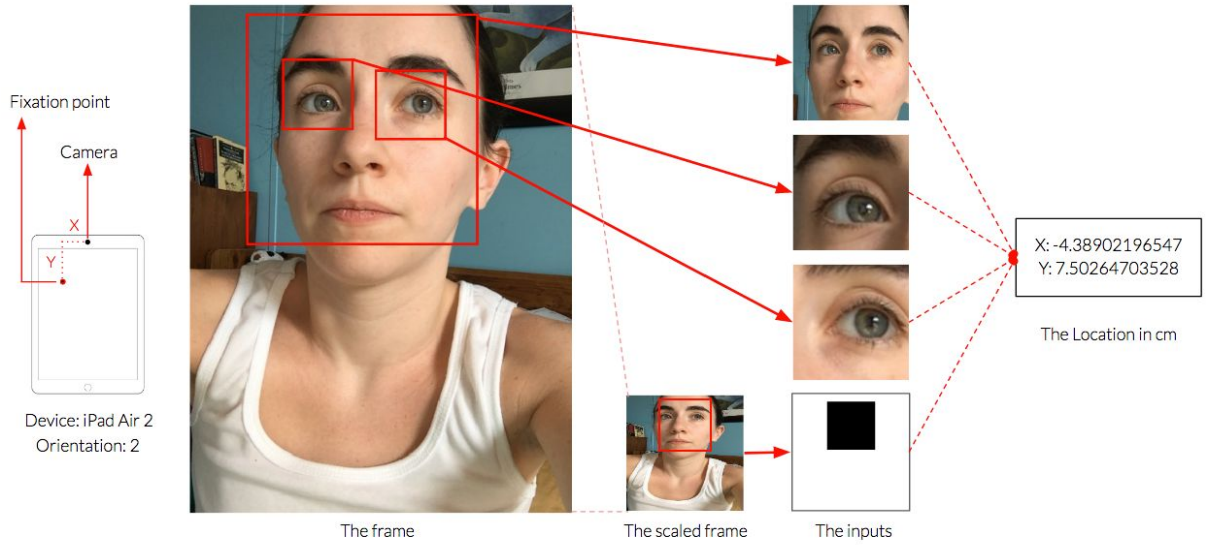


Figure *. A sample frame metadata.

2.2.3. Dataset Analysis

Khosla et al. has created a precious large-scale dataset with high variability and incomparable with the previously generated gaze 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 gaze 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 gaze 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 the closed-eyes and looked-away frames, the chance of iOS detecting eyes' locations erroneously, even though insignificant, is not eliminated.

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

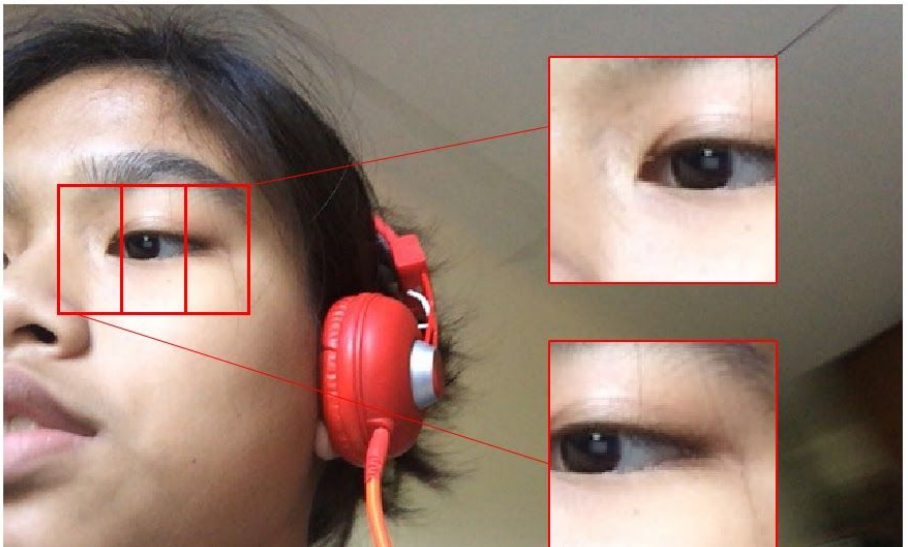


Figure *. In the folder 02967, for the frame 02728, iOS detected the left eye for both left and right eye crops and tagged the frame *valid*.

