Computer Vision Final Project - CS6643

Title: Sign Language Detection

From: Disha: dp3074 , Harshul: hj1393 , Omkar: opk208 , Vijay: vg1203

A. Problem Statement

In a situation where you have to interact with a person who only understands the sign language, it can be a daunting task to come up with a way to teach him the sign language at that very moment. The goal of this project is to provide an interactive, accessible learning platform for users to learn American Sign Language using computer vision. We were largely inspired by popular language learning apps like Duolingo and using the principles of computer vision and deep learning in it.This project implements a hand recognition and hand gesture recognition system using OpenCV on Python 3. A histogram based approach is used to separate out a hand from the background image. Background cancellation techniques are used to obtain optimum results. The detected hand is then processed and modelled by finding contours and convex hull to recognize finger and palm positions and dimensions. Finally, a gesture object is created from the recognized pattern which is compared to a defined gesture dictionary.

B. Proposed Solution

We are building a web application based on computer vision and tensorflow deeplearning library that allows users to use just their webcam and their hand to learn American Sign Language and receive real time feedback. The learning environment is composed of three main parts: Letters, Hand signatures and progress bars. On the right, the user is shown an image of a letter and is asked to present the sign of the alphabet using the hand. If he does it correctly, Green progress bar on the top will be updated, but if it is wrong, the red progress bar below the green one is updated. User needs to make sure he reaches the green progress bar to clear the training level.
We are using handtrack.js and classical CV methods to do so and compare both of them.

C. DataSet Description

The Sign Language Recognition dataset (https://www.kaggle.com/grassknoted/asl-alphabet) consists of images that represent the american sign language hand gesture for each of the alphabets. 27455 images are available for training and 7172 images for testing. Each image is a grayscale image of size (28,28) whose pixel values are in the range of 0-255. The dataset is robust as the same hand gesture is repeated with multiple users and against different backgrounds.

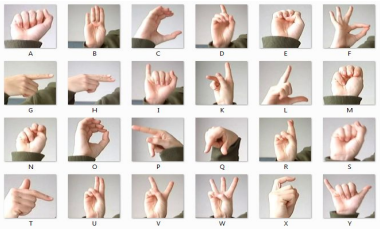


Figure 1: American Sign Language hand gesture images for each of the alphabets

D. Programming Languages and Frameworks

Python : To implement the Computer Vision Algorithms and Deep Learning Models.

Javascript : To implement the client side logics in the web application.

Flask : For the backend implementation of the web application.

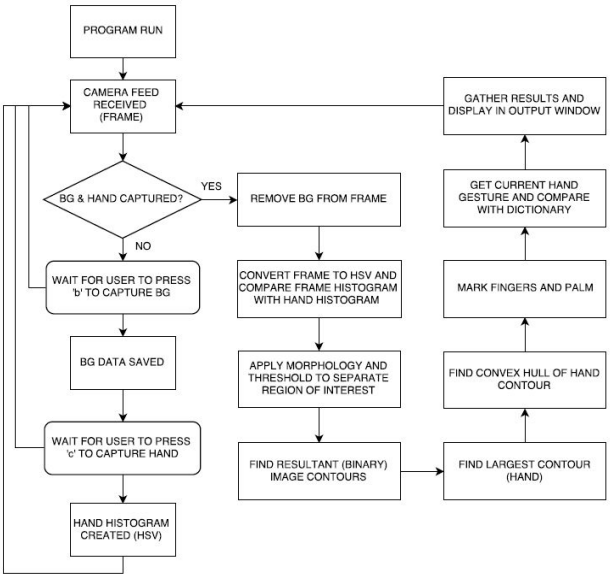
OpenCV 2.4.8 : To implement a hand segmentation algorithm.

Keras, TensorFlow, Pytorch : To train and test the deep learning models.

