Toward Hand Gesture Recognition for Sign Language Communication

ENSC 482

1st Justin Singh
Applied Science (Engineering)
Simon Fraser University
Prince George, Canada
jksingh@sfu.ca

2nd Maryam Darbehani
Applied Science (Engineering)
Simon Fraser University
Vancouver, Canada
mdarbeha@sfu.ca

3rd Alexis Golding-Ulm
Applied Science (Engineering)
Simon Fraser University
Vancouver, Canada
alexisg@sfu.ca

***Abstract*—This project report will introduce a method for hand gesture recognition which can be used to help people with hearing impairment to communicate with others. The objective is to capture the hand gesture using a camera, recognize the corresponding American Sign Language sign, and translate to a letter of the alphabet. The project is accomplished using Python programming language and OpenCV library, and using any camera that can stream to a computer or processor, such as a laptop camera.**

I. Introduction

The project focuses on successfully capturing a sequence of hand gestures via camera and image processing and translating the hand gesture to letters. The most difficult and important part is to differentiate between different hand gestures. Moreover, the algorithm shall work for any different hand size and for any skin color. The starting point of some of the algorithms have been from Reference [1].

II. Objectives:

The project shall successfully achieve the objectives in the following order:

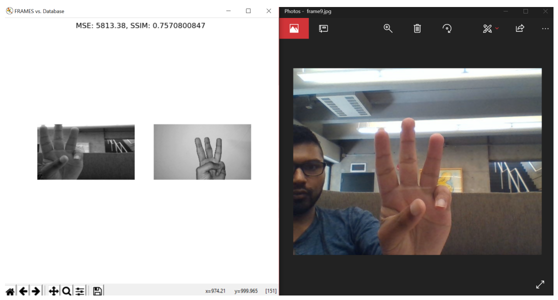
1) Rendering video footages of the camera into frames
2) Segmenting the area of interest from video frames
3) Feature extraction
4) Comparison algorithm
5) Training
6) Translation of the hand gesture to the text

III. Rendering Video Footage from Camera into Frames:

A. *Initial Approach*

The first approach was to extract frames from a given video file of American Sign Language just to see how many and how fast frames should be captured when the program finally decided to capture from live video. Doing this, we observed that capturing after long periods of time often led to the wrong moment being captured and most letters not being optimally oriented in the frame; capturing every second or under, although producing the most frames to later compare to a database, also produced blurry frames of movements in between sign-language that we did not want. We saw when comparing to a database, that these frames made it difficult for comparison as it resulted in incorrect matches after vectorizing the image.

This observation led us to research potential ways to filter out the blurriest images before comparing to a final database for the user. Many different implementations were found and experimented with, however two methods, whose tolerances could be set to improve the rate of sucessful matches, were eventually implemented in our solution: The Mean Squared Error approach and the Python-native function Structural Similarity Index Method. MSE, although a simple algorithm that sums over the square differences of a picture pixel by pixel when vectorized, does give a rough enough estimate of a potential match despite the image's typical noise such as lighting and background shadows. The closer the MSE's vaue was to 0, the closer the two images being compared would be, so this was usually set at less than 5000 after numerous testing in different environments. The SSIM comparison was provided by the library "scikit-image" that specializes in image processing functions. SSIM works similarly to MSE, however, SSIM accounts for the texture of the image, while MSE fails to do so. Based on a scale of 1 representing an identical match and 0 representing a disimilar image, the tolerance for the SSIM vector was originally set at 0.9. Figure 1 below displays a successful trial run using this implementation with the MSE and SSIM values outputted to the screen.



```
MSE: 5813.38, SSIM: 0.7570800847
```

Fig. 1. Successful Run using Initial Frame Extraction

The fatal error found in this approach ended up being the fact that the background noise behind the sign was too much to differentiate from, resulting in either too many false positive matches when tolerances were set lower and almost no matches when tolerances were set higher. This code that represented this portion of our research can be found in the deliverables titled *VideoFrame2.py*

B. *Final Version*

The first approach was to save the video frames and loop over them to get the hand gestures. However, there were some ailments such as the movements while changing the hand gesture and the memory required for saving and looping over the frames of the live video. Therefore, the team decided to add a key for capturing only the desired frame when the user is sure about the hand gesture. This way we eliminated having to compare multiple frames against the database and having to worry about the frame capture rate. In order to achieve that, the user shall press a key (in our case, the keyboard letter "B", but a wireless key for use by a robot could be implemented in the future) and ask the program to find the meaning of the hand gesture. Thus, allowing the frame to be taken again if the user decides the frame taken was not of the best quality.

