**Congenital Muscular Torticollis Testing Through Machine Learning in An Android Application**

**Dr. Tamirat Abegaz and Jamie Arrowood-Forrester**

**27 of April 2019**

**Abstract**

This paper attempts to convey the process and hardships of designing and implementing a working model of the first implementation of Congenital Muscular Torticollis Testing in an Android application using machine learning. The paper addresses failed attempts to create a working model of the test app using various machine learning technologies afforded by Google’s in-house machine learning kit and OpenCV. The paper also addresses our means of calculating an intersection angle presented by the official documentation granted to us by Dr. Kimberly Castle and Dr. Rachael Walton-Mouw of the Health and Natural Sciences department of UNG (7). Finally, this paper will try to express performance issues and bugs and technical issues beyond our control.

**Introduction**

Congenital Muscular Torticollis Testing has always been hard to test accurately due to manually having to take a picture, draw perfect lines through the eyes and upper shoulders and then calculating an intersection angle. Dr. Kimberly Castle and Dr. Rachael Walton-Mouw of UNG’s Health and Natural Sciences department wanted to change that. Their idea was to use an Android device to automatically locate the eyes and shoulders, draw the lines through the eyes and shoulders and then calculate the angle of intersection to access the severity of congenital muscular torticollis on infant babies. Considering the new challenge with creating a reliable app to test for the intersection angle, Dr. Castle and Dr. Walton-Mouw pitched the idea to Dr. Abegaz who accepted it and gave me a starter project. The starter project was Google’s sample machine learning kit which was released last year. The starter project contained each subset of detection that the machine learning kit can handle. The two important assets in the machine learning kit in the starter app was text detection and face detection. Dr. Abegaz and I already knew that Face Detection must be used to detect the child’s face, however, we differed slightly on ideas for detecting the shoulders of the child which will be discussed in the following sections.

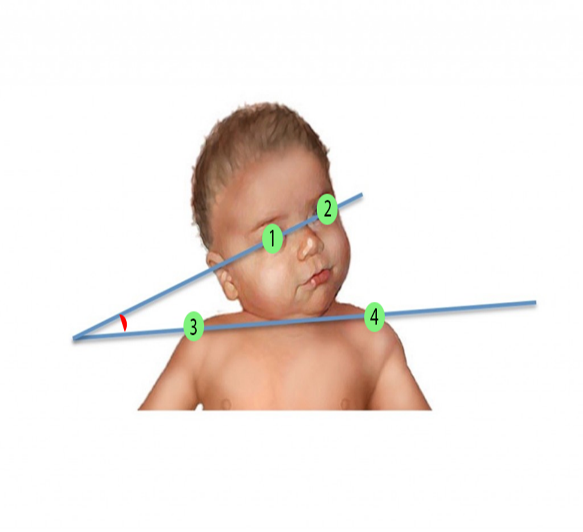
**Face and Text Detection Attempt**

When we started out, I believed that face detection would not be able to work for detecting both the shoulders and the child’s face. I found that text detection was a somewhat viable option as we could place text stickers on the child’s shoulders along with running face detection at the same time. The shoulders could be recognized, however, there were several big problems with this approach. The first problem was that we could not separate out individual text detection results in order to draw a line that spans from one shoulder to the other. Technically speaking, a line could be draw between different text elements if they were no more than an inch away from each other and had very little rotation between them. The rotation allowed was less than two degrees which is unacceptable as the shoulders can have a wide angle of movement even for a baby. The second problem was that the text had to be a straight as possible in relation to the device or the text recognition processor would lose sight of the text and therefore no results would be sent to the accompanying graphic instance. The third problem was speed. The speed issue stemmed from the detector having to identify each text object and then run through multiple nested for loops in order to break the text down into individual elements that the graphic instance class, text graphic, could use for drawing the text elements and accompanying text boxes in the graphic overlay instance. Lastly, we found that we could not reliably run Face and Text Detection simultaneously. Camera Source Class and Simultaneous Detection Issues