E. Methodologies

Below are the 2 techniques we have implemented and tested for the hand recognition:

1. **Hand tracking using Classical CV**



Running **HandRecognition.py** presents a window that acts as our output frame, currently displaying the camera feed input. **Frame** is generated as a Numpy array. Numpy library has a lot of functions for easy access and manipulation of array elements. All that, plus Numpy functions are really fast. Script is based on the background subtraction technique. First the program captures the background. To capture **background**, the user will remove his hand or any body part from the rectangle on the right side of the frame and press ‘b’ on the keyboard. Now that we have a plain background image detected, we go about capturing a **histogram** of our hand that we need the program to recognize in each frame. For this the user moves his hand to cover the 9 small boxes in such a way that all or most of his hand shades are covered in them with no air gaps or

dark shadows on it. This step is very necessary for a proper recognition. The user presses ‘c’ to capture his hand histogram. To calculate the histogram, we do the following:

a. Convert the image frame (with hand on the 9 small boxes) to HSV color space. We choose HSV because it is a better color space than RGB when it comes to detecting colors because Hue and Saturation channels are not affected by lighting and other image parameters and extracting color information is thus easier.
b. Extract the pixel values from those 9 boxes in HSV color space (our region of interest).
c. Create and return a histogram using those pixel values. To create a histogram, ‘cv2.calcHist’ function from the cv2 library is used. Once the background and hand histogram are captured, the program proceeds to an infinite while loop which can be interrupted by the user by pressing ‘q’. So that means ‘q’ exits the program. Users can also press ‘r’ to reset the program to start everything all over again.

We create a mask using the background model. This mask is then applied on the input frame, to result into a frame containing only the foreground. Now, frame contains a frame without background. Screenshot shows image before capturing background and image after bringing objects to the frame (after capturing background) Then we proceed to searching for a hand using the histogram we generated.

We define **hand_threshold** function does the following tasks:

1. Blurs the input frame to reduce noise

2. Convert frame to HSV

3. Separate out the part contained in the rectangle capture area and discard the rest

4. Calculate back projection of image using **cv2.calcBackProject** function from cv2 library. The histogram generated previously is passed as a parameter to this function call.

5. Apply **morphology and smoothing techniques** (Gaussian and Median blur) to the back projection generated.

6. Apply **threshold** to generate a binary image from the back projection. This threshold is used as a mask to separate out the hand from the rest of the frame. The back projection generated during the process is what separates out the hand from the rest of the image. Morphology and blurring is used to create a proper shape of hand which can be easily used in the next steps. Screenshot shows foreground image containing hand and a book. The result of histogram comparison leaves only the hand as a large blob.

The next step is to find out **contours** of the image generated in the previous step. We know that no comparison method is 100% efficient and so no matter how perfect the histogram is or how efficient the back projection algorithm is, there will always be false detections and noise. So we don’t just find the contour of the image but also apply some methods to remove and false detection. For this, we first find out all contours of the image and then validate the largest contour to verify if it matches the profile of a hand or not.

cv2.findContours is a function from the cv2 library used to find the contours.

We then find the convex hull of the contour using cv2.convexHull

We do not go for the more traditional method of finding convex hull defects as used by most other similar projects, because the method used in this project is more effective and useful for what we aim. Once we have the convex hull of the largest contour, we go for detecting the **center of the contour** by the following way:

a. Finds the range of coordinates that span the largest contour (extreme end points that form a rectangle that bounds the contour). x,y,w,h = cv2.boundingRect(cont)
b. In the range determined thus, run a loop that takes every point in the rectangular range and measures the distance from that point to the nearest point on the contour. 'pointPolygonTest' function does this work. This function is really slow and so we try to keep the range over which this loop runs as small and optimized as possible to reduce computation time. Note that the distance returned by pointPolygonTest is negative if the point lies outside the contour which is really good for us because that eliminates any need to check if the point lies outside or inside the contour.
c. Find the point with the largest such distance. The point is the center of the largest circle that can be inscribed inside the contour and the distance is the radius of the circle. And because the value would be negative for a point outside the contour, the point with the largest distance is definitely inside the contour. Thus, the point is the center of our hand and the radius is the dimension of our palm.
d. Check if the hand radius is large enough because a very small radius might suggest falsely detected objects in case the frame is actually kept empty. Again, a healthy condition for this project. Return palm coordinates and Radius Once we have the palm center coordinates and radius, we go for finding fingers in our hands. frame,finger,palm=mark_fingers(frame,hand_convex_hull,hand_center,han d_radius) 'mark_fingers' is a very important user defined function which does the following tasks:
e. Go counter clockwise direction and eliminate all convex hull points that are very near to each other. This helps in achieving '1 point per finger'. Otherwise the original convex hull would show about 10 points near every finger. The traditional method of finding convex hull defects does this automatically in the process but we still won't prefer that way for this project.
f. Eliminate all convex hull points too far or too near the hand center obtained in the 'mark_hand_center' function. This will leave us with just a single point per every finger that is stretched out.
g. Optimizes the results for best results and so that we don't get more than 5 fingers. What we have after this, is a palm center, radius and exactly only those many points in a convex hull that belong to our fingers. The plus point is, we have proper coordinates of each and every finger after this function. This makes it easy for us to model our hand based on the following parameters: Palm center position; Palm radius . Finger positions in the coordinate system From these parameters, we can then figure out the angles between different fingers, the count of fingers shown and ultimately the gesture defined.