IV. Segmenting the Area of Interest from Video Frames:

A. *Initial Approach*

Our initial idea for segmenting the area of interest came from another GitHub project that we drew inspiration from when it came to transforming an image and comparing it against a database set [2].

The idea involved cropping the bottom 15 pixels of the image, as those represented where the arm typically would be in the frame. Then the image would be resized to match the dimensions of the database image for a 1-1 comparison of pixels when vectorized. The CV2 library *resize* function was used to resize the frame taken from the live video before comparison, but it was soon realized after many tests and debugging that this method of resizing did not maintain the pictures aspect ratio, resulting in the image becoming distorted quite heavily before comparison and resulting in matches becoming more random than actually accurate [4]. Figure 2 displays this behaviour.

Conducting more research to find a more effective *resize* function started to take up a large portion of time, which was allocated between the tasks of finding a more preferred method to resize and an alternative method to compare the images other than pixel by pixel. A proper image aspect ratio preserving function would be found, but a better solution was eventually decided upon. The code with a modified image resize that preserved the aspect ratio can be found in the Python script *VideoFrame2.py*.

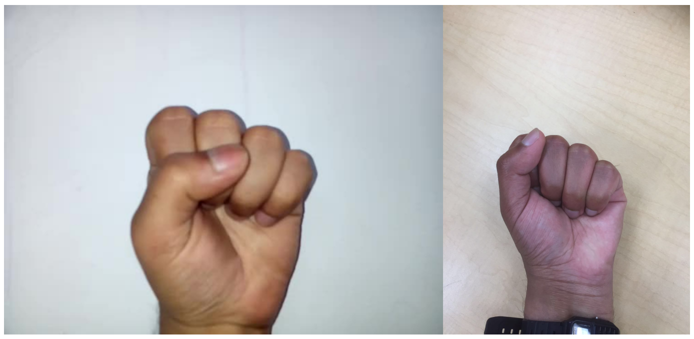


Fig. 2. Distortion of Database Photo (Left), When Resized to Match Frame (Right)

B. *Final Version*

There are several points that need to be considered for segmenting the area of interest, namely background, body, and face of the user. In order to have a clear hand gesture, we decided to create a box (rectangle) at the top right side of the video footage and have a predefined point for the position of the center of hand. This feature eliminated the need for subtracting the body and the face from the frame as only the hand of the user will be placed within the box. This method also addressed the size of the image when comparing pixel by pixel in vectorized form as the image is now initially the same size as the images in the database.

Although this fixed our resize issue, we had already dedicated time to creating a database of a different size and now this would have to be redone. Additionally, background noise was still a dilemma and needed to be solved before a comparison of this image could be done. It was realized that not only did the frames of live video suffer from background noise, but also the database pictures themselves, as they were taken of different group members' hands at different times during the day in different locations. There was still the need to subtract the background from the frame.

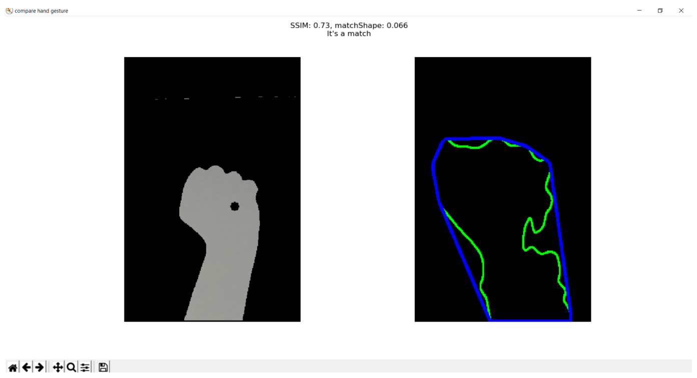
A built in function from OpenCV, *CV2.BackgroundSubtractorMOG2*, was used to subtract the background and get the foreground (hand). This function only works for video footages and subtracts the moving foreground from the stationary background, resulting in a model background. The background can then be eliminated from the frame by building a background subtracter model and applying the function *CV2.bitwise_and* from the OpenCV library to acheive a masked frame.

