Distraction Detection using Pose Estimation with OpenCV and TensorFlow

Jacob Augustus Mellichamp*
jakemellichamp@gmail.com
Clemson University
Clemson, SC, USA

Figure 1. Facial Detection and Rotation with OpenCV and TensorFlow

Abstract

Recently our society has been thrust into a work-from-home centric workforce because of the COVID-19 pandemic. Society transformed from working on job sites, learning in classrooms, and sweating-out projects in the library to computing all these tasks on a centralized workstation from home. It has thrown a wrench in work-life balance and I hypothesize that most people are suffering form copious distractions while doing work remotely. This hypothesis inspired me to create an application that could quantifiably measure an individual's attentive vs inattentive behavior while working remotely. By using Python's Tensor flow library for face detection and OpenCV for facial rotation an application was developed to test whether someone is engaging with their workstation.

CCS Concepts: • **Computer Vision**; • **Pose Estimation**; • **TensorFlow**; • **OpenCV**;

Keywords: Python, neural networks, gaze detection, PnP

*Undergraduate Computer Scientist

, ,

ACM ISBN . . . $
https://doi.org/

ACM Reference Format:
Jacob Augustus Mellichamp. 2021. Distraction Detection using Pose Estimation with OpenCV and TensorFlow. In *Proceedings of* . ACM, New York, NY, USA, 3 pages. https://doi.org/

1 Introduction

1.1 TensorFlow Facial Detection

TensorFlow is an end-to-end open-source python library for machine learning. That is, it contains the tools necessary to create and train robust machine learning models. This software has been available since late 2015 and is responsible for developing thousands of machine learning models. Instead of reinventing the wheel and training my own face detection algorithm, I decided to utilize a pre-trained model for facial detection. This model was published by *Yin Guobing* on GitHub and detects 68 facial landmarks that can be utilized in defining a face object[2].

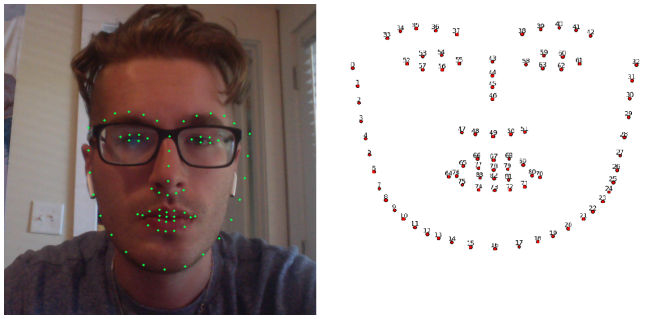

Figure 2. Facial Detection implementation using CNN Classifier

2 OpenCV Pose Estimation

This is where the project deviates from supplied code to experimental code. The Pose Estimation problem boils down too calculating the relative rotation/orientation of the facial object detected. Our class was assigned to solve this problem in one of our previous Computer Vision assignments with the use of Aruco Markers.

This is a notorious mathematical problem within Computer Vision known as the *Perspective-n-point problem (PnP).* [1] The *"n"* represents the number of known and identified points in the image plain. The PnP problem is special because it gives us away of solving the extrinsic camera properties given **predefined 2D coordinates**, **Realtime 3D coordinates**, and the **intrinsic camera parameters** from the current camera.

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

2D Image Coordinates | Intrinsic properties (Optical Centre, scaling) | Extrinsic properties (Camera Rotation and translation) | 3D World Coordinates

Figure 3. Perspective-n-point problem equation

Therefore, when comparing a current image-frame to a previously defined face detection model, the mathematical function will be able to solve the Extrinsic Camera Properties defined in Matrix *R* and *T*.

For this project I will only be concerned with the Rotational Matrix *R* which can be computed with openCV's **solvePNP()** function.

2.1 Implementation Initial Short Coming

After finding the rotational matrix I thought my project was finished, boom, easy. I simply converted the radian values returned from the solvePnP function into degrees and displayed the information. However, although the rotational matrix was solved and seemed to currently illustrate the current rotation of my face. The actual roll, pitch, and yaw values were extremely skewed about the camera.