From the hand recognition section, we have the following end results with us:

Palm center, Palm radius, Finger positions (and thus count as well). This project creates a gesture recognition system using a ‘Gesture’ class. The class variables and functions are defined in ‘GestureAPI.py’ Every instance of the ‘Gesture’ class has following attributes:

name – A name given to the gesture

hand_center – Center of palm

hand_radius – Radius of palm

finger_pos – Array of position of fingers

finger_count – Total number of fingers

angle – Angle between each finger and X axis Out of these, when creating a new instance, we supply name, hand_center, hand_radius and finger_pos by use of class functions. The remaining are calculated by calling class functions. ‘GestureAPI.py’ file contains a dictionary of gestures that are used when searching for a gesture found in current input frame. One example on how to define a gesture in this dictionary. A gesture with name “V” is defined. The name is later used as identifier or unique id to the gesture. Palm center is set as (475,225) with a radius of 45 pixels. This gesture, as the name suggests is a hand showing 2 fingers making a ‘V’. So we have the coordinates of those 2 fingers as (490,90) and (415,105). The finger count will be generated automatically as we pass the finger coordinates.

V.calc_angles() will calculate angles of each finger for instance V. The last line dict[V.getname()]=V will append this gesture to a dictionary ‘dict’. This dictionary is generated when ‘DefineGestures()’ function is called. So we place the following line somewhere at the beginning of our ‘HandRecognition.py’ code:

GestureDictionary=DefineGestures()

Now we have a dictionary we can use to compare our detected gesture with. Now, we need a ‘Gesture’ instance to store information about the gesture made by the hand in each input frame.

frame_gesture=Gesture("frame_gesture")

Every time the hand recognition code is run and every time we have a new coordinate for our hand and fingers, we update the attributes of this instance.

frame_gesture.set_palm(palm[0],palm[1])

frame_gesture.set_finger_pos(finger)

frame_gesture.calc_angles()

Then we run the 'DecideGesture' function defined in GestureAPI.py gesture_found=DecideGesture(frame_gesture,GestureDictionary) This will return the name of gesture detected and will return "NONE" if no suitable match is found. The 'DecideGesture' function works in the following algorithm:

1. Run a loop to compare current frame gesture passed as 1st argument in the function with every element in the dictionary passed as 2nd argument in the function.

2. While comparing, first check if the number of fingers in the current frame gesture (src1) and gesture being compared (src2) are same.

3. If the number of fingers are same and more than 1, generate an array of differences in angles made by each finger of src1 and that of src2 i.e. angle by finger 1 (src1) – angle by finger 1 (src2), angle by finger 1 (src1) – angle by finger 1 (src2) and so on. If the values turn out to be nearly constant, we can say the hand has a constant tilt and the gestures might be same. Example a "V" with hand kept straight and a "V" with a tilted hand.

4. If comparison passes the angle stage, we go for doing the same for fingers. But here we take the ratio of finger lengths instead of difference. So that we know if the hand has shrunk or grown by a constant factor. If the array thus obtained has absurd variations, it means the wrong fingers are making the correct angles and this will again reject the gesture from the list.