Unfortunately with our database already completed, this method was not able to be done for all of our pictures. Instead, a small database was used in our final design of previous positive test results, as seen in Figure 3. Reference [3] was the inspiration for segmenting the area of interest using a box on the screen.

V. Feature Extraction:

A. *Initial Approach*

The initial approach was to extract the contour of the hand to match the shape of the segmented frame hand to the one in the database. First we needed to determine the area of interest and outline the shape of the hand. Then the background would be filled with a solid colour so that only the image of the hand would remain. The resulting image is a bi-color image of a shape on a solid background. The transformation process was done for both the database and frame image, so that the algorithm could compare the points

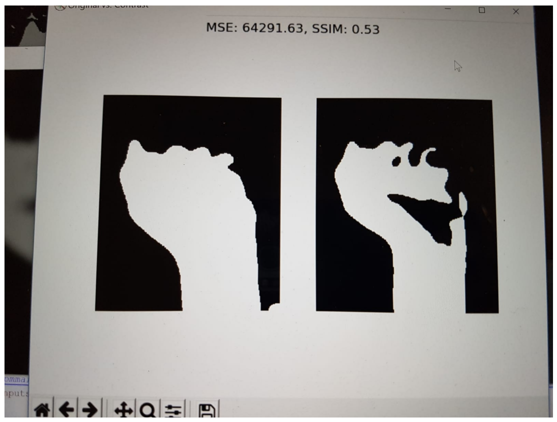


SSIM: 0.73, matchShape: 0.066
It's a match

Fig. 3. Final Version of Database Photo (Left), Comparison to Frame (Right)

of interest to determine the similarity of the hands in the two pictures.

While testing this method, issues became apparent with shadowing and lighting on the contours, causing black holes inside the largest contour and resulting in poor accuracy in comparison with the database contours, as seen in Figure 4. Although the idea failed, it showed us that extracting foreground and making the background black was imperative.



MSE: 64291.63, SSIM: 0.53

Fig. 4. Contouring Issue of Holes Present in Frame Contour (Right)

We then moved toward filtering and image correction methods that were done commonly in other image comparison projects seen on GitHub. To enhance the contrast in the image, the first thing we attempted was making the background black around the image by using CV2's built in morphology transformations to erode and dilate with image processing. The *erode* and *dilate* operations stripped out the outermost pixels in the images and enhanced the pixels near the kernel.

A kernel is a convolution matrix or mask that can be any shape, (ours being elliptical as CV2 has another useful function for this). Erosion and Dilations work as 2D convolution achieved through the kernel sliding through the image. A pixel in the original image (either 1 or 0) will be considered 1 only if all the pixels under the kernel are 1, otherwise it is eroded (equivalent to 0). The opposite is true for dilation, where the pixel becomes 1 if at least one pixel under the kernel is 1. A second mask was then also used to blur and help remove noise around the image outline.

Next the skin was made white to contrast with the now black background. To do this, the image was first converted to grayscale and blurred to help remove more noise highlighting the main object using a Gaussian filter. Now, setting the threshold to 1, the image is looped through pixel by pixel making any that are higher than the threshold white (255) and anything else black (0).

Afterwards, we centered the image using some math and the basis that a contour is all that we care about in the image by cropping the edges a bit. Finally we downsized the image vector (the aspect ratio was preserved sdespite these modifications) to match the shape of the picture in the database.

This iteration was eventually scrapped as we were able to solve the resize dilemma without needing to resize the pictures. A lot of the filtering done as well as the main feature extraction idea of contouring was eventually kept for our final version.

B. ***Final Version***

The final version, as stated previously, kept many of the vital components of filtering and extraction from the initial iteration. Converting the frame to grayscale and using Gaussian blurring to reduce the edge contents was done this way so that the frame could be converted to black and white easily by applying the binary threshold function. The reasoning behind getting only a black and white image is to increase the contrast between the hand (will be white) and the background (will be black) in order to find the contour of the hand gesture.