As mentioned in the previous section, we could not reliably run face and text detection at the same time. In the Camera Source class, a processing thread and a processing runnable is used to separate the job of running the camera and sending new frames to the underlying detector (note that I did not say detectors) from the main user interface thread. In order to run multiple detectors you have to create an extra processing thread and processing runnable to try and handle the secondary detector as trying to run two detectors on one thread is practically not possible on a mobile device due to the nature of mobile hardware and the design of Google’s machine learning kit. In order to run any detection on an Android device you must first have a Tensor Flow Lite model which until recently was forced to be run on the CPU only in either a floating-point state or an eight-bit compressed variant. In our case, we were using the model that came with Google’s sample code that was compressed already. When you start the application, the model must be uncompressed back to a floating-point state (with reduced accuracy of course) and then put in the cache for faster detection throughout the life cycle of the application. Also, it is only designed to take in frames from one model at a time hence the separate thread and runnable. A bad solution to get around this is to create an additional thread and runnable in the camera source class which I admittedly tried due to a lack of understanding of thread management and the underpinnings of the machine learning kit. Why is this a bad solution? This is a very bad solution as the outcome only results in creating a race condition between the two detectors in where they try to get to the model first. This solution does not create a deadlock; however, it comes close as the choreographer tries to manage the incoming possible results from both detectors. Also, the first detector parameter passed through the frame processor always had the priority over the secondary parameter simply due to initialization. In the end, we ended up with the worst of both worlds, very slow face detection cobbled together with even slower text detection after a month of failed attempts to make the app work in a reasonable manner.

**Face Detection Attempt**

With a month wasted away on a useless and unpractical solution, Dr. Abegaz suggested to completely throw away the use of text detection entirely and focus all our time and energy on multiple face detection. The other professors from the HNS department had already hinted that they would like to use stickers as markers for the line across the shoulders. At first, we experienced the same issue as with text detection in which we could retrieve multiple results, but we could not separate them out into distinct elements. After trying the previous bad solution of running two instances of face detection and creating multiple face detection processor classes for a few weeks, I realized that I was thinking about the project in the completely wrong way. My thinking omitted the fact that the face detection processor was returning a list of firebase vision faces when the task executed. That result list was then passed in through the on-success method where a single for loop iterated through it and sent the results one by one to the Face Graphic class. Out of sheer curiosity, I decided to create two more for loops inside the initial loop and I added two more firebase vision face parameters to the Face Graphic class. Now here’s where it gets worse, I decided to set the following for loops to start at the previous index of the prior loop plus one. For clarification, the first for loop was set up as a regular for loop with the index starting at zero and then iterating through the list of faces from the detector until the index reached the end of list and then the next for loop would start at the position of the previous index plus one and repeat the same iteration on the same list. Surprisingly, this worked quite well even though the time complexity was atrocious. Finally, we had three unique firebase vision faces that we could now use to attempt finding a very crucial angle. Before we could attempt to find any angles, we had to find a way to accurately map the child’s face and shoulders (see Figure 1). Mapping the child’s face was very simple as we only needed to draw a line from the left eye to the right eye at that current time while the bottom faces were a little different. First, recall that the output from the rear camera is flipped along the y-axis and thus causes some confusion on how to set the points. Why does this matter? If you have a face sticker on each shoulder, then in order to draw a correct line you must use the outermost eyes on the two faces. In this case, the face on your left shoulder must be mapped by the left most eye and the face on the right must be mapped by the rightmost eye (See Figure 1). In theory, this would give you the most precise trajectory of the line through the shoulders as those points are closest to the edge of each shoulder, however, this was way too ideal of an idea.

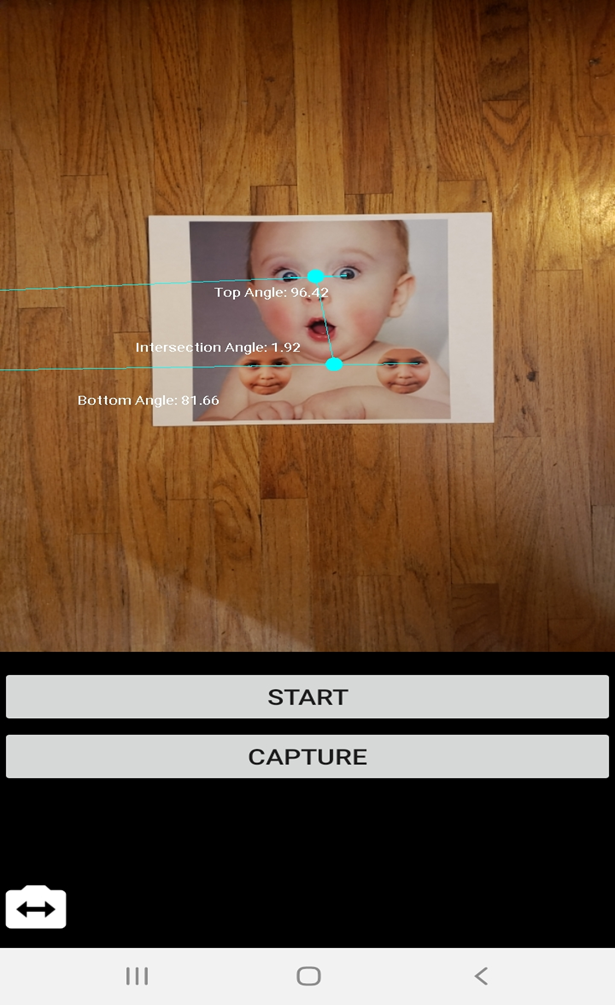
*Figure 1: Two face stickers are placed on each shoulder with the leftmost and rightmost eyes being as close to the edge of the respective shoulders as possible. The face stickers are represented by number 3 and number 4 in the picture on the left.*

