AMERICAN UNIVERSITY IN BULGARIA

COS 491: *Senior Project I*

"Sign Language Translator"

Ksandros Apostoli

Student ID: 100116366

Author: ______________________, **Date**: __/__/_____

Signature

Supervisor: ______________________, **Date**: __/__/____

Signature

Blagoevgrad, 2017

Table of Contents

Title: "*Sign Language Translator"*

Title (in Bulgarian): "*Сигн Лангуаге Транслатор*"

Author: Ksandros Apostoli

Keywords: ASL, SVM, HOG, Machine Learning, Image Detection, Image Recognition, Classifier, Support Vector Machine, Python, CVXOPT

Abstract:

The purpose of this project is to study and observe the use of Multiclass Support Vector Machine classifiers for designing and implementing a software application capable of translating real-time, static, sign language gestures into letters. Targeting at people with speech or hearing impairments, this project aims to provide an example of the potential use of machine learning classifiers in the translation of the American Sign Language (ASL) gestures into alphabetic characters. Translating characters will finally result in building words and furthermore sentences. Such an outcome, might serve as a good means of communication between the targeted audience and general public in meetings or conferences, which frequently does not understand sign alphabets. In addition, requiring only a web-camera for capturing the desired gestures, the application can be used in any environment that can interpret the Python language. To achieve the desired results, the user must first "train" the software, with real samples of each character. For each gesture the user shows to the application, he/she receives a character under the video frame. It is important to underscore that within the scope and purpose of this project, only the five first alphabetic characters were tested, considered enough to demonstrate the functionality of the solution, and its robustness. Further training can be done for letters of the alphabet, without need of additional development. However, this training as described later in the report, will be time consuming and computationally demanding.

I. Introduction

1.1 Overview of the Project

The project aims to build a reliable technology for the digital translation of sign languages using machine learning classifiers. American Sign Language was the sign language chosen as a base for the implementation of the project as it is the most popular hand gesture language in English. However, the application can be trained with any other format of sign gestures in different languages. The potential extension and real-life implementation of the results in this project, offer a very efficient, economical, and robust solution for a well-known problem to the restricted audience domain of users with speech or hearing disabilities.

Resembling a standalone application, the project is presented to the user through a windows forms interface. As it will be described later in detail, the amount of direct physical interaction between the user and the computer is limited to the phase of “teaching” or “training” the translator, since in any other case of use the main tools required for the communication between the computer and the user are the camera and the desired gestures in front of it.

At launch, the software offers two choices: to train or to test the translator. During the first run, when no previous training has occurred, training is mandatory. Training consists in physically “showing” to the translator hand gestures for each alphabetic character that the user wants to be translated during the training phase. Once all desired gestures accompanied with the character they resemble are shown to the translator, the user is given the option to complete the training and proceed to testing phase. If previous training has occurred, i.e. it is not the first time the application is running, the user can proceed directly to the testing phase, since previous training data, already processed help the translator do its job.

Testing is the second option that the user is given in case the application has been already trained in some past execution. During testing, the whole interaction user-application is based on the camera input, without the need of using any other input, unless he or she needs to clear the output translated. During testing, translation is done by asking the user to present his or her hand gesture to a highlighted rectangle on the screen. Once the user has placed his hand inside the marked area, the application automatically detects the presence of an object, and in case the object shown matches to any of the characters that the application is trained to predict, the output appears under the camera frame. Usually this whole process happens within seconds.

1.2 **Personal Purpose for the Project**

As a Computer Science undergraduate, my interest in finding methods that incorporate new technologies into means which benefit society has played a vital role in my range of present and future interests. The purpose of the project is to study and observe the potential use of machine learning methods in the implementation of technologies that promote a better life quality for general users, and in particular for users who belong to disadvantaged minorities. As far as I am concerned, software solutions that emphasize the support towards such groups must be vastly promoted when it can help these users regain part of the basic human needs they have lost due to their disability. Most of fully capable individuals nowadays, do not speak or show interest in learning any sign languages, because they simply do not need to do so. This results in an increased difficulty in fulfilling the basic need of communication for people with speech and hearing impairments. Hence, by paying attention to the development of software technologies that can alleviate inequities like these in everyday life, a better life quality can be promoted.

In addition to pursuing this interest of mine, through this project I was given the chance of studying relatively new technologies such as Support Vector Machine classifiers, which are examined in detail, and basic image processing, that has been also touched for a good part of the implementation.

1.3 **Scope**

In accordance to the above stated purpose, this project aims to build a fully working software solution that will serve to research, understand, and demonstrate the potential of SVM-s and image processing in the development of software that supports people with disabilities. Rather than presenting a finalized version of the application, which would take a much greater amount of time, and would require higher computational resources, the project build a core sample of the working solution, which can be easily extended without further need of software development.

Taking this into consideration, the application the application will be expected to run correctly and completely for a limited number of alphabetic characters, and for an allowed maximum size of samples for the training of each gesture. This will allow the software to run quickly, and require less training, helping this way fulfill the purpose of this project, and respecting the time span available for its completion. The number of learned hand gestures by the end of the project will be 5 (five), and these will be the first characters “*A*”, “B”, “*C*”, “*D*” and “*E*” of the alphabet. For each character, around 30 - 45 training samples will be used, so the training will be relatively fast. A higher number of letters, would result in a much higher number of samples, which would contribute as result to a much higher time of training.

1.4 Target Demographics

The project aims to find application in a demographic group of any age, gender, and nationality. The only “requirement”, if I can call it so, for the application is that the users can sign in American Sign Language. These people on the other side, are most often those who have lost hearing and/or speech, and who need to communicate frequently with a general public throughout their daily routines. Therefore, we would restrict our target demographics to:

〉 People with speech/hearing impairments
〉 Signers of the American Sign Language
〉 In particular, signers who would like to communicate efficiently in front of a mixed audience consisting of people who understand ASL and people who do not; without the need of typing to a computer or hiring a translator.

1.5 Technologies Implemented

The software development of the application uses Python as its language. Anaconda, a Windows based version of Python, is used to make installation of packages easier on Windows.

The following list of packages is required in Python:

1.5.1 NumPy

Used for operations and manipulations of N-dimensional arrays, and also for more linear algebra based requirements.

1.5.2 OpenCV

Necessary for implementing the computer vision interface, i.e. accept camera input. Responsible for the main image processing functionality through its methods.

1.5.3 CVXOPT

Package that performs the optimization, crucial for building a Support Vector Machine. To achieve classification through the Support Vector Machine, we need to maximize the separation between our classes. This is already a maximization problem, which we reduce into a quadratic programming problem. CVXOPT provides solvers for quadratic problems, performing the algebraic calculations.

1.5.4 Tkinter

Python package that brings Windows Forms into life through Python.

These packages, as described above, we use to build the two main components of our application:

- **Support Vector Machine** (Machine Learning Classifier), from now and on referred to as **SVM**

- **Histogram of Oriented Gradients** (Feature Extractor), from now and on referred to as **HOG**

We will examine both these in detail in later sectors.

II. Application Domain Problems

2.1 Functional Requirements

- At launch user is presented to a menu that allow him/her to choose between "Training" or "Testing" the translator.
- When the user chooses "Train" he/she is redirected to the training window.
- The training window, must allow the user to choose from a dropdown list the alphabetic character which corresponds to the gestures that are going to be stored as samples.
- The user is shown on the bottom of the window, under the frame, which character he is storing samples for.
- The training window must contain a video stream showing camera captured frames in real time.
- In the video frame, a rectangular area must be highlighted, where the user should place his/her hand to capture the sample.
- If an object is detected on this selected area of the video input, the user must be indicated by a text on the top-left of the window, showing "Hand Presence: True", or otherwise if no object has entered the area "Hand Presence: False"
- If the user has his/her hand inside the highlighted area, and if he presses "*C*" from the keyboard, then the frame with the hand gesture, or object in the area is to be stored as a sample, classified by the character choice of the user.
- Any time a sample is stored, the user is to be notified about this activity and to be shown the name of the sample stored.
- If "Test" is chosen in the launch menu, the user is to be taken to the testing window.
- The testing window contains a live stream of the video captured by the camera with the same rectangular area as the trainer.
- When the user places his hand in the marked area, without shaping any valid gesture, nothing is to happen.
- If the user places his hand in the marked area with a valid gesture, the corresponding translation is to be shown under the video frame and stay there, so words can be formed by consecutive letters.
- The user can clear this output are using a "Clear" button.
- If one letter is predicted, the same letter cannot be predicted again unless user moves his hand out of the testing area and replaces it inside.

2.2 Non-Functional Requirements:

- **Reliability**

  i. The application stores all samples in a separate folder, where it cannot be lost or deleted. Even if training model is lost, a new model can be built using these samples.
  ii. The application must predict correct labels for the trained gestures at a success rate of 90%.
  iii. A wait period of 1-2 seconds is applied between two predictions, so no random predictions occur when switching handshape from one gesture to another.
  iv. To avoid repeated predictions, the same letter cannot be predicted two times consecutively without removing the hand from the testing area first.

