Eye Tracking

Ágoston Csehi

AIT Deep Learning course 2021/1

Introduction

The importance of technologies enabling the tracking of the eyes has been well known to researchers both in the industry and education. I've met with this field last semester when I was working with a classmate on a project to transform a virtual camera's projection based on the user's eyes positions in a coordinate system relative to the screen. The goal was to achieve an effect, with which we could convince the user, that the objects seen on the screen actually exist in his physical space as their movement behaved according to intuitive rules like the parallel effect. We implemented our theory, but one of the main obstacles was the detection of the eye's position for which we settled on using only a webcam, as it is commonly available for most users. The quality and speed of the algorithm we used (Haar's cascade) was poor and insufficient so I decided this semester to implement my own by using deep neural networks.

Task

More specifically, the goal was to construct and teach a deep neural network in order to approximate a function which tells the x and y coordinates of the left and right eyes of the person seen on a picture given to the network as input.

Previous Solutions

This task exists in more advanced and complex forms, like predicting where the eye's gaze point falls onto the screen, so my simplified version can be called a solved problem. In [1] a research on the same topic is conducted, but with more effort behind and much better results.

Dataset

For training data, I used the Gaze Capture [2] dataset. It was retrieved from over 1450 people using IPhone devices and their built-in eye tracking feature. It contains 1,490,959 frames with supplementary information such as the coordinates of the eyes, parameters of a bounding box of the face and so on in json files. The frames aren't scaled for a uniform size, but the width and height parameters are also provided. The total size of the compressed dataset is around 150 GB.

The positions provided have quite a huge noise in them as mobile devices were used to calculate them. Considering this, there is no way my solution will be state of the art, but this project is only for educational purposes. For a real application, I would consider collecting a dataset made by human predictors who would specify the eyes based on an image as this would eliminate all the errors and noise made by algorithms or sensors.

The subjects are from different ages, sexes, and ethnics so it should provide enough diversity for a more generic model.

To be able to download a copy I had to ask for access from the owners for which I had to prove that I am a student at the Budapest University of Technology and Economics, and that I will use the dataset for educational purposes.

Data preparation

Local processing

For training on such a vast dataset, loading it all at once into the memory is just not feasible. To solve this problem, I had to implement a data generator what provides shuffled batches of images and their result data (eye positions) and loads the next batch while the model is still training on the previous one. Luckily there is a class called *ImageDataGenerator* which does exactly that. It has a feature for instantiating a dataframe containing the samples as rows and the path to the image and the label informations. To create such a dataframe I could use a csv file so that's what I had to do. I have broken down the problem into these steps:

- decompress the dataset
- iterate through the frames and the json files
- resize, rename frames one by one and save them to a common folder.
- calculate the eye positions based on data from the json files
- for each frame add an entry to a csv file with the path to the frame and the coordinates

I executed the data preparation on my local machine as uploading the unprocessed dataset to a virtual machine wouldn't seem reasonable. After compressing it again for delivery it came out taking a toll of only a bit more than 1GB in storage space.

Some more details

- I rescale all the images to 256x256 pixels which deformed them but not so much that it would render the prediction impossible.
- The eye coordinates are relative and normalized, meaning the pixel indexed as (0,0) would be in the left bottom corner of the image and the (1,1) would be the upper right one. This way the predictions are applicable easily to the original images, as a product of the scaling being a linear operation.
- I generated the test data to have a separate csv file as a registry, but the frames are in a shared folder. For separation, I randomly choose some frames to put in the test registry instead of the main one based on a probability of 0.1
- I put a limit of 100 of frames that will be included in the preprocessed dataset from each subject, to contain the complexity of the task in the boundaries of educational purposes.

Remote data processing

I ran the training algorithm using virtual machines provided by Google via the Colab framework. To be able to access the preprocessed data, I uploaded it on my Google drive, from which it is accessible from the script by a hardcoded public link. After downloading I unzipped it, loaded it in *dataframes* and plotted a part of it to make sure that it is useable for training.