**Initial Calculations**

After determining the start and end of the upper and lower lines, Dr. Abegaz decided that we should implement the calculations of a side-angle-side triangle. Using a side-angle-side triangle we could easily compute the angles using basic trigonometry and geometry. First, we decided that we needed a better point to represent the upper face in order to create a usable triangle. To create a usable triangle, we merely found the average x and y coordinate between the child’s left and right eyes and set a point at that location on the canvas. Next, we drew lines extending from the upper middle point to the leftmost and rightmost eyes of the lower faces. Then, we computed the upper slope by a variable y equals to the y coordinate of the child’s left eye subtracted by the y coordinate of the child’s right eye and another variable x set to the x coordinate of the child’s left eye subtracted by the x coordinate of the child’s right eye before dividing y/x. The same approach was taken for the lower slope as well except the variables were the x and y coordinate of the leftmost eye and the x and y coordinate of the rightmost eye (2).

For our purposes, finding just the slope was not enough as we still lacked knowledge of the point of intersection. Thus, we needed the y-intercepts of the upper- and lower-lines using variables named x, y and b for simplicity. For the upper line, y was set to the y coordinate of the child’s left eye, m was set to the slope of the upper line and x was set to the x coordinate of the child’s left eye while the variable b was set to the y-intercept. To calculate b, we retrieve the variable y from earlier and set x equals to the slope of the upper line multiplied by the x coordinate of the child’s left eye, and finally, b is set to y-x. Finding the y-intercept of the bottom line followed the same algorithm except the x and y coordinates of the leftmost eye were used (2). With the equations of the upper and lower lines computed, the point of intersection for the non-parallel lines can be determined by setting x equals the y-intercept of the lower line subtracted by the y-intercept of the upper line that is then divided by the slope of the upper line and subtracted by the slope of the lower line. The result of this calculation is then used in either the equation of the upper line or the equation of the lower line for accuracy checking (5). Now that a triangle is made using the gathered points, the distance formula had to be implemented. To find the distance of each line we first need the coordinates of both the upper and lower middle points on the screen, the leftmost bottom eye and the rightmost bottom eye. In following our current methodology, the upper line distance was calculated first by taking the square root of the x coordinate of the point of intersection subtracted by x coordinate of the upper middle point (all squared) plus the y coordinate of the point of intersection subtracted by the y coordinate of the upper middle point (all squared). The same is done for the distance between the upper middle point and the lower middle point as well as the distance between the lower middle point and the point of intersection (6).