- **Performance**

  i. Trained gestures must not take more than 3-4 seconds to be translated.
  ii. With 5 classes of gestures, and at most 128 samples for each pair of classes, training must take up to 10 seconds to complete, storing the SVM model onto a file.
  iii. Samples are stored in less than one second.

- **Usability**

  i. User friendly and easy to use interface.
  ii. User mostly interacts with the computer using gestures.

- **Implementation** (Prerequisite Technologies for the Environment)

  i. Python (2.7)
  ii. NumPy
  iii. CVXOPT
  iv. OpenCV
  v. Tkinter

- **System Requirements**

  i. 256 MB of RAM memory (based on consumption while testing)
  ii. Any OS that can interpret Python
  iii. Web Camera

- **Operating Constraints**

    i. Shiny / Overcrowded Background Surfaces behind user must be avoided.

2.3 User Cases

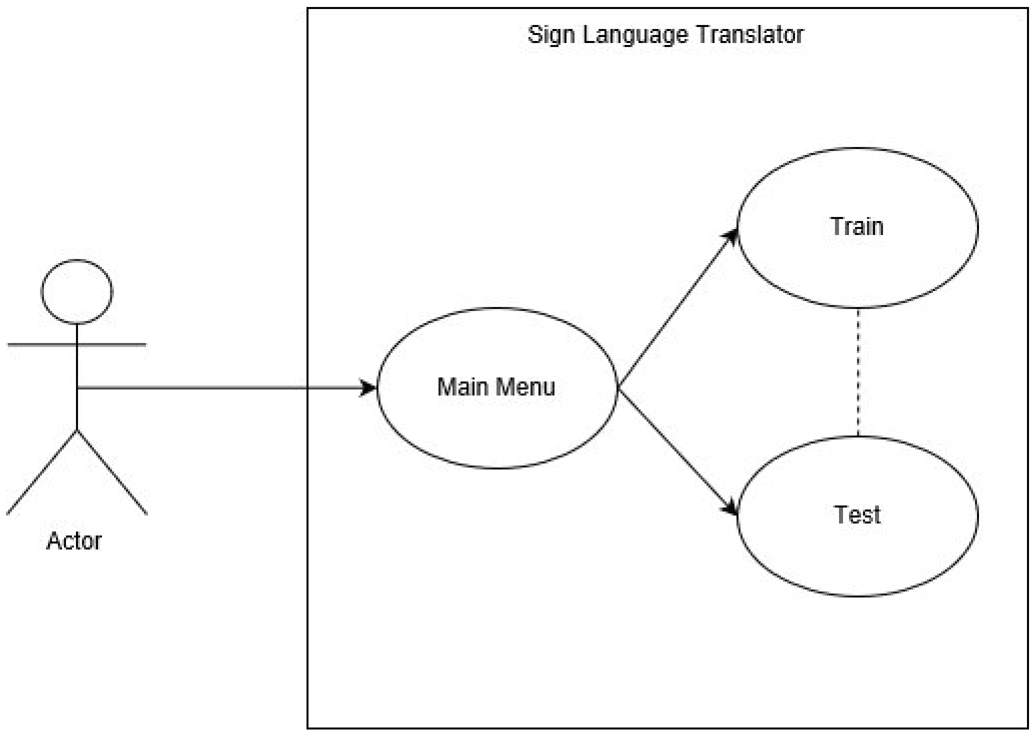


Figure 1 | Top User Case Diagram – The very basic outline of the application

2.3.1 Top-Level Use case

The figure above summarizes the top-level use case decomposition of the application. We can identify from there two core functionalities mapped into use cases. To select between the two options available, that are interrelated to each other, the user is at first introduced with a simple menu, offering him to select his desired activity for that use of the translator. The first option, is to train the translator. Training is mandatory, if it is the first time of usage, or if no training has been completed previously. The training is basically the part where the user "teaches" the computer the new sign language helping it build a predictive model for the testing phase. Testing on the other side, which is the second option in the home menu, is basically the component which fulfills the purpose of the application, i.e. the translation of any previously trained sign language. We will explore each of the two cases in separate use cases below.

2.3.2 Training Use Case

As mentioned earlier, before the user can proceed to the actual translation of their sign language, the application must be trained or taught to understand that set of gestures. If this has not happened, i.e. no training model has been built (we will see what the training model is later), the user cannot proceed to the testing/translating phase.

During training there are three main options available:

- ❖ Choose a letter to train the translator for,
- ❖ Capture the gesture corresponding to the selected character,
- ❖ Complete training.

When training starts, the character “A” is chosen by default. The user can proceed training for that letter, or he/she can select different letter from a dropdown list. After selecting the desired letter, the user can capture as many samples as the want for that character. To do so the user needs to do two things: first place his/her hand in the bounded area on the screen, shaping the correct character, and second when he/she has clearly shaped the hand gesture inside the box, press “C” from the keyboard, which stands for “Capture”. Once completed sampling the selected letter, the user can either complete training, or continue sampling by choosing a different letter. When user has register all samples he/she wants the application will build a new model or extend the existing one based on that data. The user will be automatically redirected to testing.

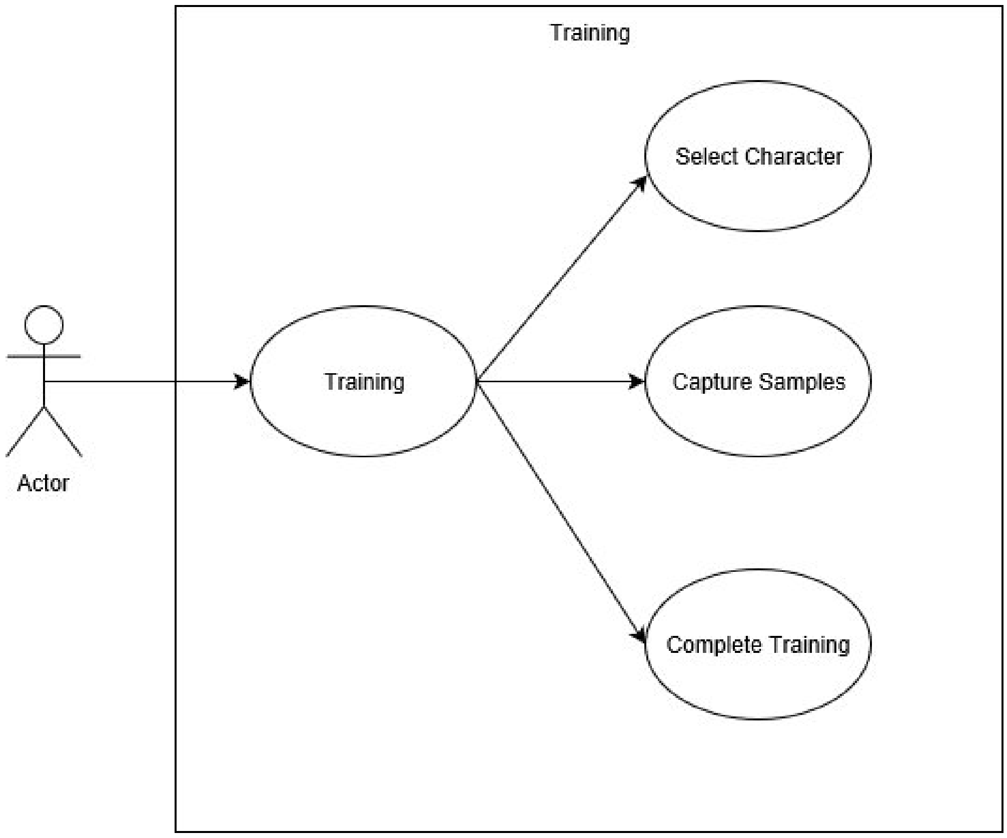


Figure 2 | Training Use Case

Use Case: Train		
Actor:	*User*	
Entry condition:	- *User Selects Training in the home menu*	
Exit condition:	- *User selects "Complete Training" or closes the window*	
Flow of events	*Actor*	*System*
	1. User chooses "Training"	*1.1 System exits the home menu window and continues to the training subsystem, creating a new training window* *1.2 Default selected character for training is "A"*
	2. User Selects letter for training.	*2.1 System notifies user that the chosen character is selected for training.*
	3. User captures hand gesture sample by pressing "C".	*3.1 System checks if user has placed his hand in the bounded area* *3.2 System triggers sample storage by the triggered event of pressed key "C".* *3.3 System notifies the user that about his stored sample displaying its name.*
	4. User repeats as many times as desired.	*4.1 See* ***3.***
	5. User presses "Complete training".	*5.1 System build training model and redirects user to translating window.*
Exception conditions:	- *If there is no hand detected in the marked area, the sample will not be stored.*	
Special Requirements:	- *Preferred background is basic and unicolored.*	

Table 2 | Training Use Case Table

2.3.3 Testing/Translating Use Case

For a user to enter the testing/translating phase, previous training must have taken place before. If not, the "Translate" button in the home menu is disabled and user cannot access it. Once the user enters the translation phase, the main interaction between user and computer is via the Web Camera and his gestures. The user has two main choices in this use case:

- Translate a gesture,
- Clear the translation prompt.