5. If both the stages are passed, return the gesture thus obtained.

6. If the number of fingers is 1, compare angle and length of finger to radius of hand ratio. In this case, a tilted hand with a single finger making the same gesture also gets rejected. By comparing ratios of length of finger to radius of hand, we get to know whether the correct finger is being used in the gesture. Return if match found. The name of the gesture thus detected is displayed on the output screen.

2. **Hand tracking using Handtracking.js**

Handtracking.js is the deep learning model deployed as the js library using tensorflow.js. Library works very accurately and has a significant improvement than the classical CV approach. It also provides the bounding box around the hand.

3. **Sign classification using Deep Learning model**

Using principles of deep learning, we tried different iterations of using deep learning models for detection of gestures. To handle rigid invariance in classifiers, the idea would be to use data augmentation which could help in getting over some invariance. Since by using handtrack.js we could get proper bounding boxes over hand gestures and could closely replicate the dataset, the sign-language-mnist data which was initially used consisted of images of size 28x28x1. Training and testing on this dataset on a custom CNN model was a trivial task.

F. Results :

Github repository for the demo is hosted at **: https://github.com/p-disha/Sign-Language-Trainer**

Instructions to run the application are also mentioned in README.md.

Web application results:

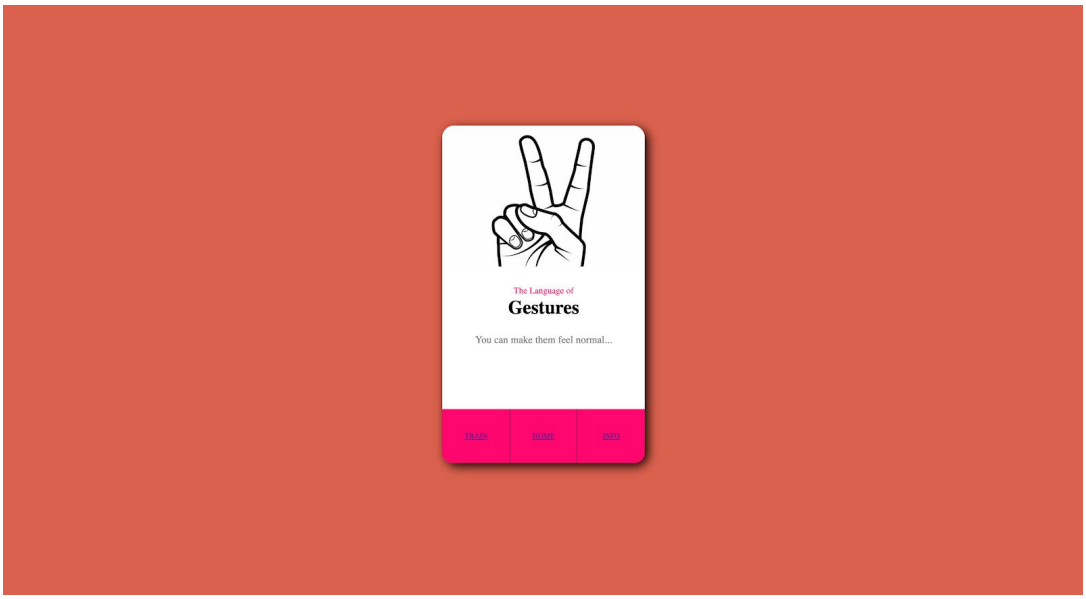


Fig 1: Web app Home Page

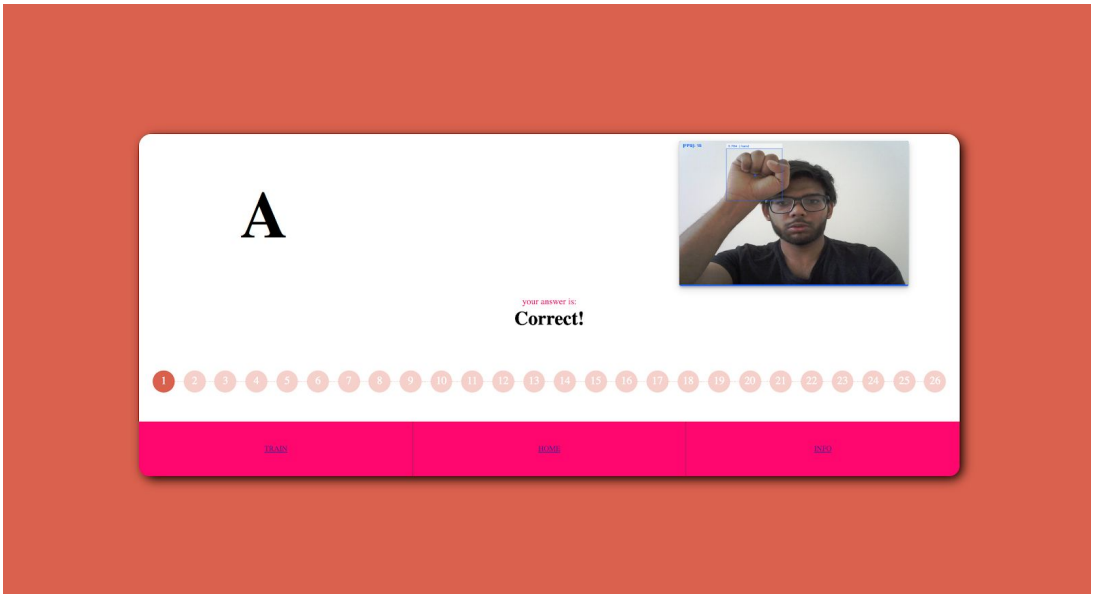


Fig 2 : Hand Tracking and sign detection using handtracking.js and Deep Learning model