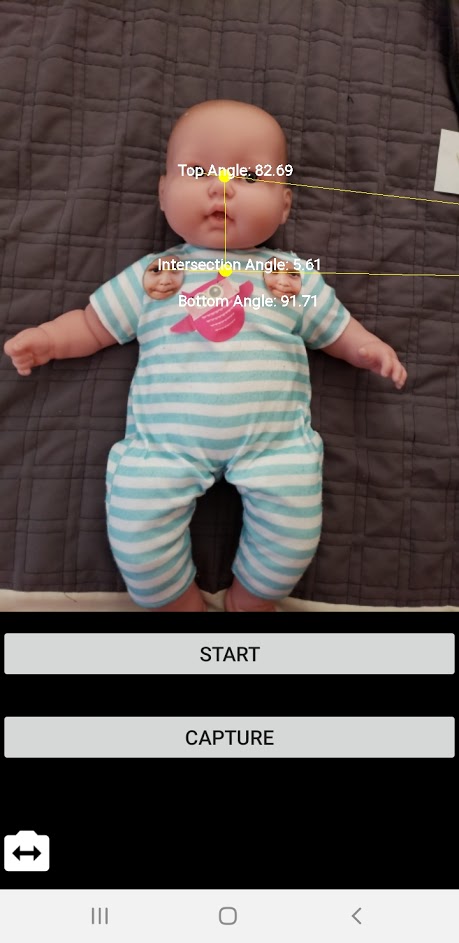
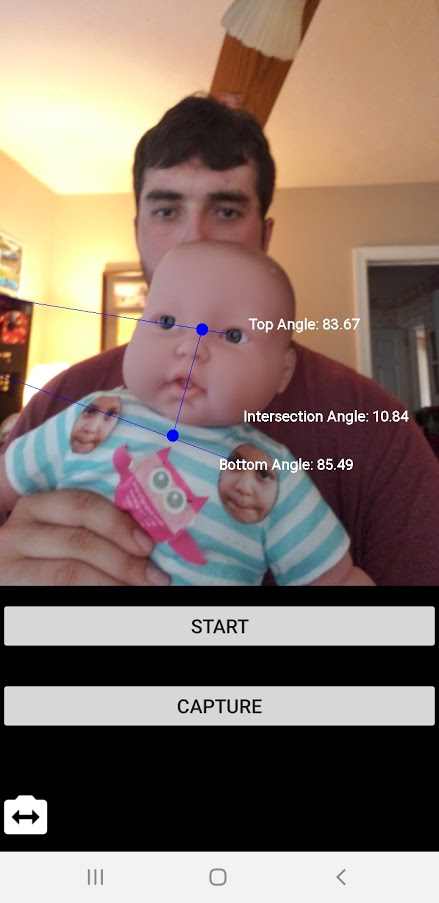
Finally, we can now obtain the top, bottom and intersection angles using the shorthand version of the law of cosines. The top angle equals the middle connecting line distance squared plus the lower line distance squared that is subtracted by the upper line distance squared which is divided by two and multiplied by the middle connecting line distance and the upper line distance. The intersection angle equals the upper line distance squared plus the lower line distance squared subtracted by the middle line distance squared and then divided by two multiplied by the lower line distance and the upper line distance. As the inner angles of a triangle must always add up to 180 degrees, the bottom angle was set to 180 degrees (converted to radians) and subtracted by both the top and intersection angle which is displayed on the screen (See Figure 2) (6).



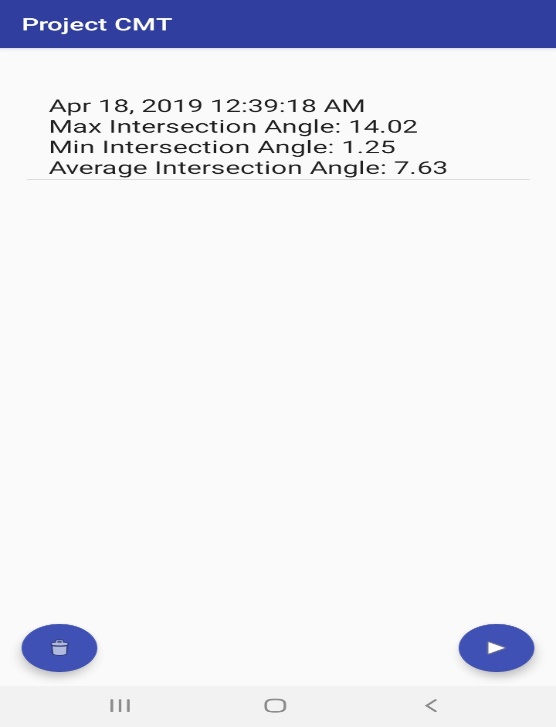
*Figure 2: The top, bottom and intersection angle values displayed on the screen after successful detection during early testing. Two main issues here are that the lower faces could get switched under a heavy angle causing a bad result and the face stickers could be incorrectly positioned by the therapist.*