I created three *ImageDataGenerator*-s for training, validation and testing with frame counts of 112574, 5924 and 13041 consecutively.

Training

Model

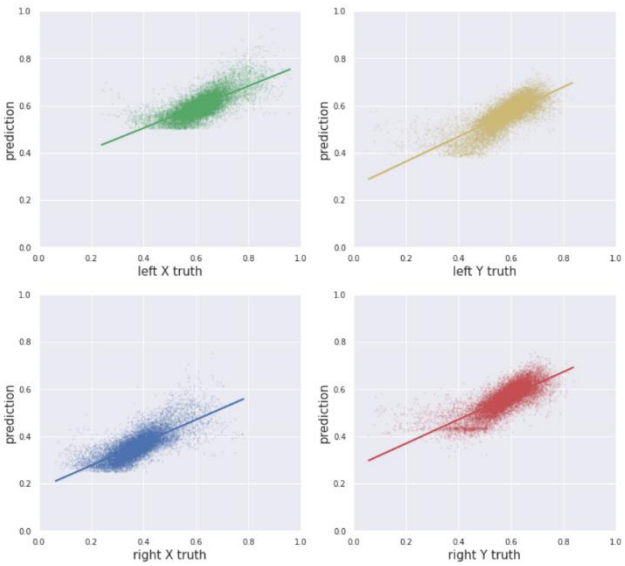
The model contains convolutional, maximum pooling and dense layers. For regularization I tried batch normalization and dropout. The structure is sequential without any skip or residual connections. I tried many configurations but, as stated below, it could not be determined exactly what the optimal one would have been. In the final version the model has 65,924 trainable parameters, which could be a bit too few considering the number of inputs (256x256x3) and outputs (4) but any more than that would have rendered the computational power at my disposal insufficient. One solution to this part would be using transfer learning with a much larger pretrained model, but I wanted to create something from scratch.

Training

For weight optimalization I used Adam algorithm and mean squared error as a loss function as this is a regression problem. There was an early stopping introduced against overfitting, but it can be omitted because the dataset is vast compared to the model.

In the last run the training ran for 30 epochs with a batch size of 2048 using a Tesla K80.

Evaluation



Regression

As seen on the figure, the model could achieve some correlation, but it's far from perfect. The main problem was that on most of the frames the eyes are close to the center, which lead to the regression of the dense layers to this distribution. After finding such a local minimum, it can be quite tricky to force the CNN layers to actually learn something useful instead of leaving all the job to the dense layers.

Positional information

Another possible explanation to this behavior would be, that convolutional kernels introduce a possible loss of positional information. Adding any max or average pooling layers resulted in even weaker correlation of the prediction and the ground truth. It is understandable as their very role, apart from compressing the frame dimensions, is introducing a generalization capability to the network, such as identifying same objects on a frame but at different locations. Simple convolutional kernels have that same generalizing property, with the significant difference, that they are learnable parameters, so they can fit to the frame. This would mean, that they, in fact, can contain positional information as well, but it is hard to extract and even harder to train.

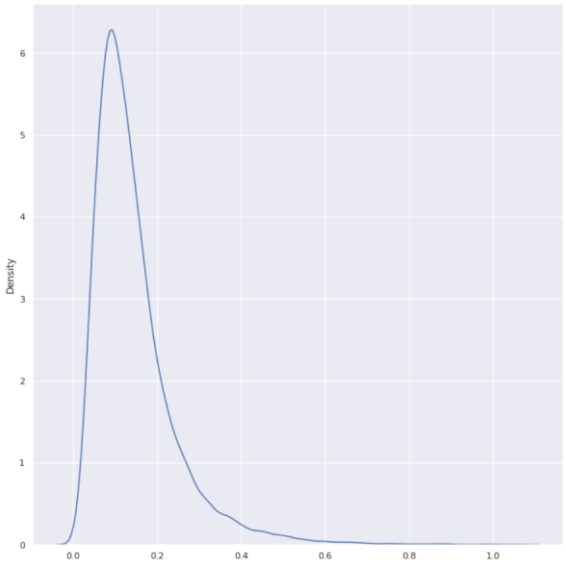
Using a zero-padding at the border of each frame seems to help a bit, as it serves as boundary to which the kernels can adjust their weights. In order to propagate this information to the inner pixels as well, deeper networks could be considered, even though, they introduce more complexity, with which comes inevitably longer training times.