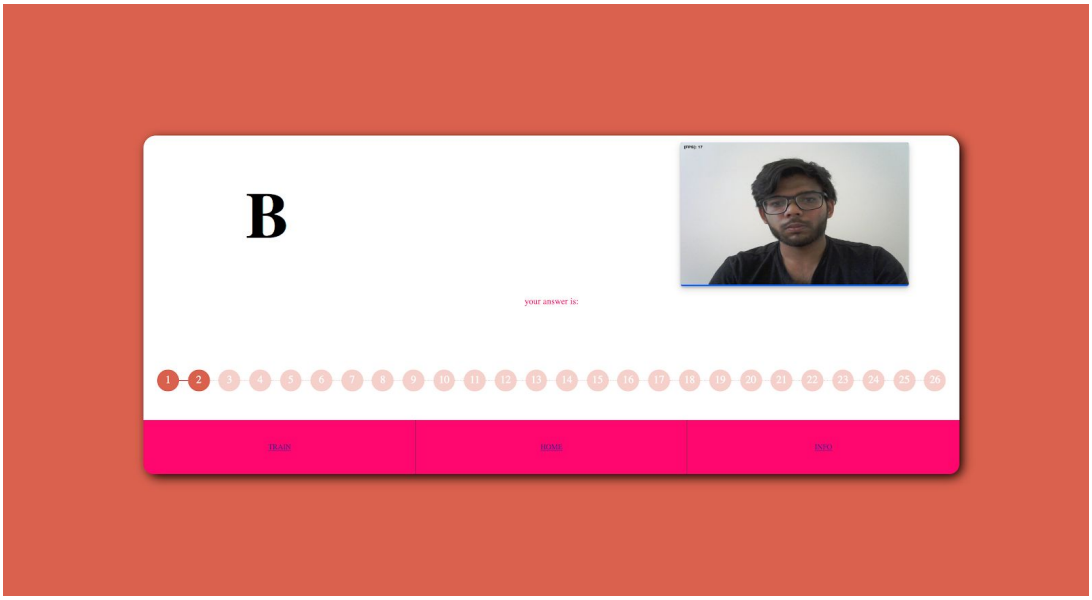


Fig 3 : Progressing to next letter when previous was successful

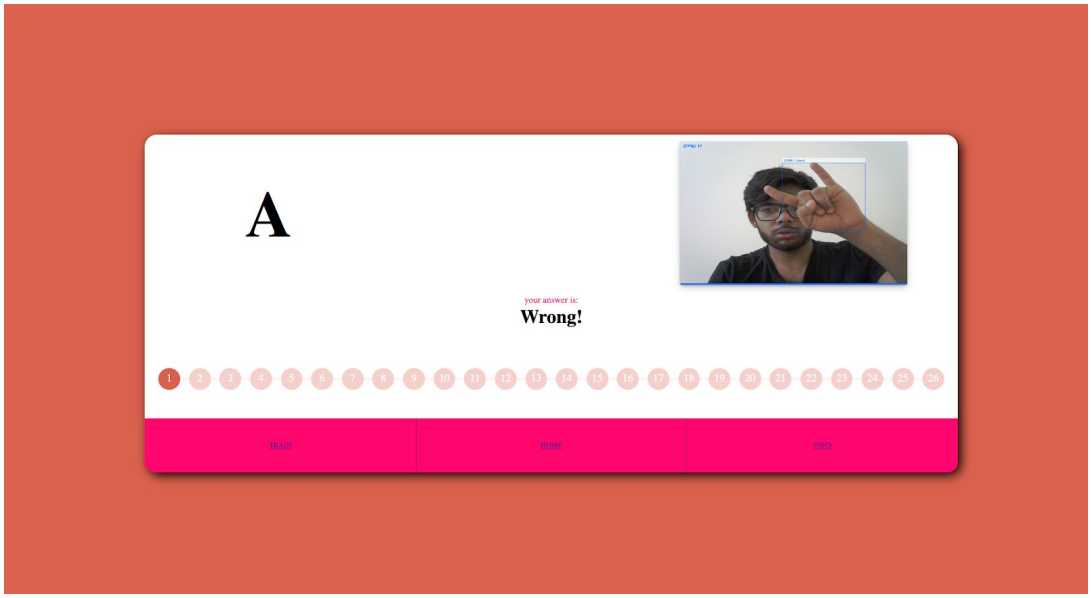


Fig 4 : No progress if the results are wrong.