**Testing**

Until the middle of April, the only testing we were able to perform was through using a piece of paper as our mock baby as in figure 2 above shows. As there is a great variance between hardware capabilities between Android devices, we used three different devices from Samsung. The devices were a Samsung Galaxy J3 Eclipse, a Samsung Galaxy J7 Eclipse and a Samsung Galaxy Note 9. We went with these devices to try and get the best performance index between devices due to the overall popularity of the Samsung brand in the Android market. The main difference, however, was the camera which proved to be our main issue regarding initial testing. We found that the lower resolution camera in the J3 compared to the J7 greatly affected the overall likelihood of even detecting the face stickers on the shoulders. This, of course, means that the Note 9 could detect all three faces with relative ease due to its higher resolution camera. In the middle of April, we were given a baby doll for simulated real life testing by Dr. Castle (see Figure 3). As expected, the devices had a much harder time of detecting the faces. It became evident that persistent storage of the maximum intersection angle was needed if detection should fail or be sporadic in nature due to lighting on the lower faces or extraneous device angle. Not only would we need the maximum intersection angle, but we would need to store the minimum and average intersection angle due to the possibility of outliers while the detector follows the movement of the three faces (Figure 4). Also, the nested for loops mentioned previously in the on-success method were removed in favor of a simple check of if the list of faces returned by the detector was equal to three or not.

*Figure 3 (above): Initial testing with the doll which proved to be much harder to detect than the 2d baby faces of previous testing. Also, the lower face switching bug was resolved by taking the middle point from each lower face as we did for the upper face. Note: We were not told that all testing would be done through the front camera while a physiotherapist holds the baby until the middle of April.*

*Figure 4 (below): Results of the max, min and average intersection angle along with a current timestamp after completion of a testing event.*

In the initial documentation received from Dr. Castle, all testing was shown to be performed with the child laying on a table with a camera positioned above the child while a physiotherapist gently bends the child’s neck from left to right (1) (7). Once the child’s neck was bent as far as possible without distressing the child, another physiotherapist would take a picture from which calculations could be made (7). As mentioned in the subtext of figure 3, Dr. Castle wanted to hold the baby and move the baby’s entire body from left to right until the child’s body and head is nearly vertical. Unfortunately, the range of face detection in Google’s machine learning kit cannot handle such an extreme rotation in relation to the entire child’s body regarding landmark detection. Between 18 and 36 degrees in either direction, no results could be given as either one of the eyes could not be detected. For instance, if the doll was rotated past 36 degrees to the left, the right eye could be detected (this limitation is clearly written in the official documentation) (3). There is one landmark that can be followed throughout a greater range of motion which happens to be the base of the nose. Nevertheless, using only the base of the nose for the child’s face would result in useless results due to the undirected slope of the upper line.

**Conclusions and Future Work**

The initial and final requirements portrayed a false hope that a mobile device could not only be utilized for congenital muscular torticollis testing but be a reliable replacement for the current methodology of testing using still photography and manual intersection angle calculations. Google’s in-house and easy to implement machine learning kit proved to be invaluable to accurately position the needed non-parallel intersection lines for which simplistic geometry and trigonometry calculations were made. Nevertheless, the app simply could not cope with the final requirements pushed on at the last minute due to device versus the baby’s full body rotation in the arms of a physiotherapist. In order to fulfill the final requirements, the therapists would need a camera connected to a computer (such as a laptop and not a mobile device) that is trained to follow dots placed on areas of interest on the child’s body those being namely the face and shoulders. With this approach, there would be no need to hide the therapist’s face from the camera which invalidates the testing process entirely. This new approach, however, will not be implemented or tested by me or Dr. Abegaz as far as we are aware of.

**References**

1. Bhaskar, A. & Harish, U. & Desai, H. (2017). Congenital muscular torticollis: Use of gaze angle and translational deformity in assessment of facial asymmetry. Indian Journal of Orthopedics, 51, 123–130. [https://www.ncbi.nlm.nih.gov/pmc/articles/PMC536146 1/](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC536146%201/)
2. Equation of a Line (slope and intercept form). (2011). Retrieved from <https://www.mathopenref.com/coordequation.html>
3. Face Detection Concepts Overview. (n.d.). Retrieved from <https://firebase.google.com/docs/ml-kit/face-detection-concepts>
4. Firebase Quickstart Samples for Android. (n.d.). Retrieved from <https://github.com/firebase/quickstart-android> \*
5. Intersection of Two Straight Lines (Coordinate Geometry). (2011). Retrieved from <https://www.mathopenref.com/coordintersection.html>
6. The Law of Cosines. (2017). Retrieved from <https://www.mathsisfun.com/algebra/trig-cosine-law.html>
7. Rahlin, M. & Sarmiento, B. (2010). Reliability of Still Photography Measuring Habitual Head Deviation From Midline in Infants With Congenital Muscular Torticollis. Pediatric Physical Therapy, 399-406. doi:10.1097/PEP.0b013e3181f9d72d **\***

\*: An older version of reference 4 was supplied by Dr. Abegaz in January. Reference 7 was supplied by Dr. Castle in January.