To do this, the methodology was kept the same as the initial iteration. The contour was then found using an OpenCV function called *findContours*. An example of all the blurring and filtering can be seen outputted every time anyone runs the final script titled *project2.py* The script outputs the mask, the Gaussian blur of the mask, the contour outline and finally the contour outline with a overlayed Convex Hull (used for more filtering and edge detection of the object) seen fully in Figure 5 below.

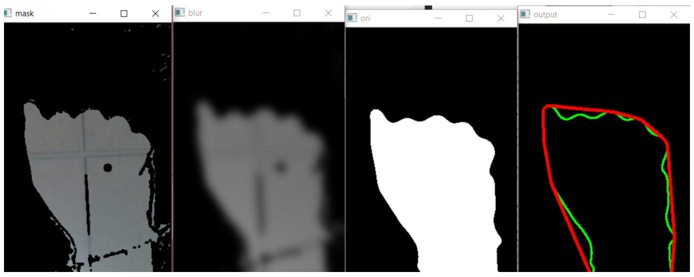


Fig. 5. Image Transformations Done on a Frame before Comparison to Database (In Order from Left to Right)

VI. Comparison Algorithm:

A. ***Initial Approach***

Originally, as mentioned previously when extracting frames, the comparison algorithm was based off of a Mean Squared Error and Structural Similarity Index Method calculation to tell the similarity between two pictures. We realized this was not going to be enough from the beginning, but went ahead with the method in an attempt to remove intermediary frames from the large amount that would then be compared to a database of pictures through machine learning models (see Figure 6). As our idea for this project became more realistic however, we eventually didn't need this intermediary frame extractor for our final result, but it was completed and can be perused throughly in the *VideoFrame2.py* Python script.

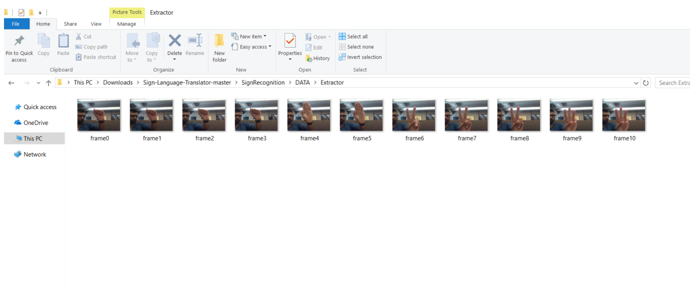


Fig. 6. Frames Left in Extracted Database after Script to Remove Intermediary Frames was Run

B. ***Final Version***

The comparison is completed by comparing the contours of the live video frames with the contours of the images in a database as well as using the Structural Similarity Index Method which compares the similarities between the image of the hand from the live video and the images in the database. There is no requirement for resizing the frames as the database has been created using the same size of the box which the hand is placed in during the live video.

As for comparison of the contour shape, the function *matchShape* of the OpenCV library is being used and the closer the result is to 0, the more similar the shapes of the contours are. As for the Structural Similarity Index Method, the *ssim* function of Python is used and the closer the number is to 1, the more similar images are.

The threshold for these comparison values are set according to try and error and having different size of hands with different distances from the camera in the database.

The program loops over the database and tries to find the match and displays the match in a separate window upon recognizing a match.

VII. **Training:**

A. ***Initial Approach I***

Training was initially done with the inspiration from Reference [2]. In his project, the user is given a choice to specify between the machine learning models "SVM","k-NN" or "Logistic", as all three classifiers are built in algorithms from the library *sklearn*. The library *sklearn* has a variety of functions supporting machine learning with integration of OpenCV with Python.

For example, if one were to invoke the *knn* object from the sklearn library, a vectorized form of a training set, comprised of positive images, coupled with their corresponding labels, would need to be passed to the *fit* function for the knn object [7],[8]. By experimenting with this project, it was found that with training from k-NN, invoking the *predict* function on a test image from live video script resulted in false positives and rarely found correct matches. When trained with the SVM method, the predictions became very sparse, and rarely ever predicted at all. When training with the Logistic method, it acted similarly to SVM with sparse predictions as well. When comparing live video testing to comparing static test photos to the trained database, it was found that the results from test photos were largely more accurate.