Perfect solution

An intuitive configuration would be using simple eye matching kernels to generate a heatmap and using that with dense layers for prediction. Unfortunately, at least according to my findings, converging on such weight distribution is very slow, as it would mean, that most of the information extraction would be present in the convolutional layers, but dense layers are faster to adapt to the training data, rendering the reaching of this state slower or even impossible.

Loss

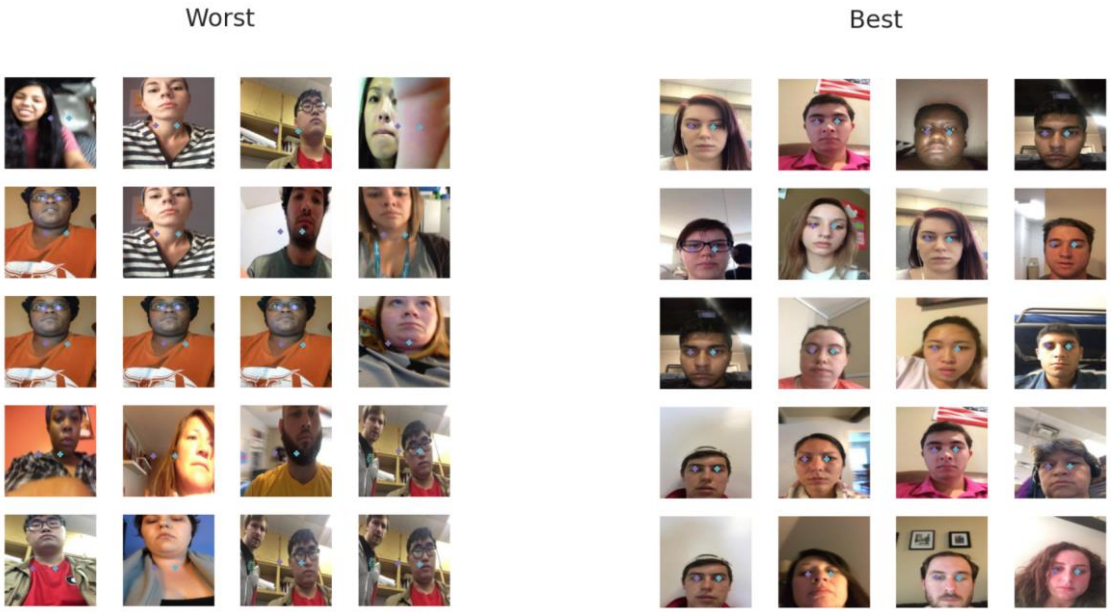
The mean squared loss of the final configuration on the test data set was 0.004



The distribution of the sum of norm 2 distances of predicted eye-positions from the ground truth. Considering the values are relative, so 0.1 would mean the 10% of the whole width of the frame for example, this error is quite significant, as we will see later.

Samples

Here are the worst and the best results.



It is crystal clear, that the problem was not solved, due to the fact that many examples from the worst samples shouldn't have been so hard for the model to solve.

Acknowledgements

[1] Karahan and Y. S. Akgül, "Eye detection by using deep learning," *2016 24th Signal Processing and Communication Application Conference (SIU)*, 2016, pp. 2145-2148, doi: 10.1109/SIU.2016.7496197.

[2] K.Krafka*, A. Khosla*, P. Kellnhofer, H. Kannan, S. Bhandarkar, W. Matusik and A. Torralba, "Eye Tracking for Everyone", *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016*