During the first activity, which is the main one, the user must place his hand again in the marked area on the live video streaming frame, on his window. In case the user has created a valid ASL gesture with his/her hand, the corresponding character will appear on the translation prompt. If the user wants to translate the same letter twice, consecutively, he/she has to take his hand out of the recognition area and replace it inside. For differing consecutive gestures, this is not required. Once the user has created his word using his gestures, he might need to clear the translation prompt. To do that, a clear button is at the bottom of the window, letting the user refresh the prompt and start over. Below we can see the use case table and sequential diagram:

Use Case: Translate		
Actor:	*User*	
Entry condition:	- *User Selects "Translation" in the home menu*	
Exit condition:	- *User closes the window*	
Flow of events	*Actor*	*System*
	1. User chooses "Translation"	*1.3 System exits the home menu window and continues to the testing subsystem, creating a new testing window*
	2. User places hand gesture in the marked area.	*2.2 System predicts the gesture and outputs the result in the translation prompt*
	3. User Clears the Prompt	*3.4 System empties the output prompt.*
Special Requirements:	- *Preferred background is basic and unicolored.*	

Table 2 | Testing/Translating Use Case Table

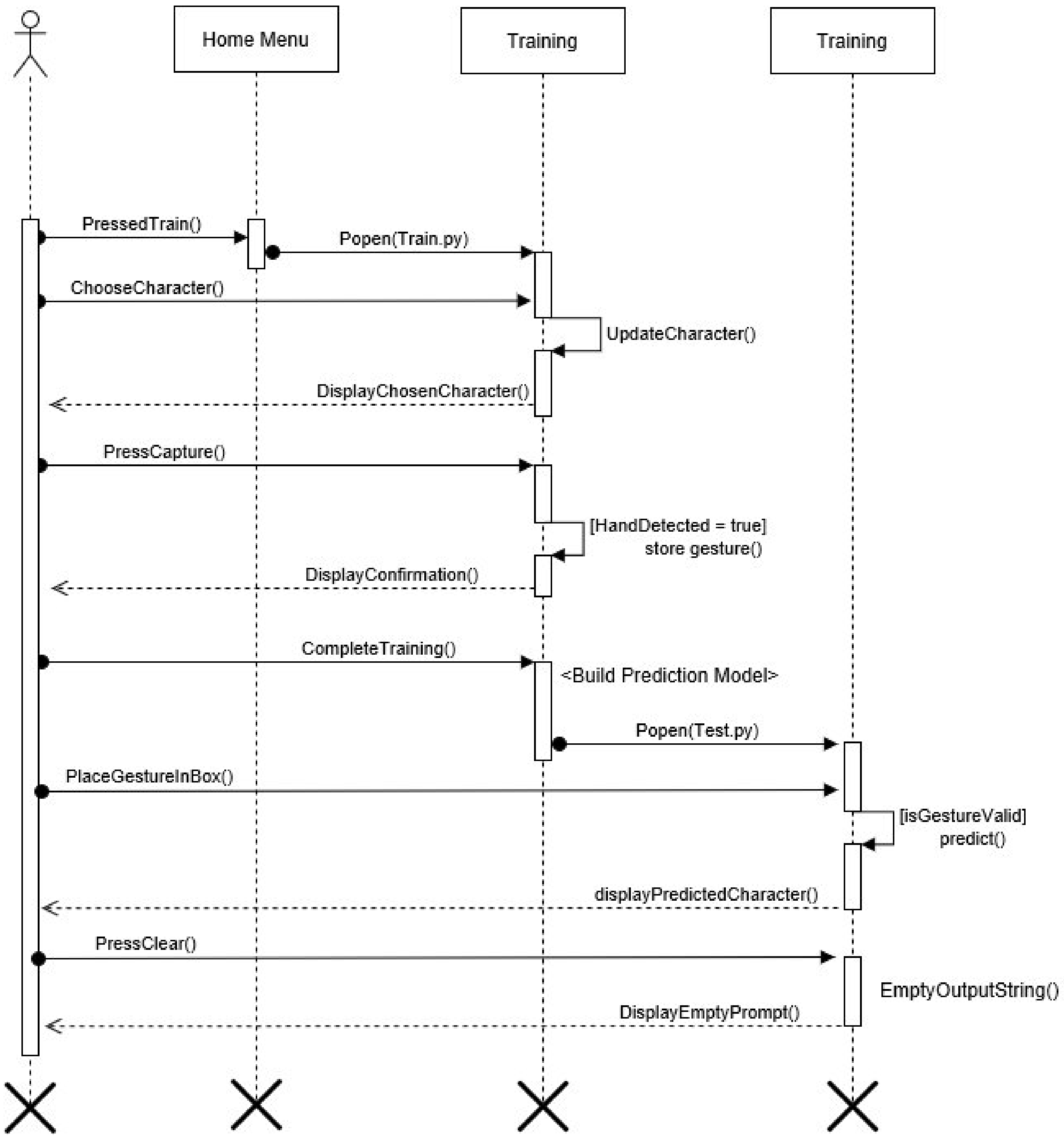


Figure 3 | Sequential Use Case Diagram – Training / Testing

III. Software Problem Analysis and Design

3.1 Overview of Design Approaches and Algorithms

Before proceeding into the software solution for this project, it is important to get a deeper understanding of the problems related to it. Translating or predicting hand gestures, as we are going to mention from now and on, might seem to be a trivial task, however the solution for building such a system is of complex nature from both the algorithmic and implementational prospective. In this section, I will explain, or try to depict the algorithms applied and a mathematical regression, necessary for building a good understanding of them. We start with classification, to continue later with image processing and feature extraction.

3.2 Classification and the Support Vector Machine

The main goal for the project as mentioned multiple times earlier, is to build a software solution capable of translating hand gestures from the American Sign Language, into characters of the English alphabet. To understand how we are going to accomplish such a prediction, I will first start studying the case of only one letter. Once we have an understanding and insight on how the solution works for a single letter, we will see how easily we can extend this algorithm into **n-letter** cases, or, as we will later denote them, *classes*. To understand the solution to the single-letter prediction problem we need to visualize what Support Vector Machines or **SVM**-s, as we are going to call them from now, work.

3.2.1 Two-Class SVM Problem

Let us reduce our hand gesture prediction problem into a much easier problem to visualize. Assume we want to build an algorithm, that will allow us to *classify* two dimensional points into two categories: those located above and those located below the line with the following equation in the ordinary two-dimensional plane:

$$y = \mathrm{x}$$

or equivalently,

$$\mathrm{y} - \mathrm{x} = 0$$

In this case, all we must do is plug in the coordinates of the point, and analyze the sign of the following function in two variables:

$$\mathrm{f(x,y)} = \mathrm{y} - \mathrm{x}$$

For instance, if we want to check the point *(-4,2)* we simply evaluate $f(-4,2)$ and check its sign. If that sign is *negative*, it means that our point is under the lie. Otherwise, if the value of $f(x,y)$ at *(-4,2)* is positive, we can infer that our point is located above the line. In this particular case:

$$f(-4,2) = (2) - (-4) = 6$$

Then, we have:

$$sign(6) = 1,$$

so, our point (-4,2) is located above the line $y = x$. The illustration below helps visualize that.

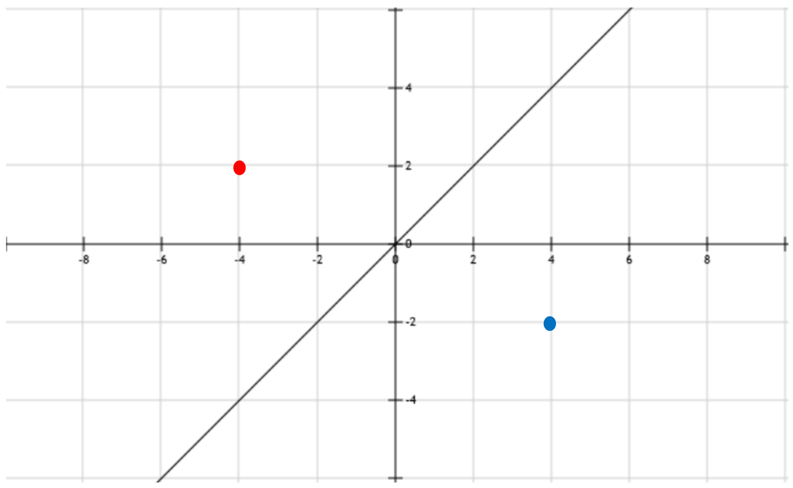


Figure 4 | Point Classification in 2 dimensions

In fact, what we just did is what we refer to as *classification*. By using the function of a line that separates two *classes*, we can find the class where any random point belongs to, by simply plugging its coordinates into the line equation, and observing the output's sign.