This agreed with our results as testing from live video introduced multiple possibilities of potential impurities into the system causing random outcomes. These "forces of nature" were foreseen whenever possible and solutions were found to overcome them. Solutions were not found for all of them however, given our limited resources and time for this project.

In general, it was found that the more samples and the more diverse your database is when training your algorithm, the better results you will have. Our database was unfortunately unable to be that large given time constraints, but this was compensated for by limiting the possibilities to only a few letters of the American Sign Language alphabet (see Figures 7 and 8).

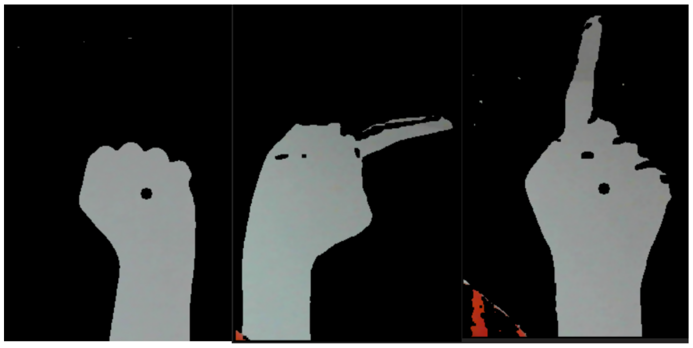


Fig. 7. Letter "A", "G" and "I" That can be Detected with The Final Algorithm

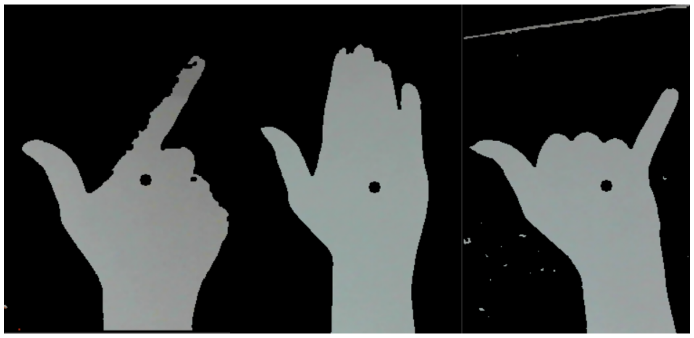


Fig. 8. Letter "L", "B" and "Y" That Can be Detected with the Final Algorithm

B. ***Initial Approach II***

Upon more investigation into this subject, it was discovered that coupling a detector with a classifier could be a feasible approach to this problem [9]. The detector and classifier would have to utilize Histogram of Oriented Gradients (HOG) vectors of images as their training sets. HOG vectors account for the gradients of each group of pixels of an image which quantify the general shape of the features within an image, rather than the colour. Stronger gradients dominate the overall gradient direction of the HOG vector [10]. Ideally, a hand in the image would be the dominant feature. Additionally, given the fact that this solution could be used by a variety of different people against a variety of backgrounds, choosing to ignore features such as skin colour and background colour provides an optimized solution.

Training the HOG detector from the CV2 library using *hog.setSVMDetector* with a Linear SVM classifier (derived from the scikit-learn library) vector trained with the HOG vectorized training set, comprised of positive and negative samples, would ideally be able to detect a hand in the image frame. From there, the test image could be cropped down to where the hand was found in the image, and passed to a Linear SVM classifier.

The Linear SVM classifier is trained with HOG vectorized images of each letter of the ASL alphabet and their associated labels such as 'A', 'B', 'C', etc. using the function *fit* upon instantiating the SVC object. Once the test image from the HOG Detector has been processed, it is fed to the classifier. Functions such as *predict* and *predict proba* return the predictions and their probabilities, respectively, made by the classifier.