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, so far, neither a natural nor a legal person developed the iTracker model, in either of experimental or commercial capacity, into an application for mobile devices for using in massive scale; the only exception is one of the authors of the "Eye Tracking for Everyone" paper, Kannan, H. D., who built an application based on the model, for limited and experimental usage, as a part of her doctoral dissertation [*].

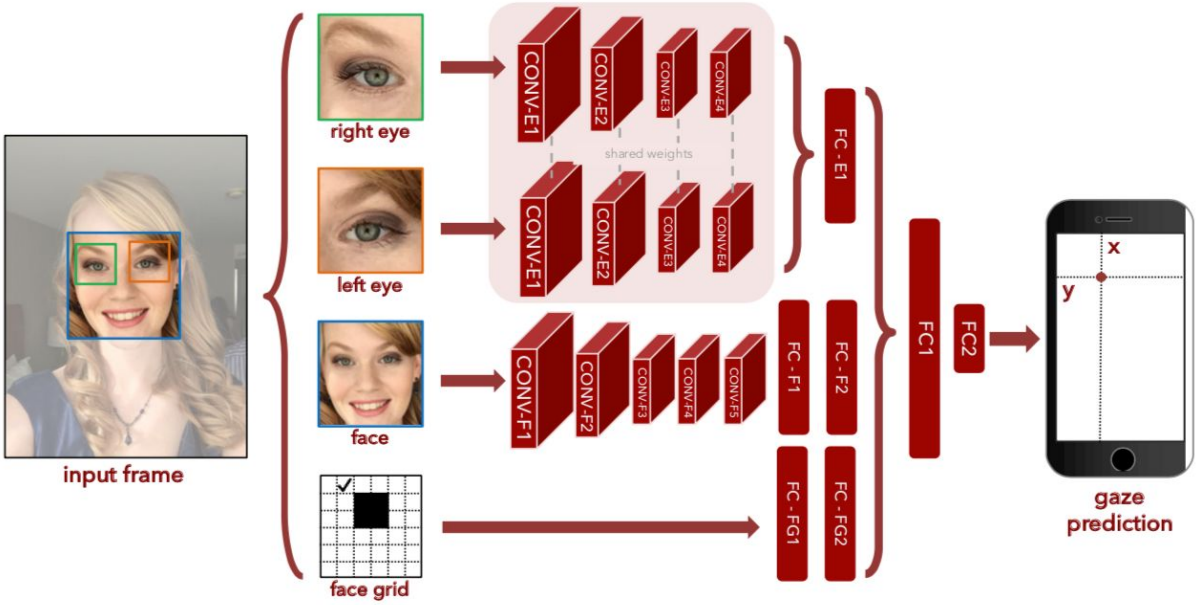


Figure *. The architecture of the iTracker Model (Khosla et al. 2016)

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 subnet whose output is concatenated with the others', on different levels.

The left and the right eye crops are fed into two identical CNNs that share the weights. In practice, that means feeding both eyes into one single CNN, but separately. The eyes' CNN has 4 convolutional units with 96, 256, 384, and 64 filters, 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. But Then the output of the eyes' last convolutional layer is flattened, concatenated, and fed into a single dense layer with 128 neurons 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 neurons, 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 neurons, 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 x 320 vector and fed into a dense layer with 128 neurons and ReLU activation function. The final and output layer is another dense layer only with 2 neurons (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

Dataset Splits. 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 1251983, 59480, and 179496 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.

Parameters. In the paper "Eye Tracking for Everyone" [*], 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 results by training the network "for 150,000 iteration with a batch size of 256" [*]. Although, they have never discussed the number of epochs. In the neural networks literature, the end of an *iteration* during the training is punctuated with the update of the trainable parameters of the model (usually, the weights), that is, after training the network over one full batch. On the other hand, training the network over each and all data samples for one round, is called an epoch [*]. Therefore, knowing the data size, one can calculate the number of iterations (per epoch) by having the batch size, and vice versa. On this account, while considering that the neural networks do not have a set-in-stone terminology, it seems that by *iteration*, the authors meant *epoch*, which is a common mistake. With this assumption, the authors trained their best model with a learning rate of size 0.001 for 75,000 epochs and then continued training it for 75,000 more epochs with a learning rate of 0.0001. They also used a momentum of size 0.9 and weight decay rate of size 0.0005 over all the 150,000 epochs of training. Khosla et al. did not discuss the optimizer that they have used for training, but according to their published codes [*], they applied stochastic gradient descent (SGD).