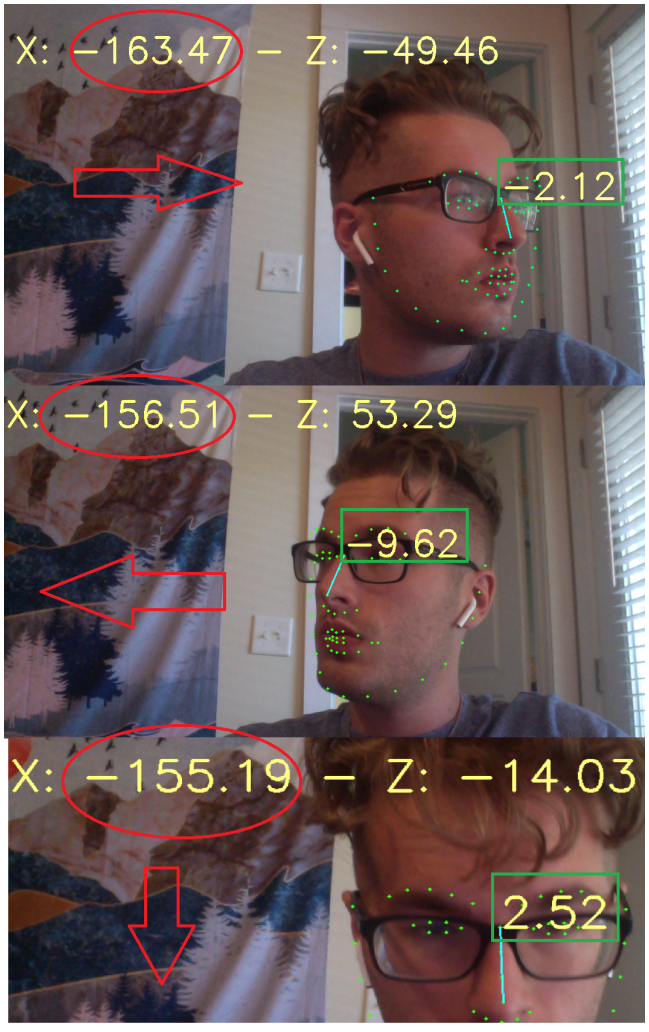

Figure 4. Demonstrating Minimal Change to the Camera Matrix and Skewness

For example, simply looking straight into the camera yields an X rotation of -155 o when in reality it should be 0 or 180 degrees. A slight inconvenience, but something I could fix with adding weights to rotated axis values.

After adding weights, I then added timers and calculated whether the user was paying attention to the camera by examining whether or not the face was rotated within a bound of (-15, 15) degrees for the Y- axis (head tilt) and (-10, 10) degrees for the X-axis (head turn).

2.2 Rotation Testing

*Please Refer to **Figure 6** at the end of the paper to review test data.*

3 Simple Approach

Looking at 23.9 percent-error from the rotation matrix, to simply put it, was not impressive. This was a computer vision inspired project, so I wanted to solve the problem using a computer vision concept. I could not help but notice however, there are three facial marks on the nose marked by Yin Guobing's facial detector (basically the big-dipper star constellation). Therefore, we can utilize two of those points to make a line! Using some algebra and geometry we can solve to find the slope of the nose-line and then calculate the angle via an arc-tan function:

3.1 Arc-Tan Testing

*Please Refer to **Figure 7** at the end of the paper to review test data.*

This means that when a user is facing the camera, the angle will be small (or 90 degrees if perfectly straight) and as the user looks away from the camera, the angle will increase! Let's compare results:

4 Conclusion

All in all, it was a fun experimental project to predict distractions via a pose estimation algorithm. It took me down the rabbit holes of TensorFlow, facial markers, the PnP problem *(and actually understanding it!)*, as well as some geometric functions. It also expanded my problem solving skills as I was initially derailed with the rotation matrix being… not so accurate. The geometric example was clearly the more accurate of the two functions but failed if the user was looking downward at a phone screen. With more time and patience, a better Pose Estimation algorithm could probably be developed to detect when a user is distracted or not.