Now, assume that we have two sets of 2-D points, each representing a class. If we can find the equation of a line separating the two sets from each other, then we can follow the exact same procedure as described above to classify any new point among these two classes. Indeed, this is easy to apply for 3 dimensions as well. It is equally trivial to think about 3-dimensional points being separated in two classes. This time however, a line will not suffice to make the division of the 3-dimensional space into two. In this case, we would need a 2-dimensional plane to classify points in 3 coordinates. See the picture below:

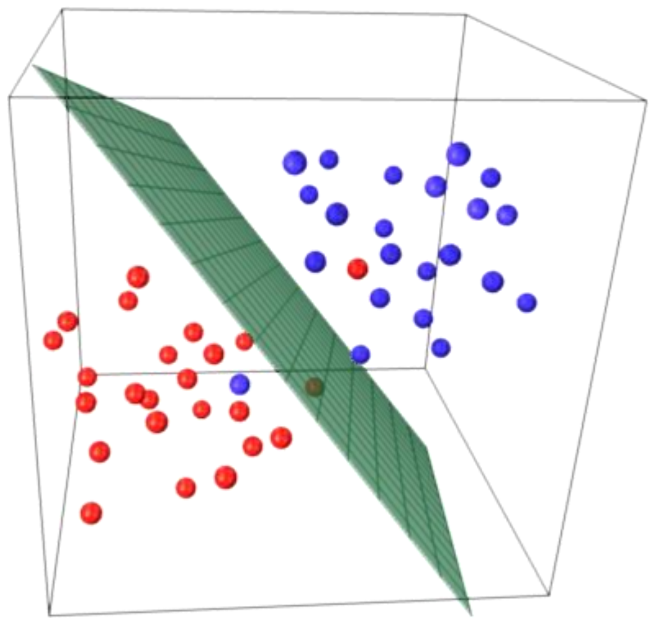


Figure 5 | Point Classification in 3 dimensions

The procedure described above, for classifying points of two and three-dimensional spaces, can be obviously generalized into n-dimensions. For any two sets of n-dimensional points and an (n-1)-dimensional hyperplane (*hyperplanes are simply n-dimensional planes*) separating those two sets, we can always predict the class in which any other n-dimensional point belongs to, by simply checking the sign of the value we get by replacing its coordinates into the equation of the hyperplane. Classification after knowing the equation of that hyperplane must be simple and quick then, simply a matter of algebraic operations. Hence, the real challenge, is to come up with a precise way to build that hyperplane, so it can be later used for classification of new instances or points.

So, our problem is now reduced to the computation of the correct equation for an (n-1)-dimensional hyperplane that will clearly separate two sets of n-dimensional points already given to us. In fact, we are not only looking for a hyperplane that does the job and separates the two sets of points, but we are looking for the optimal hyperplane that does this. Simply put, we need to find the equation of the hyperplane that maintains the greatest distance from the closest points of each of the two sets to it, knowing only the coordinates of all points on each set. But how can we do that?

Before we answer this question, I need to clarify that there are two ways for solving this task. The first one, is merely brute-force algorithmic, i.e. makes maximal use of computer resources, to loop through the points of each set, and find the most efficient hyperplane. In this project I did not make use of this way for two reasons. First, I think that this method does not help in providing a better insight of the problem, which is one of the purposes of this project. Second, the approach selected for this project is much more resource and time efficient.

To tackle the problem of finding an optimal (n-1)-dimensional hyperplane for separating two sets of n-dimensional sets, in this project I followed an analytical approach. This analytical method, transforms our task to an optimization problem. To explain this, let us paraphrase the problem in a slightly different manner.

We know now that for any two-class problem, it is enough to find the equation of the hyperplane dividing two sets and plug in the coordinates of any point of that dimension whose class we want to determine. Any hyperplane in n-dimensions, is characterized by two values: its normal vector $\boldsymbol{w}$ and its shift b. Hence our class determining function would look like this:

$$f(x_i) = \begin{cases} 1, & if \quad \boldsymbol{w^T} x + b > 0 \\ -1, & if \quad \boldsymbol{w^T x} + \boldsymbol{b} < 0 \end{cases}$$

All we need to find up to this point is an equation for the optimal hyperplane dividing our two sets of points. These points, call them x_i, we can tuple with their corresponding *class* or *labels*, in the shape (x_i, y_i), where y_i represents the class of the point x_i and $y_i \in \{-1,1\}$, for i starting from 0 up to the total number of our points:

$$D = \{(x_i, y_i) \mid i = 1, \ldots, m, x_i \in \mathbb{R}^n, y_i \in \{1, -1\}\}$$

The equation of what we need to compute based on these tuples, i.e. the equation of the optimal hyperplane dividing the two classes has the following shape:

$$a_0 * x_0 + a_1 * x_1 + \cdots .. + a_n * x_n + b \ (\mathbf{i})$$

or equivalently,

$$(a_0, a_1, \ldots\ldots, a_n) * \begin{pmatrix} x_0 \\ \vdots \\ x_n \end{pmatrix} + b = \boldsymbol{w}^T \boldsymbol{x} + b \ (\mathbf{ii})$$

From the dot product represented in this equation we need to compute $\vec{\boldsymbol{w}}$ and $\mathbf{b}$ so the hyperplane has the greatest distance from each set, and we are done. Below is an illustration of a hyperplane (in red) that would be optimal among many separating hyperplanes for our 2-D case:

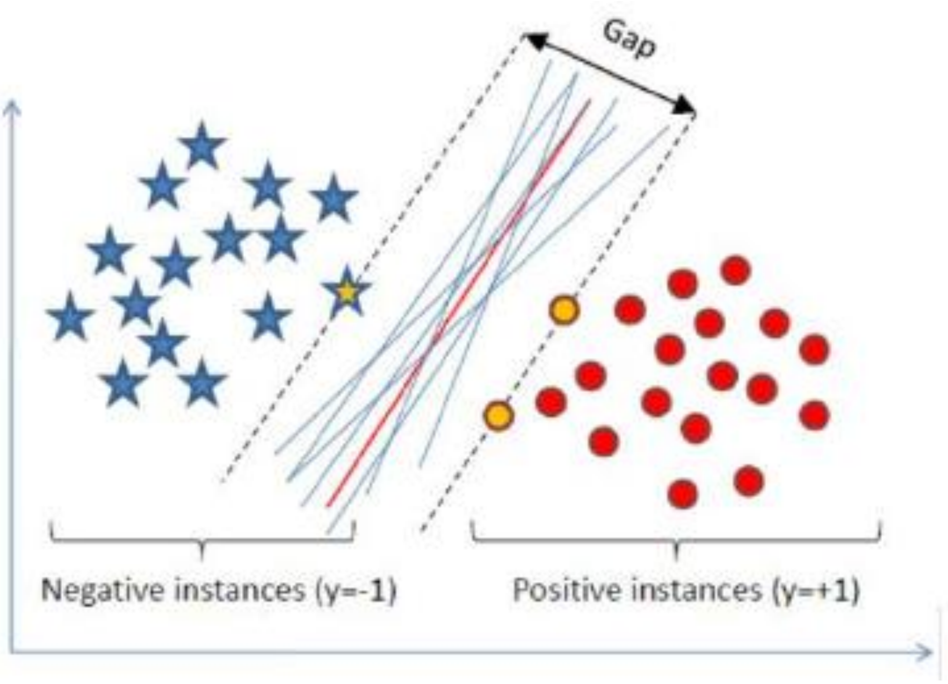


Figure 6 | Optimal Hyperplane

From an algebraic viewpoint, in equation *(ii)*, $\vec{\boldsymbol{w}}$ would be the normal vector defining our hyperplane, while $\boldsymbol{b}$ would be the shift, just like a line, but in n-dimensions. If we want to find the hyperplane which optimizes the *margin* between the two sets, then we are looking for that hyperplane which has maximal distance from all points that are closer to it in any of the two sets. Therefore, we need to somehow compute the distance between an (n-1)-dimensional hyperplane and an n-dimensional point. Once again, we proceed just like we do in the 2-D plane. We know that the distance between a line with equation $ax + by + c = 0$ and a point (x_0, y_0) is:

$$\mathbf{distance}(ax + by + c = 0, (x_0, y_0)) = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}.$$

The same applies to n-dimensions, for a point expressed as a vector x_i and a hyperplane with normal $\vec{\boldsymbol{w}}$ and shift $\boldsymbol{b}$. In this case we get:

$$distance(\vec{\boldsymbol{w}}^T x ++ b, \vec{\boldsymbol{x_i}}) = \frac{\left|\langle \overrightarrow{\boldsymbol{w}}, \overrightarrow{\boldsymbol{x}} \rangle + b\right|}{\|\vec{\boldsymbol{w}}\|}$$

To maximize the distance of the hyperplane from the whole set, we need to disregard any other points of each set apart from those which are closer in distance to that plane. These points are called Support Vectors. In figure 6 Support Vectors, are represented by (★) and (●).