Accuracy Metrics. The authors reported the iTracker's error as the average over all the Euclidean distances between the fixation points and the model predictions in centimeters.

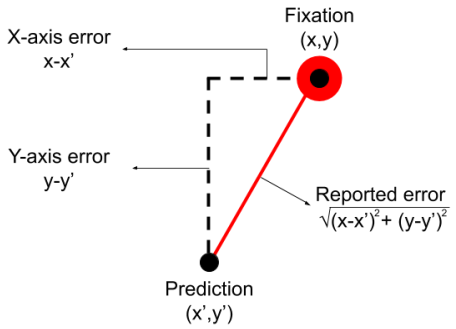


Figure *. The authors calculated the error as the Euclidean distance between the fixation and the predicted points.

However, they also calculated the average errors for the smartphones and the tablets, separately. According to the authors, the reason for this decision was to cancel out the screen size, and also the user-to-screen distance differences among different devices. In addition, for the best results that the authors have achieved, they also "fine-tuned" [*] the iTracker model for

each device and *each orientation*. But they did not give any details about the quality of these fine tunings. Eventually, they truncated the model predictions based on the screen size. Taking these arrangements into account, they achieved a prediction error of size 1.71 and 2.53 centimeters for smartphones, and tablets, respectively. The author also found out that they can reduce the error size to 1.34 for smartphones and 2.12 for tablets, in case of 13-fixed-point calibration based on the works of Xu et al. [*].

Furthermore, the authors also reported the model *dot error*, which is the average Euclidean distance over all frames with dots at the same location. The advantage of calculating the dot error, according to the arthurs, is that it sheds a more realistic light on the model accuracy, because in practical applications of models like iTracker, the accuracy over *a stream of frames* is in question, not one single frame. The authors reported the lowest dot error of the iTracker model 1.83 and 2.38 for smartphones and tablets, respectively.

Chapter 3

Experiments

This thesis tries to test three ideas toward improving the performance of CNNs in gaze tracking task, each of which falls under one aspect of neural networks in practical applications:

Preprocessing. Does the color information contribute to the purpose of gaze tracking?

Inputs. Does the nose localization improve the iTracker model performance?

Architecture. Do both eyes convey identical information in the gaze tracking context?

3.1. Dataset

For testing the thesis's ideas, we decided to work on a smaller subset of the GazeCapture dataset. Initially, we randomly sampled 10,000 valid-tagged frames, regardless of the device type or the orientation. Then we dropped 9 frames, due to the MTCNN failure in face detection, and one more because of faulty metadata. Therefore, the miniature dataset of the thesis consists of 9,900 frames. Additionally, we adjusted the eyes and face crops sizes by rescaling them from 224 × 224 to 64 × 64 pixels.

The immediate necessity of reducing the size of dataset and decreasing the images size was the shortage of computational resources for processing tera-scale data size with a model with about 7 millions (9.6 millions in the nose crop test) of trainable parameters, for 150,000 epochs. The system that we had at our free disposal was a gaming machine with a GPU of NVIDIA GeForce® GTX 1070 with 8GB GDDR5 VRAM, that is, 1920 CUDA cores. Using Google Colab was another option with a higher GPU power. Nevertheless, the disk space for uploading the data to the cloud was not enough. In addition, even with a Pro Google account, which is not even accessible for non US-based users, the maximum possible runtime on Colab is 24 hours; while replicating the original experiment needs weeks of training.

The miniature dataset is considerably smaller than the GazeDataset, both in number and size. Regardless, that could not affect the reliability of the results of this thesis, as our primary purpose was to investigate the outcome of some adjustments for improving the *performance*, rather than to raise the benchmarks for improving the *outcome*. To that purpose, we also trained the original iTracker model on the miniature dataset to be able to compare the results.

3.2. Model

For the thesis's experiments, we have worked with the same iTracker model as the authors outlined in the paper [*]. But for finer details, which are missing in the paper, our work heavily relies on the Pytorch codes available in the GazeCapture Github repository [*]. However, we coded the model in TensorFlow 2.1.0, and *tf.keras* submodule with version 2.2.4-tf. And for replicating the complex architecture of the model, we used Keras *Model()* class with the *functional* API. The codes is accessible on the thesis's Github repository [*].