C. ***Final Version***

Following Initial Approach II above, an attempt at the implementation of HOG detector and Linear SVM classifier ensued. The HOG detector was trained with a Linear SVM vector trained with positive and negative samples of the training set. The training set, retrieved from Reference [9], involved ground-truth bounding boxes at which each image had to be cropped to have only the hand remain. The negative data set was retrieved by sliding a window over an uncropped positive sample and computing the Intersection over Union (IoU) between the negative window and the cropped positive sampled. If the IoU was $<$ 0.5, then the negative window was appended to the negative data set. The code for the sliding window was derived from Reference [11], and the code for computing the IoU was derived from Reference [12]. 5 negative samples were retrieved from a positive sample, to ensure that the negative training set was larger than the positive training set. Each set was associated with a 'hand' and 'not hand' label, respectively.

Subsequently, once the HOG detector was trained, the detector was improved using Hard Negative Mining (HNM). HNM follows the same algorithm used to determine if there is a hand in the image, namely sliding a window across the image, feeding the image into the detector, determining a prediction, and then determining the window with the greatest area across the hand using Non-Maximum Supression Algorithm [13], as multiples will arise from a multitude of matches. HNM includes the false positives, however, by computing the probability of the prediction and adding those with lower probability to the negative training set. Then, the HOG detector is retrained.

The Linear SVM classifier was trained as mentioned above. This implementation of the classifier resulted in surprisingly poor results. Although the predictions for the test image was predicted correctly among 5 predictions, the correct prediction was never the best probability. Moreover, any fist-based signs were never correctly predicted, indicating that the thumb as a feature was not a strong gradient. Additionally, it is suspected that the quality of the test images versus that of the training set resulted in skewed HOG vectors, as the former was much noisier than the latter. It would have been prudent, had time allowed, to construct a training set for this particular instance. This exposes a problem with this implementation, as it indicates that any new user would have to re-train the classifier with their particular hand and background environment. In later versions, this could be implemented as an out-of-the-box calibration sequence. Due to the unreliability of this method, this training implementation was not included in the final version of the project.

VIII. **Problems:**

A. ***Found and Fixed***

1) ***Resizing and Keeping an Image's Aspect Ratio***:

- Originally found that resizing a picture to another size causes distortion.
- Function was created that used thumbnail design to keep aspect ratio degradation minimal.
- Vital component of many machine learning algorithms with live video feed.
- Eventually not needed, but extremely useful for future image processing projects.

2) ***Understanding Versioning Between OpenCV and Python 2.7***:

- Incompatibility is incredibly common throughout different versions of different libraries with each other in Python.
- No backward compatibility for many libraries meaning having the newest iteration does not guarantee compatibility.

- Sample code commonly does not come with a list of dependencies with versioning of what version they had when it was originally developed.
- If this occurs, must update function headers for current versions or find new functions that preform the same task if the function is now deprecated.

B. *General*

1) *Detection Is Unreliable:*

- Ideal conditions do not occur in real life, and since all hand gestures are quite similar, false-positives occur frequently.
- This is to be expected. Current laptop facial recognition technology cannot reliably recognize their user anywhere other than in a specific location under certain conditions. Why is this? Shadows and lighting play a major role in detection. Additionally, minute features that are not "strong" features cannot be detected [14].

2) *Other Common Problems:*

- Inclusion of database with left hands vs. right hands?
- How far back or forward does the user need to keep his hand to be able to match a picture in the database?
- What type of lighting does the picture need to be taken in?
- What if the user wears rings or other jewelry?
- How does one compare against dynamic signs like ‘J‘ and ‘Z‘?
- How do we set the tolerances for the SSIM and MSE functions?

IX. **Future Work:**

- Add a reward and punishment algorithm for when the program is correct – it will use this knowledge to build its knowledge of its database in different conditions (lighting, background noise etc.).
- String together multiple learning methods in a type of chain i.e. deep learning.
- Dynamically subtitle sign language in videos using frames from a live video stream.
- Implement video comparison for dynamic letters like “J” and “Z”.
- Dynamically “Autocorrect” user for fast sign language or different positioning of hand movements.
- Display an English vocabulary for the user in real time as a 1-1 translation.
- Build up training database and dynamically add successful runs to the database for further learning.