Say these points have a distance *d* from the hyperplane. All points x_n which are not *support vectors* will have a distance $d(\boldsymbol{w}^T\boldsymbol{x}+\boldsymbol{b},\boldsymbol{x_n}) < \boldsymbol{d}$ from the hyperplane**.** Therefore, the two hyperplanes parallel to our optimal hyperplane, which contain support vectors from the two classes, will create a separating margin between our hyperplane and the sets, equal to *2*d*. Our goal now has changed one more time: to find the optimal hyperplane, we must optimize this margin. The two marginal hyperplanes will have the following equations:

$$W_0\text{:}\,\boldsymbol{w}^T x + b = d$$

$$W_1\text{:}\,\boldsymbol{w}^T x + b = -d$$

While in the normalized form:

$$W_0\text{:}\,\boldsymbol{w}^T x + b = 1$$

$$W_1\text{:}\,\boldsymbol{w}^T x + b = -1$$

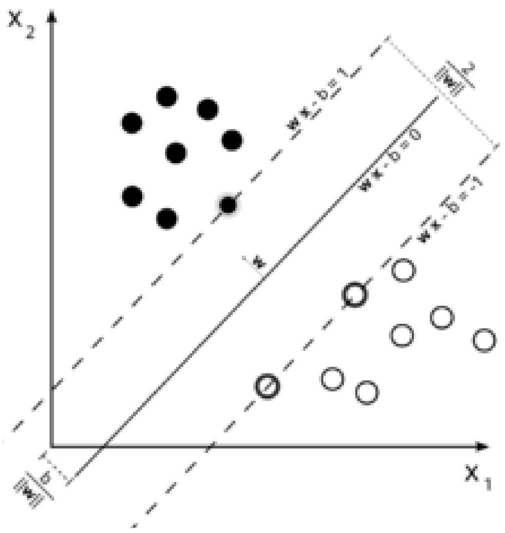


Figure 7 | Margin of Hyperplane

Recall now that *d* is the marginal separation distance, or the distance between our separating hyperplane and the closest point to it. Furthermore, the distance between W_0 *and* W_1 can be calculated by:

$$\frac{2}{||\boldsymbol{w}||}$$

since the distance between W and W_0 is $\frac{|w\hat{}Tx+b|}{||w||} = \frac{1}{||w||}$ (same for W_1).

To find the optimal separating hyperplane, we need to maximize this distance hence, minimize $||\boldsymbol{w}||$. In addition, we want that any point x_i belonging to a class $y_i, y \in -1{,}1$ to lie outside of our margin, so:

$$\boldsymbol{w}^T x_i + b \geq +1 \quad if \quad y_i = +1$$

$$\boldsymbol{w}^T x_i + b \leq -1 \quad if \quad y_i = -1$$

We can combine the two conditions of this inequality to get a new condition:

$$y_i(\boldsymbol{w}^T x) \geq 1$$

We can now state our maximization problem with constraints explicitly:

$$\underline{\textit{Minimize}}\ \|\boldsymbol{w}\|\ ,\ \textit{s.t.}\ y_i(\boldsymbol{w}^T x + b) = 1$$

That can be rewritten as (Math ∩ Computer Science, 2017)*:*

$$\underline{\text{Min}}\ f{:}\tfrac{1}{2}\|w^2\|\ ,$$

such that,

$$g(x_i) = [y_i(\boldsymbol{w}^T x_i + b) - 1] = 0, \forall i \in \{0, \dots, number\ of points\}$$

This is a constraint maximization (minimization) problem, and we already now a very well-established solution to it, *Lagrange Multipliers*. Obviously, we are going to have as many constraints as many points we have.

3.2.1.1 Lagrange Multipliers

Lagrange method is used for maximizing or minimizing a general function f(x,y,z) subject to constraints (or side conditions) of the form $g_i(x,y,z) = k$.

Assumptions made (Stewart, 2014):

- the extreme values exist i.e. $\nabla g \neq 0$
- Then there are numbers $\boldsymbol{\lambda_i}$ such that $\nabla f\ (x_0, y_0, z_0) = \Sigma \lambda_i\ \nabla g_i\ (x_0, y_0, z_0)$ and $\boldsymbol{\lambda}$ are called the Lagrange multipliers.

In our case all $\lambda_i \neq 0$ will correspond to support vectors!

Let us know see how our margin maximization problem can be written according to Lagrange. In general, using Lagrange multipliers to optimize a function we have:

$$L(x, \lambda) = f(x) - \sum_i \lambda_i * g_i(x)$$

In our case, we have:

$$f(x){:}\frac{1}{2} * \|\mathbf{w}^2\|,$$

$$g_i(x){:}\, y_i(\boldsymbol{w}^T x_i + b) - 1 = 0$$

Therefore, our Lagrange problem will look like:

$$min\, L = \frac{1}{2} * \|\mathbf{w}^2\| \; - \sum_i \lambda_i * [y_i(\boldsymbol{w}^T x_i + b) - 1]$$
$$= \; \frac{1}{2} * \|\mathbf{w}^2\| \; - \sum_i \lambda_i * y_i(\boldsymbol{w}^T x_i + b) \; + \sum_i \lambda_i$$

This formulation is precisely what will solve our problem. For any $\lambda_i \neq 0$ we choose our *support vectors* and therefore we can build our hyperplane. However, this solution offers a Lagrangian solution with respect to two variables: $\boldsymbol{w}$ and b while we can only deal with one at a time. For this reason, unless we find a way to transform our problem, we won't be able to get results.

To proceed further and overcome this difficulty, we switch our problem to what is called a *Dual Problem*. The property of derivatives being 0 at minima gives us the following equalities:

$$\frac{\partial L}{\partial \boldsymbol{w}} \; = \; \boldsymbol{w} \; - \; \sum\nolimits_{\boldsymbol{i}} \boldsymbol{\lambda_i y_i x_i} \; = \; \boldsymbol{0}$$

$$\frac{\partial L}{\partial b} = \; \sum\nolimits_i \lambda_i y_i \; = \; 0$$

Therefore,

$$\boldsymbol{w} \; = \; \sum\nolimits_{\boldsymbol{i}} \boldsymbol{\lambda_i y_i x_i}$$

$$\textbf{and}$$

$$\sum\nolimits_i \lambda_i y_i \; = \; 0$$

Now, by substituting the above equalities into our initial Lagrange form, we get:

$$min\, L = \frac{1}{2} * \|\mathbf{w}^2\| - \sum_i \lambda_i y_i(\boldsymbol{w}^T x_i + b) + \sum_i \lambda_i$$
$$= \frac{1}{2} \sum\nolimits_{\boldsymbol{i,j}} \boldsymbol{\lambda_i y_i \lambda_j y_j (x_j x_i)} - \sum_i \lambda_i y_i \left(\sum\nolimits_{\boldsymbol{j}} \boldsymbol{\lambda_j y_j x_j \, x_i} \right) + \sum_i \lambda_i$$
$$= \sum_i \lambda_i - \frac{1}{2} \sum\nolimits_{i,j} \lambda_i y_i \boldsymbol{\lambda_j y_j (x_j x_i)}$$

$$such\ that,$$

$$\sum_i \lambda_i y_i = 0 \quad and \quad \lambda_i \geq 0 \;\; \forall i \; \in \{0, \ldots, \#\, of\ points\}$$

Now that we have,

$$L(\lambda) = \sum_i \lambda_i - \frac{1}{2}\sum_{i,j} \lambda_i y_i \boldsymbol{\lambda_j y_j}(\boldsymbol{x_j x_i})$$

and we know $y_i,\ x_i\ \forall i \in \{0, \dots, \#\, number\ of\ points\}$, we can finally substitute, differentiate with respect to λ, and have it equal to 0 so we can get a solution for λ_i:

$\sum \lambda_i y_i = 0, for\ 0 \leq \lambda_i \leq C$, where C is our tolerance.

Note that this tolerance gives us valid results even when points are near or inside the margin.

Solving for λ_i in the above equation, we can now select our support vectors and build the hyperplane.