Hand Tracking using classical CV results

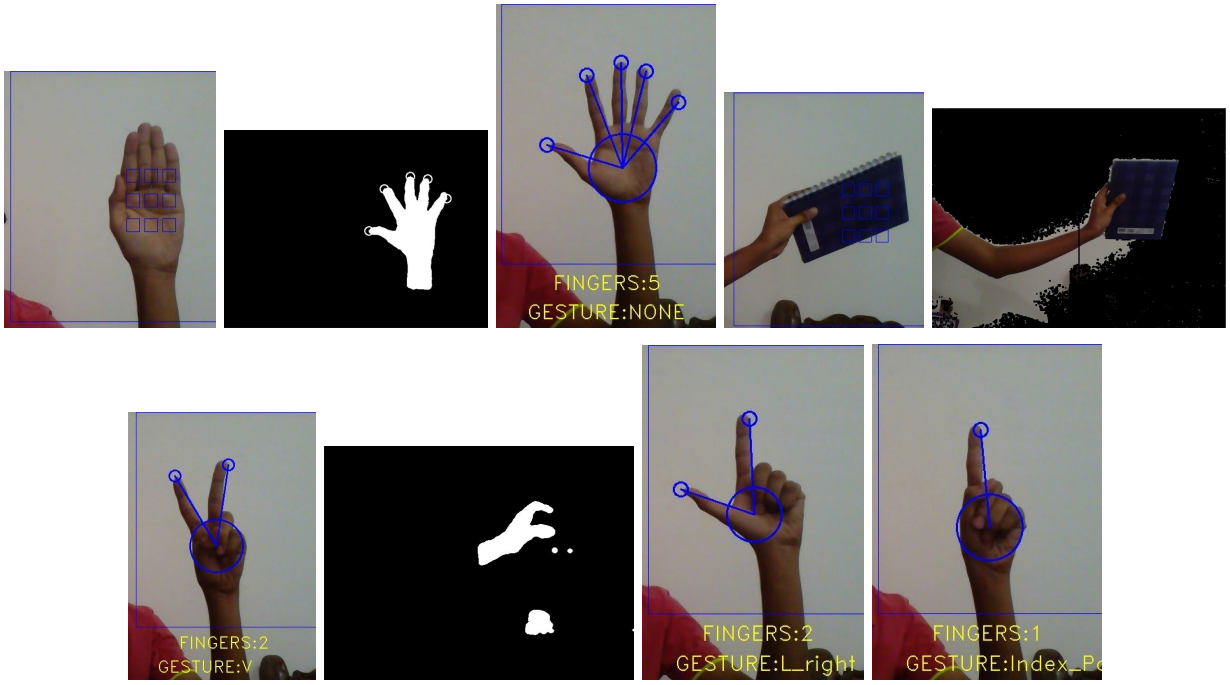


Fig 5: Hand tracking using traditional CV methods

Illustrated are some result screenshots of hand recognition and gesture being detected as required. The last image is an example of how measuring finger length ratio eliminates a gesture made with an incorrect combination of fingers. Here the difference between ratio of lengths of index to middle finger compared to that of thumb and index finger helped eliminate this false detection. The end results show an output window with the input image overlaid by the hand model generated and displays the finger count and gesture detected.

Deep Learning Model Results

G. <u>Challenges Faced:</u>

Challenges in Classical CV approach:

1. Convert the image frame (with hand on the 9 small boxes) to HSV color space. There are numerous good reasons which one can understand by searching a bit on the internet. Basically we choose HSV because it is a better color space than RGB when it comes to detecting colors because Hue and Saturation channels are not affected by lighting and other image parameters and extracting color information is thus easier.
2. Extract the pixel values from those 9 boxes in HSV color space (our region of interest).
3. Create and return a histogram using those pixel values. The above method is to determine the sin tone in the classical method of Gesture Recognition.
4. In the step is to find out contours of the image generated in the previous step. We know that no comparison method is 100% efficient and so no matter how perfect the histogram is or how efficient the back projection algorithm is, there will always be false detections and noise. So we don't just find the contour of the image but also apply some methods to remove and false detection. For this, we first find

out all contours of the image and then validate the largest contour to verify if it matches the profile of a hand or not. We've addressed more challenges and how we dealt with them in the above sections.

Challenges in Detecting Gestures:

However, the main challenge here was replicating the real world environment of different users using their webcams in different lighting conditions or different skin tones. Sign-language-mnist dataset consisted of images from different users against different backgrounds. This doesn't tell much about the coverage of the distribution, but was hugely helpful while training and to avoid bias related to skin tones to some extent.