3.2.1. Architecture

During the translation from Caffe to Keras, two problems came up: implementing the Pytorch *torch.nn.CrossMapLRN2d* layer [*] in Keras, and adjusting the *groups* argument of the *tf.keras.layers.Conv2D* layer [*].

The CrossMapLRN2d layer is a local response normalization layer that normalizes activation function output in a local neighborhood, over all the feature maps of the previous convolutional layer. This type of normalization is helpful, particularly when using ReLU as the activation function, since ReLU is a linear function and therefore its responses are unbounded. Normalizing ReLU response over a local area keeps the model sensitivity sharp for the most important features captured in the data. The Keras API had an equivalent LRN2D layer for the CrossMapLRN2d, but the developers decided to remove the layer in 2015. However, the LRN2D class's code is still accessible through the remove commit on the Keras repository [*]. Therefore, we employed the same code for implementing an equivalent local normalization layer as a child of the *tf.keras.layers* class.

The argument *groups* in the *tf.keras.layers.Conv2D* layer is an option for making groups out of the input channels / feature maps, and run the filters on each group separately. Basically every filter convoles each and every channel. Thus, one can consider *grouping* as one more step towards feature localization. In both Pytorch and Keras, *groups* is an argument in the convolutional layer, nonetheless, our code threw error after setting the *groups* to 2 (as in the Pytorch code). Our efforts to solve the problem came to a dead end, especially because changing the *groups* value (from its default that is 1) is not a popular practice in Keras. Therefore, we could not find enough resources addressing the issue, except a brief mention to version conflicts. Consequently, we commented the line out.

Other than that, our model is a loyal replication of the iTracker. However, since we feed smaller size crops, the total number of trainable parameters of our model reduced from the original 6,900,674 to 3,460,034 with the miniature dataset.

3.2.2. Training

For training, we made all the decisions, mainly in line with an effort to compensate for the small dataset. For all the experiments, we uploaded the corresponding weights of the reported results to the thesis's Github repository [*].

Dataset Splits. To overcome the small size of the dataset, we decided to exercise the K-fold cross validation method to use up the entire set of samples. Therefore, instead of splitting the dataset into three sets of train, validation, and test, we activate the *validation_split* argument in the *fit* method and set it to 20%. That is, in every epoch, the code trained the model on 1592 samples and held 398 ones out as the test set.

Parameters. Regarding the dataset size, we also trained the network for a proportionally fewer number of epochs, to avoid overfitting. Therefore, we first trained the network for 100 epochs with a batch size of 128, and then 20 more epochs with a batch size of 32. The learning rate was 0.1 for all 120 epochs. We also worked with Adam optimizer with all its default parameters, except for the learning rate. For the loss we implemented the *Mean Squared Error* (MSE).

Accuracy Metrics. For comparing the accuracy of the experiments, we measured the error with the *Mean Absolute Error* (MAE) metric (in centimeter). In the original experiment, Khosla et al used Euclidean distance which works with squared error (L2 distance), whereas MAE measures the absolute difference (L1 distance). We decided to train the model, while observing the L1 distance because doing so, the model would focus on minimizing its misjudgments over X- and Y-axis, independently. And that is an optimal treatment when distributed data along with the X-axis conveys different information, compared to the Y-axis. In addition, as Kannan, H. D., one of the authors of the "Eye Tracking for Everyone" paper [*] discusses in her doctoral dissertation [*], the Euclidean distance metric, because of its squared terms is sensitive to outliers and wastes training efforts on including the extreme cases, instead of fine-tuning the weights on average ones.

In measuring our experiments' prediction errors, unlike in the original experiment, we did not average the reported MAE over devices, orientations, or dots, nor did we fine-tune the model for any of these variables.

3.3. Experiment 1: Grayscale Images

In this experiment, we converted all the color inputs (eyes and face crops) to grayscale images. For the conversion, we used a method of the *OpenCV* library: `cv2.COLOR_BGR2GRAY`. The inputs in this experiment are the same as in the paper: eyes and face patches and face grid; except that the crops are in grayscale and have dimensions of 64 × 64 × 1. The model in this case has 3,413,570 trainable parameters.

The motivation behind this experiment is to reduce the data size and thereby the computational workload. By dropping the color channels, the dataset will shrink to one third of its original size. And even though it is insignificant considering the total number of parameters, feeding crops in grayscale cut off about 47,000 trainable parameters.