The **Support Vector Machine** used for the translation of hand gestures in this project follows the exact logical path as above. The reason why this works well with our data is because it is mostly *linearly separable*, i.e. no points on one set can be find in between points of the other. However, in many SVM classifiers, points are *linearly inseparable*. To solve this, we use kernels, or functions which map our points into higher dimensions, making their separation possible. We will not put emphasis on kernels here, but the below visualization might help in creating a perception on how they work

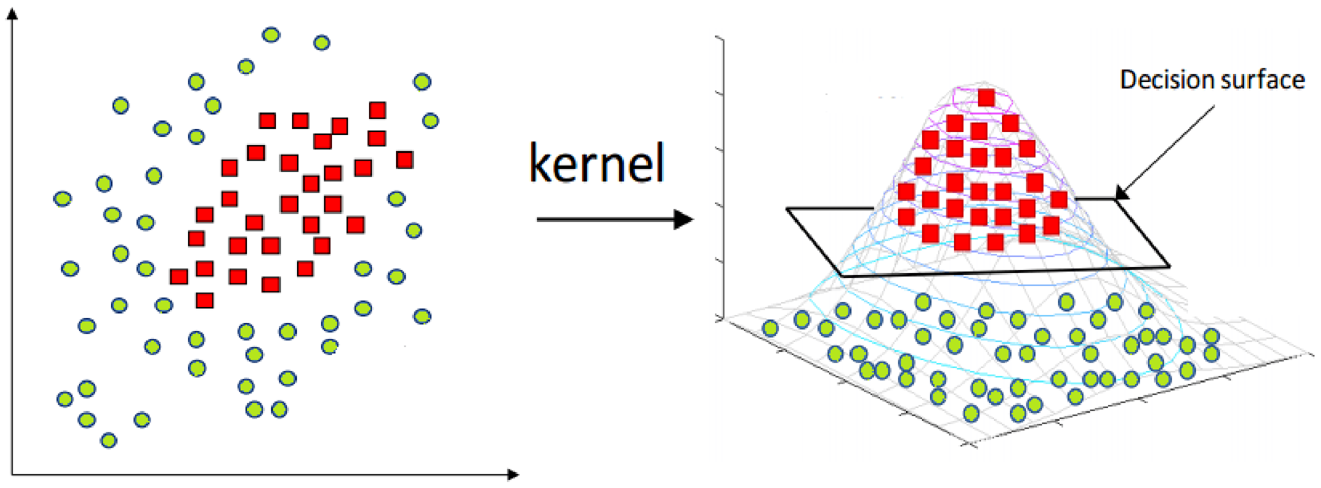


Figure 8 | Separation of Data, using kernels

All we have to do now, is consider one set of points to be the set of gestures corresponding to the character "A", and the other set to represent points of hand gestures with no meaning which we define by the label "O". Our classifier of one letter would then be complete, capable of predicting only "A"-gestures.

3.2.2 Multiclass Classification

Support Vector Machines were explained above, and hopefully understood in detail for the classification of only two classes. Now, for our problem domain, there are technically 26 letters, which means 26 classes to classify. What to we do in this case? To tackle this problem we have to extend our SVM to what is called a Multiclass or Multilabel Classifier. For such classifiers, there are two primary approaches to be usually followed: One-vs-One and One-vs-All. In this project, I chose to implement the One-vs-One approach, for reasons I will discuss below. Before going there however, let me briefly describe both ways, and their advantages or disadvantages. I will consider the case of three classes, since the algorithm for any number *n* of classes above 2 is the same.

3.2.2.1 One-vs-All

Assume we have a working SVM that can be trained, and predict correctly samples for two classes. However, we would like to be able to get correct predictions for three classes at the same time, call them A, B, and C. Now, if we consider one set made of samples belonging to class A and another set consisting on all samples for B and C together, we have successfully come up with two classes which can be passed into our SVM for training and prediction. So, basically, we will have one SVM model trained for A vs. (BC), another model trained for B vs. (AC) and finally one more model trained for C vs. (AB). Checking which of the three classes wins against the rest, we can predict the class of our testing sample.

Algorithmically:

```
classes=[A_Samples ,B_Samples,C_Samples]

SVM_Models [];

//One-Vs-All Training

for class in classes

      SVM_Models[class.label] = SVM.train(class, classes.remove(class))


//One-Vs-All Predict

for label in classes

      if SVM_Models[label.label].predict(testingSample) == 1

return label.label
```

The One-Vs-All approach has the main advantage of speed. The number of times the SVM model has to be trained, is equal to the number of classes we want to predict, therefore just *O*(N). However, One-Vs-All approaches, make our SVM model less robust and precise. The reason it does so, is that when you mix samples of many classes under the same training set, this training set will be very widespread and distributed in a much larger space than the single class you put against it. As a result, the sets will become highly inseparable by linear hyperplanes, reducing the precision and reliability of the classifier overall.

3.2.2.2 One-Vs-One

Examining the construction of a multiclass SVM using the One-Vs-All approach will assist us in understanding the One-Vs-One method as well. This time, to decide among several classes for our prediction, we consider all possible combinations of pairs of classes. Going back to our previous example, with A, B, and C as classes in the case of a One-Vs-One classifier, we would have to train a model for A vs B, another for A vs C and a third model for B vs C. Training a model for each possible pair of classes, we can later pass our testing sample through each model separately. Each time a class will win over its competitor in the prediction, we will add 1 vote to it. When testing has been completed for all pairs of classes, the class with the greatest number of votes will become the final prediction. The pseudocode for this approach would look like the following:

```
classes=[A_Samples ,B_Samples,C_Samples]
labels=[A, B, C]; votes[]
SVM_Models []
//One-Vs-One Training
for i=0; i<classes.length; i++
      for j=i; j<classes.length; j++
          SVM_Models[labels[i];labels[j]] = SVM.train(class[i],class[j])


//One-Vs-One Predict
for model in SVM_Models
      if SVM_Models[label.label].predict(testingSample) == 1
            votes[model.key[0]]++
      else votes[model.key[1]]++
return max(votes).key()
```

The One-Vs-One approach for multiclass classification, is in some way the opposite of One-Vs-All. I say that, keeping in mind that, in One-Vs-One, the training time will take much longer, as there are $\frac{N*(N-1)}{2}$ pairs of classes to be trained. However, the precision of this approach is also higher, considering that two classes are far more likely to be linearly separable, making prediction much easier and more robust. Using really well-separable features to train our SVM as we will discover in a minute, I chose this approach for the implementation of the Multiclass problem in this project, so I could save this nice characteristic.

Now that we have a better insight on the whole process and algorithmic route of Support Vector Machines, I need to clarify that for the design of this whole system I use aggregation. SVM has its own class in the solution, and MultiClass SVM has a separate class as well. To maintain low coupling, readability, extensibility, and separation of concerns, the SVM class is injected inside MCSVM (Multiclass SVM) class. This way, the top-level hierarchy of the application uses MSVM as an interface to the SVM.

Once a perception of SVM and MCSVM is presented, it is time to describe how we can use this classifier with our data, which, after all, are simply frames of hand gestures. In the final minimization problem with Lagrange multipliers, all that was required to build a model i.e. an optimal separating hyperplane of the two classes, was a set of points for each class, together with their describing labels. Thus, the next section explains how images or frames of gestures can be converted into points that preserve features necessary for classification.

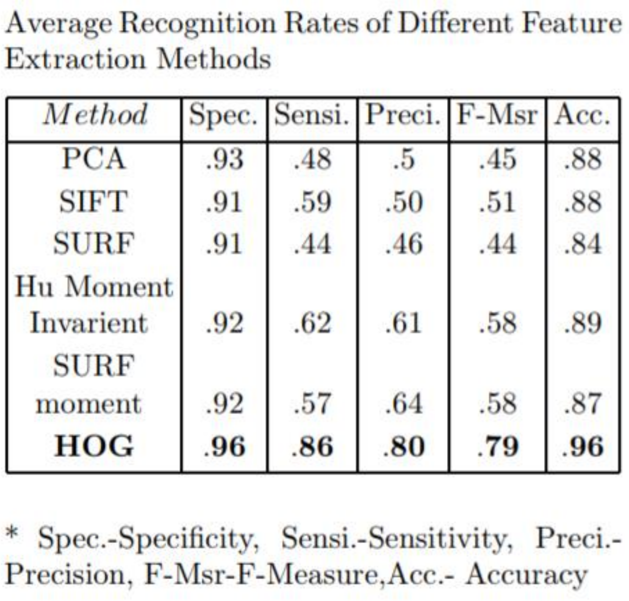
3.3 Feature Extraction and Image Processing

Now that we know what we need to classify hand gestures into corresponding characters in ASL, namely n-dimensional points, we shall proceed in finding a method for using hand gestures in front of a camera as an input, and finally converting them into n-dimensional points or n-dimensional arrays. All this process will be scrutinized in this section, which is divided into two parts, both related to image processing. The second part will serve the purpose of explaining how a human hand image will be extracted from a video frame, and processed to become ready for what we call Feature Extraction, and will be the focus of the first part. Let us start now in with the first part.

3.3.1 Feature Extraction Overview

Before making use of our, now, well-defined classifier, we need first to find a way of turning image frames into n-dimensional arrays, capable of preserving features necessary for classification. A simple way to go, is through mapping the *m x n* RGB matrix that represents the frame into a *1 x m*n* dimensional vector. This method, is used in several occasions with SVM and results were reasonable. However, light alterations, shadows or other environmental conditions that affect the picture were highly correlated to the success of the classification and prediction. Leave that apart, *1 x n*m* are big sized vectors, which would be demanding in both time and computer resources.