As I was creating this problem, I realized that pose estimation in general can not determine if a user is currently being distracted or not. Will it work most of the time? Yes, however this problem could be easily solved with an eye-tracking algorithm instead of a pose estimation algorithm.

As a final note on performance, I was also experimenting with fps times to minimize CPU usage when the algorithm

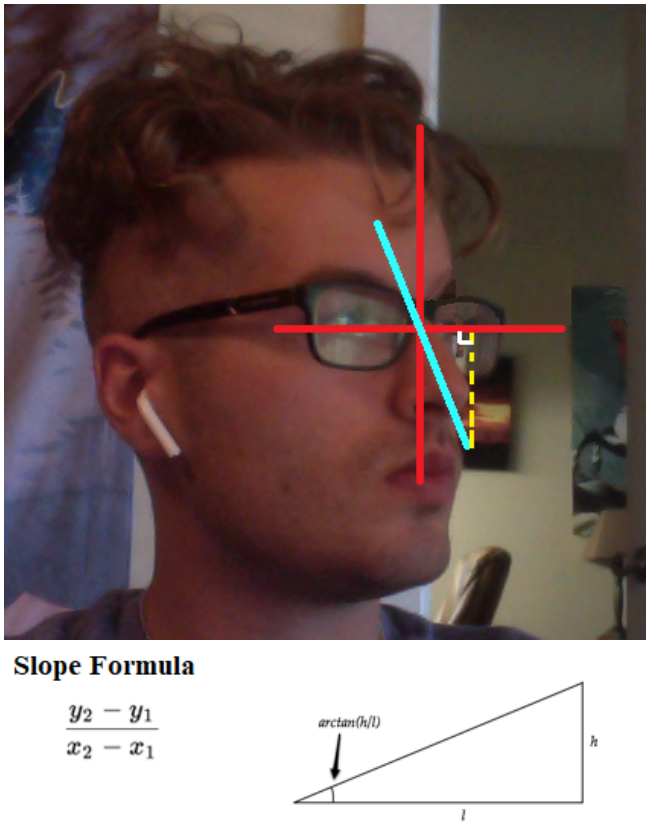

Figure 5. Theoretical image to show tangent line with noses

was running. By utilizing a 0.2 FPS *(or one frame ever five seconds)* a 2.25 Ghz i5 intel processor was able to successfully keep track of distractions with as little as 5% CPU usage throughout the duration of the tracking without an external graphics card!

References

[1] Satya Mallick. 2016. *Head Pose Estimation using OpenCV and Dlib.* Retrieved April, 15 2021 from https://learnopencv.com/head-pose-estimation-using-opencv-and-dlib/

[2] Ying Veytsman. [n.d.]. *HeadPose Estimation.* Retrieved April, 10 2021 from https://github.com/yinguobing/head-pose-estimation

Test #	Purpose	Attentive	Distracted	Expected Att.	Expected Dis.	% Error
1	Distracted Looking Right	4.5 secs	55.5 secs	1.0 secs	59 secs	*5.9 % Error*
2	Distracted Looking Left	7.2 secs	82.8 secs	2.0 secs	88 secs	*5.9% Error*
3	Attentive Straight Face	53.2 secs	8.0 sec	60 secs	1.0 secs	*11.3 % Error*
4	Attentive with variable movement	54.78 secs	18.22sec	72 secs	1.0 secs	*23.9 % Error*

Figure 6. Testing Distraction vs attentiveness with rotation matrix

Test #	Purpose	Attentive	Distracted	Expected Att.	Expected Dis.	% Error
1	Distracted Looking Right	5.5 secs	54.5 secs	0.0 secs	60 secs	*9.1 % Error*
2	Distracted Looking Left	5.2 secs	54.8 secs	0.0 secs	60 secs	*8.6% Error*
3	Attentive Straight Face	59.1 secs	0.9 secs	60 secs	0.0 secs	*1.5 % Error*
4	Attentive with variable movement	57.5 secs	2.5 secs	60 secs	0.0 secs	*4.2 % Error*
5	Looking Down at Phone Screen	50.0 secs	10.secs	0 secs	60.0 secs	***83.3% error***

Figure 7. Testing with the Tangent angle