3.4. Experiment 2: The Nose

In the nose experiment, either explicitly or implicitly, we try to investigate whether the nose localization can help improve the accuracy of the iTracker model. The idea is based on the fact that, unlike simply tracking the eyes which needs detecting iris/pupil, gaze tracking success heavily depends on the head pose detection. The head pose in turn, geometrically depends on the location of major facial landmarks, including the nose tip [*]. To reflect the nose data and convey its data to the network, we tested three methods: with a vector, with a grid, and with a crop. And to get the nose data we used MTCNN.

3.4.1. MTCNN

Multi-Task Cascaded Convolutional Network (MTCNN) is a lightweight CNN for face detection and alignment in real time which its developers claim 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. Zhang, Zhang, Li, and Qiao introduced this idea for the first time in their paper 'Joint face detection and alignment using multitask cascaded convolutional networks' in 2016 [*]. In 2017, the Github user, *ipacz* [*], wrote a Python (pip) library based on the MTCNN idea which is now available to users 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 returns 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 nose 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 run MTCNN

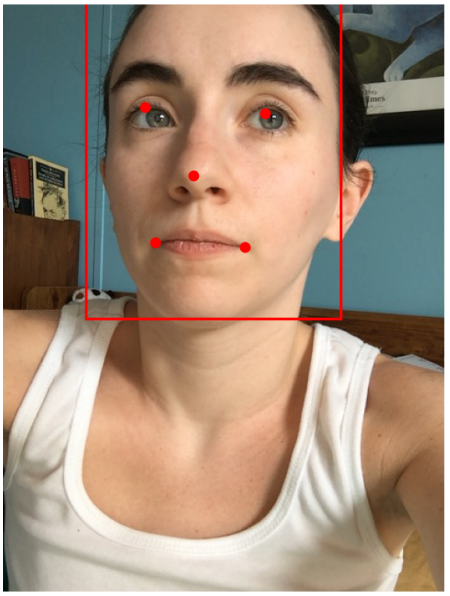
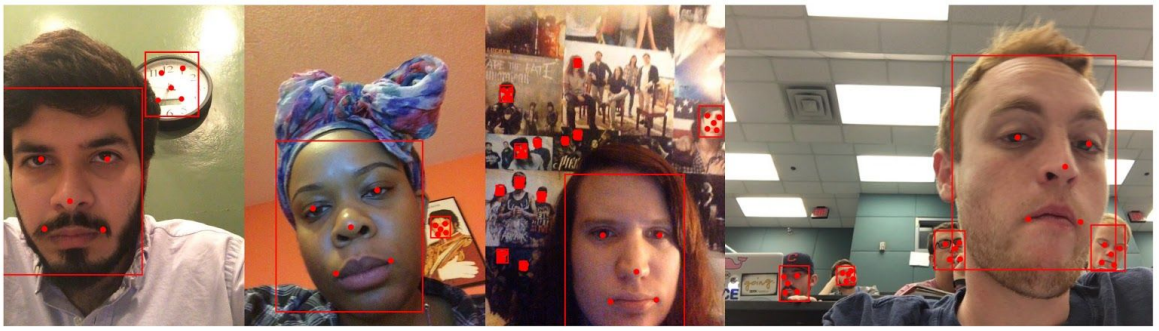


Figure *. MTCNN results are plotted with a red box and red dots.

In this thesis we took advantage of the MTCNN library for two main purposes. First for elaborating the face grid concept, numerically and visually, and secondly and more specifically, for getting the nose tip coordinates to cut out the nose crop. To do so, we ran the MTCNN on the 10,000 frames which we sampled randomly and saved the results as a Python list. However, as noted earlier, MTCNN returns a list of detections. That is, if there are more than one face in the image, MTCNN will also return the facial landmarks of all the detections, not only the face of the subject in the frame. The possible irrelevant detections include people in the background, pictures of people, and in some infrequent cases, objects that resemble a human face (Fig. *a). To locate the subjects faces among an indefinite number of detections, we validly assumed that the subject's face box would be the largest detected box. Based on this assumption, before getting the coordinates of the subject face detection, we iteratively kept comparing the first two elements in the list, removing the detection with the smaller face box width, until only one detection was left. In addition, we also had frames in which MTCNN could not find any face. We spotted them out simply by checking whether a list is empty. Although, it is noteworthy that over three trials, MTCNN had an average of 8 failures out of 10,000 frames, that is barely 0.1%.



(a)

(b)

Figure *. (a) MTCNN recognizes inanimate/irrelevant faces. (b) MTCNN recognizes no face.