A good part of image processing nowadays is dedicated to what we call *Feature Extraction*. This name, which is pretty self-explanatory, refers to the process of extracting descriptive features of image data through performing first some image processing for “normalization” of externalities such as light exposure, background etc. This process not only provides a very good filtering for image characteristics, but is also very efficient, through reducing semantically the size of its output descriptor, which most of the time happens to be exactly what we need: an n-dimensional array of features. Few of the most used feature extractors used at present are Canny Edge Detection, Scale Invariant Feature Transformation, and Histogram of Oriented Gradients. Before choosing any of them, a little research was needed in order to find the most effective one in extracting features of hand gestures. Based on an article I found in the “*International Journal of Information Processing”,* written by Indian authors Viswanathana and Idicula, Histogram Oriented Gradient was the most efficient descriptor with a rate of specificity marking 96% in hand gesture recognition, higher than any other extractor. The table below is extracted from the same paper.


Average Recognition Rates of Different Feature Extraction Methods

Method	Spec.	Sensi.	Preci.	F-Msr	Acc.
PCA	.93	.48	.5	.45	.88
SIFT	.91	.59	.50	.51	.88
SURF	.91	.44	.46	.44	.84
Hu Moment Invarient	.92	.62	.61	.58	.89
SURF moment	.92	.57	.64	.58	.87
HOG	**.96**	**.86**	**.80**	**.79**	**.96**

* Spec.-Specificity, Sensi.-Sensitivity, Preci.-Precision, F-Msr-F-Measure,Acc.- Accuracy

Table 3 |Average Recognition Rates of Different Feature Extraction Methods (Viswanathana , Idicula 2015)

We will now continue explaining the Histogram of Oriented Gradients and its importance in this project.

3.3.1 Histogram of Oriented Gradients (HOG)

Histogram of Oriented Gradients, is a method for feature extraction which offers to us the 1-dimensional arrays of frame-features we need for our SVM classifier to work. The HOG descriptor technique counts occurrences of gradient orientation in localized portions of an image - detection window, or region of interest (intel).

To begin with, one needs to start thinking of image frames as *n x m x 3* dimensional arrays where *n* and *m* corresponds to the size of the image. The last component of the array shape, that is, 3, represents the RGB colors for each pixel. For applying HOG feature extraction, we need to get rid of that number, by converting our image to grayscale. Once we do that, we also apply some threshold, to normalize pixel values, and make sure they do not exceed some limits. Now we are ready to start with the HOG feature extraction algorithm found at Intel® Integrated Performance Primitives for Intel® Architecture Developer Reference. Volume 2: Image Processing:

1. Divide the image into small connected regions called cells, and for each cell compute a histogram of gradient directions or edge orientations for the pixels within the cell.
2. Discretize each cell into angular bins according to the gradient orientation.
3. Each cell's pixel contributes weighted gradient to its corresponding angular bin.
4. Groups of adjacent cells are considered as spatial regions called blocks. The grouping of cells into a block is the basis for grouping and normalization of histograms.
5. Normalized group of histograms represents the block histogram. The set of these block histograms represents the descriptor.

To better understand the algorithm, intel provides also an accompanying visual representation:

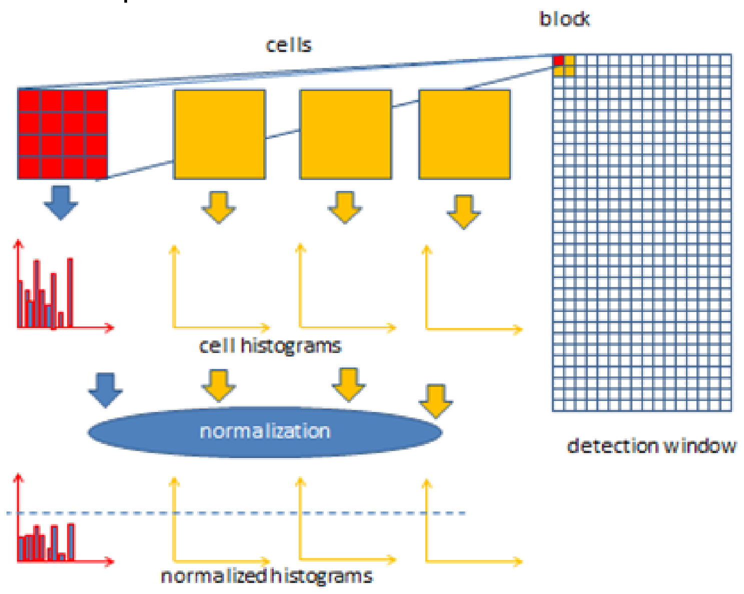


Figure 9 |

Histogram of Oriented Gradients

(Primitives for Intel® Architecture Developer Reference,2017)

During my research on feature extraction, I also decide to try and assemble an image out of HOG extracted features, for a sample of those stored in my Train Data folder. The image below is a collection of all orientation bins and their magnitudes, over a real sample image. Basically, this is how our application sees the world. The file for doing this can be found and tested with any picture. Its name is HOGVisual.py.