X. **Dependencies:**

Needed to run the code as specified are the following libraries Note: Some libraries may have dependencies on other libraries, which may need to be installed:

- kiwisolver 1.0.1
- matplotlib 2.2.2
- numpy 1.14.5
- opencv-python 3.4.2.17
- pandas 0.23.3
- Pillow 5.2.0
- pip 18.0
- Python 3.5
- python-dateutil 2.7.3
- scikit-image 0.14.0
- scikit-learn 0.19.2
- scipy 1.1.0

Acknowledgments

First, we would like to thank Zane Lee for his GitHub project [1], that we ended up building off of and drawing inspiration from for our final iteration of our project. We would also like to thank Anmol Singh Jaggi for his Sign Language Recognition project [2] that without, we would still be figuring out which filtering works best for what when it comes to image transformations for comparison against a large database using machine learning techniques. Finally, we would like to thank Cheryl Yao for providing support with Python syntax and library versioning that was a large help initially getting previously functioning projects and samples related to our project running on the current version of our dependencies used in this project.

References

[1] Zane Lee, "A simple Fingers Detection (or Gesture Recognition) using OpenCV and Python with background substraction" GitHub. [Online]. Available: https://github.com/lzane/Fingers-Detection-using-OpenCV-and-Python [Acessed: 11-Aug-2018]

[2] Anmol-Singh-Jaggi, “Anmol-Singh-Jaggi/Sign-Language-Recognition,” GitHub, 25-Oct-2017. [Online]. Available: https://github.com/Anmol-Singh-Jaggi/Sign-Language-Recognition. [Accessed: 11-Aug-2018].

[3] “Morphological Transformations,” OpenCV: Image Thresholding. [Online]. Available: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html. [Accessed: 11-Aug-2018].

[4] Opencv, “Failure in resizing very large image. Issue 5191 opencv/opencv,” GitHub. [Online]. Available: https://github.com/opencv/opencv/issues/5191. [Accessed: 11-Aug-2018].

[5] “OpenCV NoneType object has no attribute shape,” Stack Overflow. [Online]. Available: https://stackoverflow.com/questions/39833796/opencv-nonetype-object-has-no-attribute-shape. [Accessed: 11-Aug-2018].

[6] “OpenCV: Smoothing Images,” OpenCV: Image Thresholding. [Online]. Available: https://docs.opencv.org/3.1.0/d4/d13/tutorial_py_filtering.html. [Accessed: 11-Aug-2018].

[7] A Complete Guide to K-Nearest-Neighbors with Applications in Python and R. [Online]. Available: https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/. [Accessed: 11-Aug-2018].

[8] “sklearn.neighbors.KNeighborsClassifier,” 1.4. Support Vector Machines - scikit-learn 0.19.1 documentation. [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html. [Accessed: 11-Aug-2018].

[9] "Weekend project: sign language and static-gesture recognition using scikit-learn", freeCodeCamp, 2018. [Online]. Available: https://medium.freecodecamp.org/weekend-projects-sign-language-and-static-gesture-recognition-using-scikit-learn-60813d600e79. [Accessed: 11- Aug- 2018].

[10] "Vehicle Detection with HOG and Linear SVM - Towards Data Science", Towards Data Science, 2018. [Online]. Available: https://towardsdatascience.com/vehicles-tracking-with-hog-and-linear-svm-c9f27eaf521a. [Accessed: 11- Aug- 2018].

[11] A. Rosebrock, "Sliding Windows for Object Detection with Python and OpenCV - PyImageSearch", PyImageSearch, 2018. [Online]. Available: https://www.pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-python-and-opencv/. [Accessed: 11- Aug- 2018].

[12] A. Rosebrock, "Intersection over Union (IoU) for object detection - PyImageSearch", PyImageSearch, 2018. [Online]. Available: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/. [Accessed: 11- Aug- 2018].

[13] Malisiewicz, T. Malisiewicz and V. profile, "blazing fast nms.m (from exemplar-svm library)", Computervisionblog.com, 2018. [Online]. Available: http://www.computervisionblog.com/2011/08/blazing-fast-nmsm-from-exemplar-svm.html. [Accessed: 11- Aug- 2018].

[14] "Twins Vs. iPhone X Face ID", YouTube, 2018. [Online]. Available: https://www.youtube.com/watch?v=GFtOaupYxq4. [Accessed: 11- Aug- 2018].