3.4.2. Vector

For the *vector* test, we got the MTCNN generated coordinates of both eyes and the nose tip for all frames and created a *numpy* array of float32 values with a shape of [9990, 6]. Then we add an extra subnet to the original iTracker model for processing the vector. The vector subnet consists of two dense layers, the first one with 128 and the output with 64 neurons. Both layers have ReLU as their activation functions. In the final concatenation phase of the model, we concatenated the vector subnet output along with the output of other subnet. The trainable parameters of this model sum up to 3,267,842.

The motivation for running this test was to check if the model could, in addition to the visual inputs, extract an auxiliary pattern out of merely numerical values of the eyes and nose locations and map them to the fixation point coordinates. That is, the vector test stresses the absolute coordinates of the eyes and nose tip without having the face box as a reference.

3.4.3. Grid

In the *grid* test, we replaced the paper's face grid with one made with the MTCNN-derived coordinates. We got the coordinates of the face box, both eyes, and the nose tip from MTCNN, and created a white canvas with the same dimensions as the frame. We drew a black box with MTCNN's face box coordinates, and inside the box, plotted a white triangle whose vertices are the MTCNN's detections for the two eyes and the nose. And as the final step, we scaled down the canvas to 50 x 50 dimensions, that is 4 times larger than the paper's face grids. We decided to keep the face grids up to that size since downsizing them to 25 x 25 pixels would have distorted and debunked the extra, fine details of the eyes-nose coordinates.

The grid test model architecture is the same as the original iTracker. We only changed the content of one input, and because of that the number of trainable parameters of the model raised to 3,940,034.

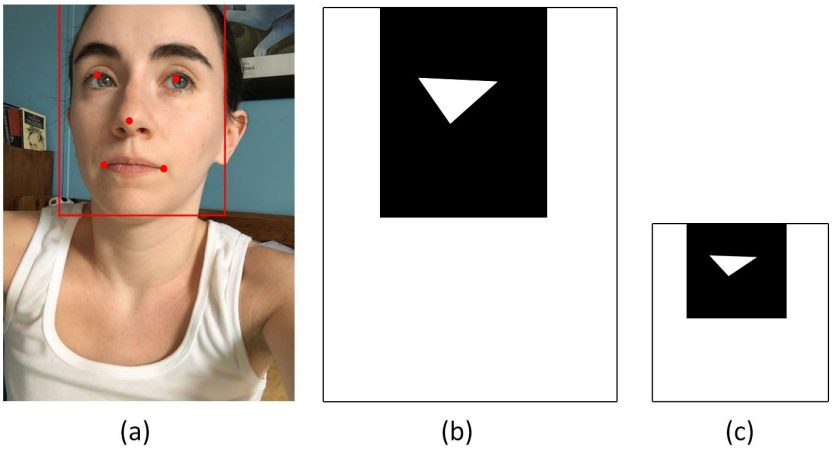


Figure *. (a) The frame augmented with the MTCNN detections in red lines and dots. (b) The grid with the face box and the eyes-nose triangle on a white canvas with the same size as the frame. (c) Rescaled grid to 50 x 50 pixels.

Multiple improvements motivated the grid test. First, keeping the face ratio in the grids. The face grids, as the GazeCapture described them, have black *squares* as the face box. In more than 99% of the frames, the face boxes are accurate squares. And almost in all the frames of the left 0.5% of the frames, the difference between the face grid height and width is only 1 pixel which is due to the noise of down scaling a processed image to such a small dimensions. And according to Kannan, H. D. [*], preserving the ratios improves the accuracy of the model.

Secondly, for generating the grids in this test, we used MTCNN detections which, considering the Apple's technology in face and eye detection at the time of GazeCapture development [*], outperform the original dataset metadata. And finally, we augmented the face grid with the eyes-nose triangle that contributes to the head pose delineation, and subsequently to the gaze tracking. However, the grid test, compared to the crop test, downplays the high-level visual features of the nose and works with the raw geometry.

3.4.3. Crop

In the *crop* test, we took the tip of the nose coordinates from the MTCNN and the width of the left eye[1] crop from the GazeCapture metadata. Then we cut nose squares out of the frames with the nose tip coordinates being at the center of them and with the same size of their corresponding eye crops. Finally, to make the nose crops consistent with the other crops, we rescaled them to 64 × 64 pixels. We added a subnet to the iTracker model for processing the nose crops, with the exact same architecture as the face subnet. In the concatenation phase, we slide the output of the nose subnet in, along with the other subnets. Except for one extra subnet, the model in the crop test is identical to the original iTracker. This increased the model's trainable parameters to a total of 5,068,738.