Rigid Invariance in Classification/ Robust Classifier:

The training data images have been collected from multiple users against different backgrounds. In addition to this, data augmentation was performed to increase the robustness of training data. To create new training images, an image pipeline was used based on ImageMagick and included cropping to hands-only, gray-scaling, resizing and then creating at least 50+ variations to enlarge the quantity. The modification and expansion strategy were filters like Mitchell', 'Robidoux', 'Catrom', 'Spline', 'Hermite', along with 5% random pixelation, +/- 15% brightness/contrast, and finally 3 degrees rotation. One important reason which helps in accurately classifying different categories of sign language is the tiny size of the image (28, 28, 1) used for training and inference. Due to the small size of the image, the modification effectively alters the resolution and class separation in interesting, controllable ways.

H. Discussion on Strengths and Weaknesses:

Variables and Parameters:

Some variables are considered for optimization purposes. Also, there are variables that define certain parameters such as capture box area, threshold values, threshold for finger lengths, threshold for hand radius, etc. All the parameters are named in such a way that it would be easy to understand what they do in a function. Also, all of them are defined at the very beginning of 'HandRecognition.py' for easy access. In most cases, for tricky variables, I have placed comments near their declaration or use. Also, most variables are used in the form of ratio of something. Example, radius_thresh=0.04 is the threshold for minimum tolerable radius of hand detected. The value 0.04 implies that (0.04)x(Frame Width) will be used as a minimum threshold, to keep a relative measure of threshold compared to input frame rather than keeping it a constant value.

I. Road Ahead

1. Training a Deep Learning model to classify the signs.
2. Developing and hosting the Rest API for the deep learning model for sign classification.
3. Further development in the web application to include the sign prediction and training and testing logics.