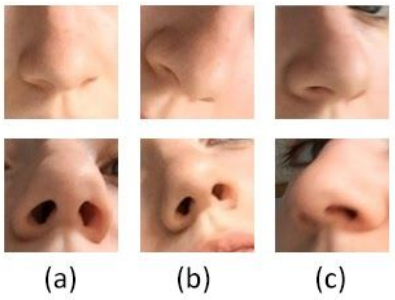


Figure *. The nose crop indications of the head rotation around: (a) X- (b) Y-, and (c) Z-axis.

The crop test is exclusively focused on the nose, not only the coordinates and localization, but also the high-level visual characteristics of the nose. That is, by feeding the nose crop into a

[1] Both left and right eye crops of a frame are squares and have the same size.

CNN, we are hoping the network extracts task-relevant features addressing nose alignment and head pose.

3.5. Experiment 3: The Eyes

In the original iTracker model, eyes subnets share the weights. Theoretically, that means with each input moving through the left eye subnet, the right eye subnet also updates its weights. But practically speaking, the idea is as simple as feeding both eye crops to one subnet. That is, with one sample into the model unlike the other subnets which update weights for one round, the eyes subnet gets two inputs and updates its weights for two rounds. In the eyes experiment, we have changed the core architecture of the iTracker model by setting *two* but *identical* CNNs processing the eye crops. We made the change by replicating the eyes subnet, removing the concatenation phase of the two eyes, and inserting both eye subnets output directly into the list of the outputs in the final concatenation phase. Despite that, the eyes experiment takes the same inputs as the original experiment. However, for splitting the eyes subnets, the number of trainable parameters increased to 5,002,882.

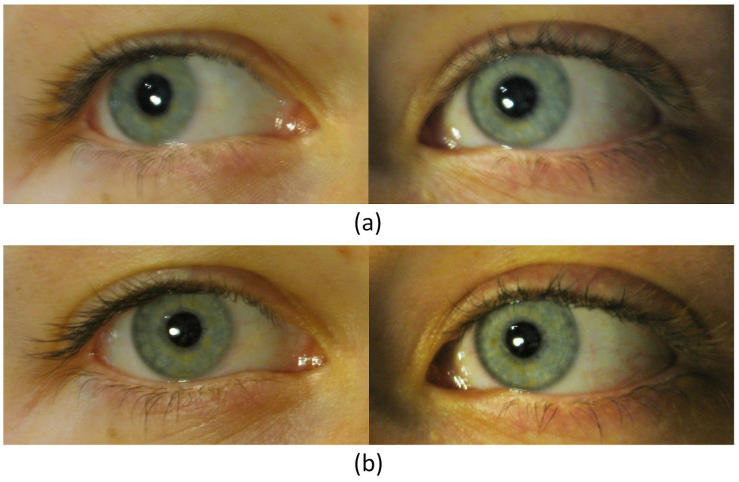


Figure *. The irises' positions in the eye outline is more consistent when the fixation point is at (a) a far distance, compared to (b) a close distance.

The idea of the eyes subnets sharing weights would make sense only if we assume that both eye crops, regarding the gaze tracking task, are conveying the same information and that the right eye is only stressing the features already extracted from the left eye. However, we believe that it is not the case. While in general, we expect that both irises keep the same position

in the eyes outlines, this will be less and less accurate as the gaze location gets closer to the eyes. And that is the case when looking at the screen of a mobile device, as small as smartphones and tablets. Therefore, we decided to test the iTracker performance with two disconnected eye subnets.

Chapter 4

Results

Chapter 5

Discussion

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).

Murphy-Chutorian, E., & Trivedi, M. M. (2008). Head pose estimation in computer vision: A survey. IEEE transactions on pattern analysis and machine intelligence, 31(4), 607-626.

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

Sharma, S. (2017). Epoch vs batch size vs iterations. Towards Data Science, 23. https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9

https://pytorch.org/cppdocs/api/classtorch_1_1nn_1_1functions_1_1_cross_map_l_r_n2d.html

https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D

https://github.com/keras-team/keras/commit/82353da4dc66bc702a74c6c233f3e16b7682f9e6

https://machinelearning.apple.com/research/face-detection