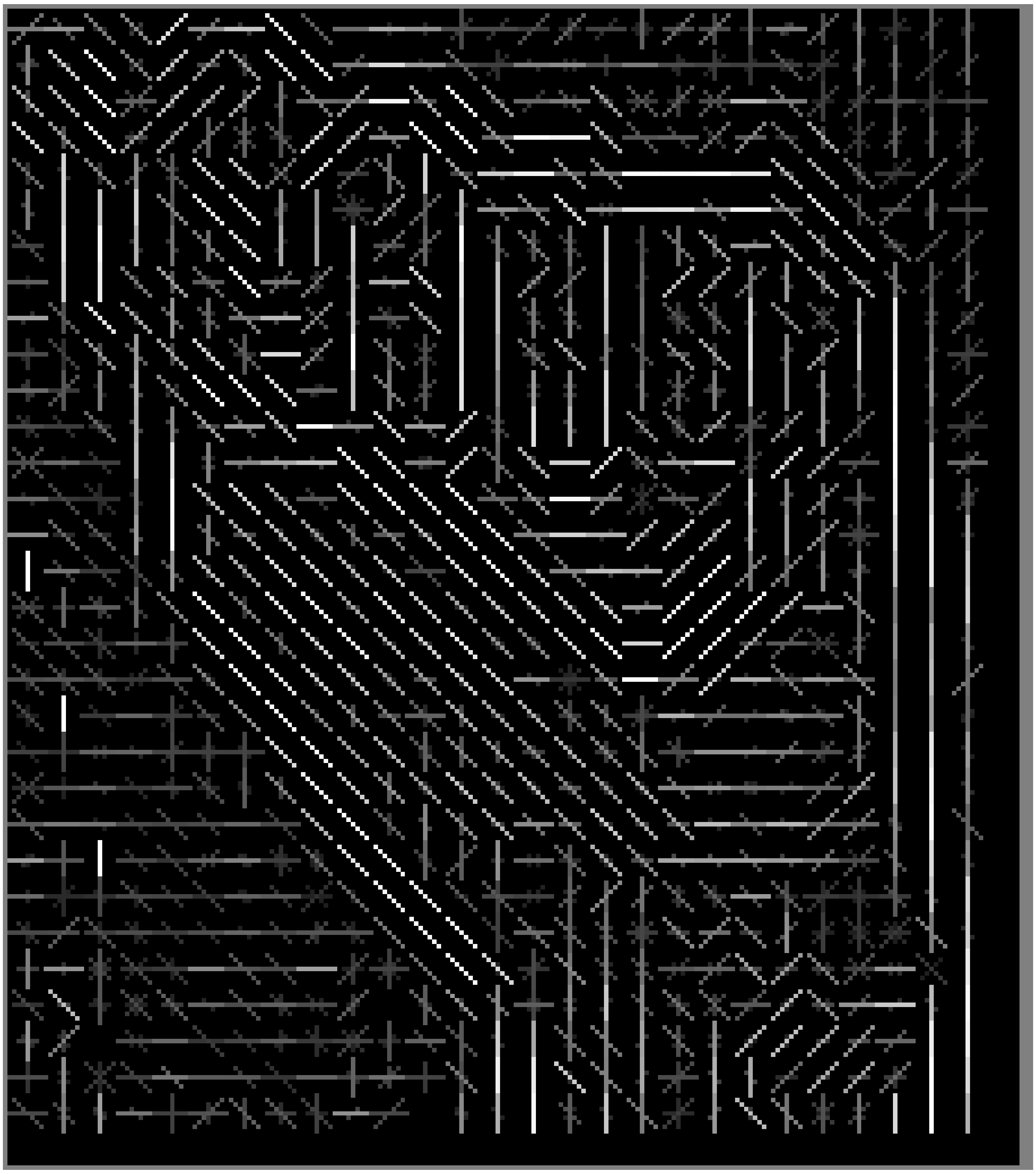


Figure 10 | HOGVisual output for hand

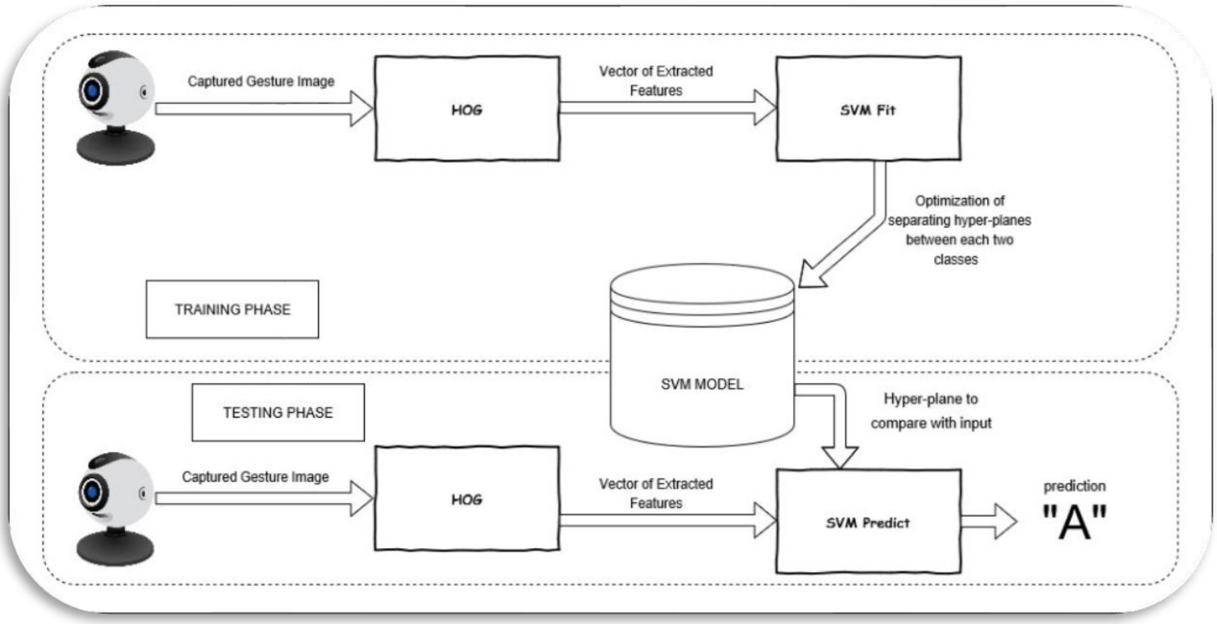
3.3.2 Basic Image Processing and Detection

What is left for the Image Processing component to be complete, is detection of hand. This task we complete by using Contours. Contours are a method of detecting objects in a video frame, using frame and pixel difference, providing a surrounding box over the detected object. Minor differences in a video frame might occur regularly, accounting for changes in light, shadows etc. Object detection is crucial part of the application, since it is the trigger of hand gesture classification. That means that if no object detection occurs, or if it occurs incorrectly, the hand gesture cannot be translated at all, or it could translate wrong. To avoid false detection based on these systematic pixel changes, we need to restrict our bounding contours of detection to qualify as valid only above a minimal surface. In this case, our image detection will be less vulnerable to become corrupt.

3.4 Component Based Architecture

Having a logical division of the software design into two functional parts, namely *Training* and *Testing* the architecture needs also to adapt to that nature and therefore decompose the implementational part into logical components that represent well-defined interfaces that involve methods (**HOG**), events (**Contours and Hand Detection techniques**) and properties (**SVM model**). Hence, the most suitable architecture to use for our problem domain, is a *Component Based Architecture.* By doing so, we can clearly abstract two main components corresponding to the two functional needs of Training and Testing. In addition, modules from both these components, grouped together, give us the third component of User Interface. Choosing such an architecture, our solution puts emphasis on several software design principles. First, we respect Separation of Concerns Principles, in the sense that the Training and Testing component, can be individually updated without affecting each other. Next, coupling is low, considering that both components communicate through each other through the *SVM model* interface, therefore, as long as the trainer builds this model correctly for the tester to consume it, there is no need to worry about altering each other when one changes. High cohesion is ensured to, since the main goal of both components remains the translation of the sign language gestures. Last to be mentioned is extensibility. If one will need to add another feature extractor different from HOG for instance, this can be done very quickly through implementing the extractor in the utilities of the application, and asking the SVM to read those new features instead of HOG features. Below, the diagram for this architecture is illustrated alongside with a more detailed UML Diagram for specific modules for the components.

3.5 System Component Diagrams



Captured Gesture Image
HOG
Vector of Extracted Features
SVM Fit
Optimization of separating hyper-planes between each two classes
TRAINING PHASE
SVM MODEL
TESTING PHASE
Hyper-plane to compare with input
Captured Gesture Image
HOG
Vector of Extracted Features
SVM Predict
prediction
"A"

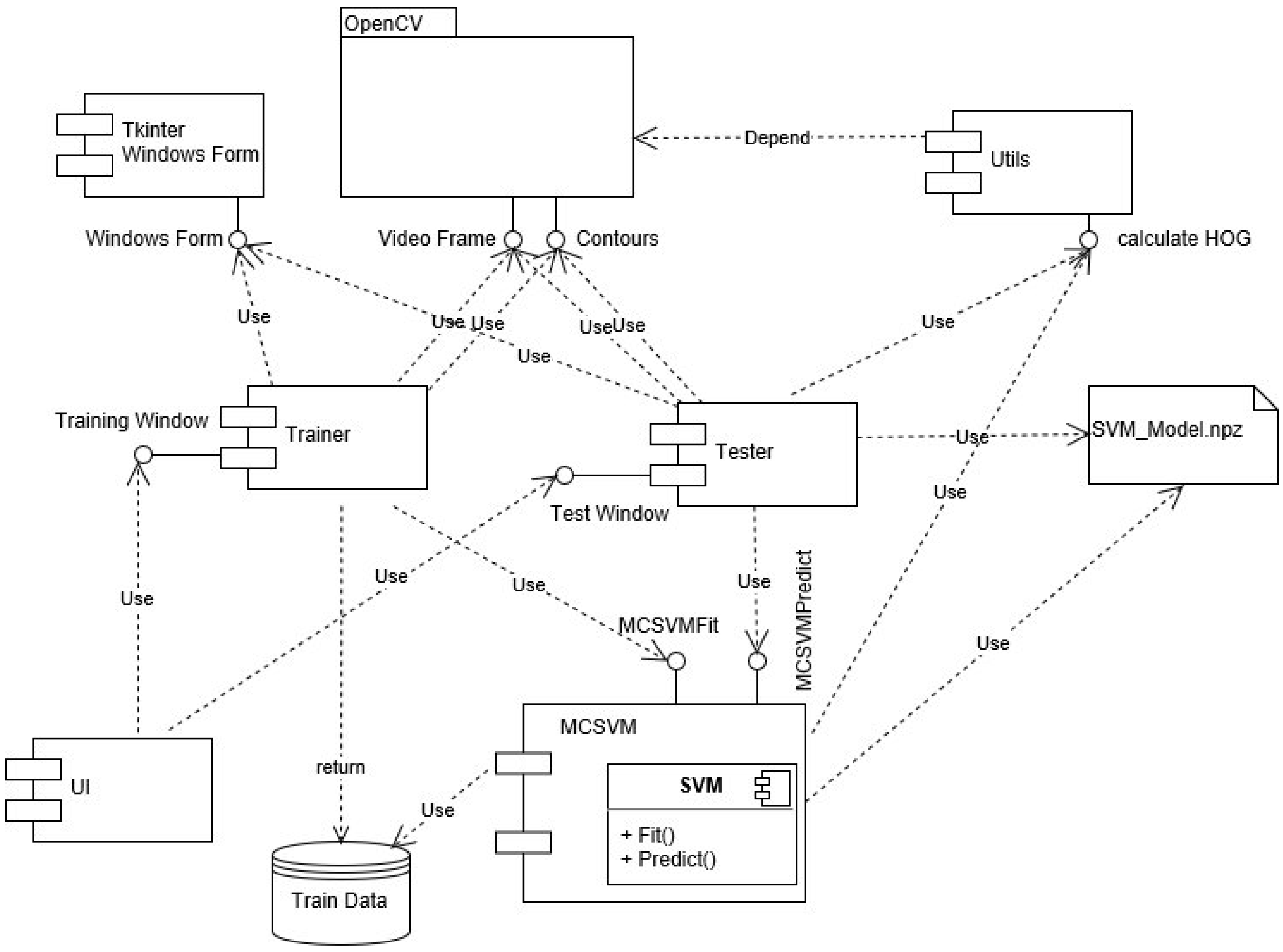


Figure 11 | System Component Diagrams

3.6 Package Diagram

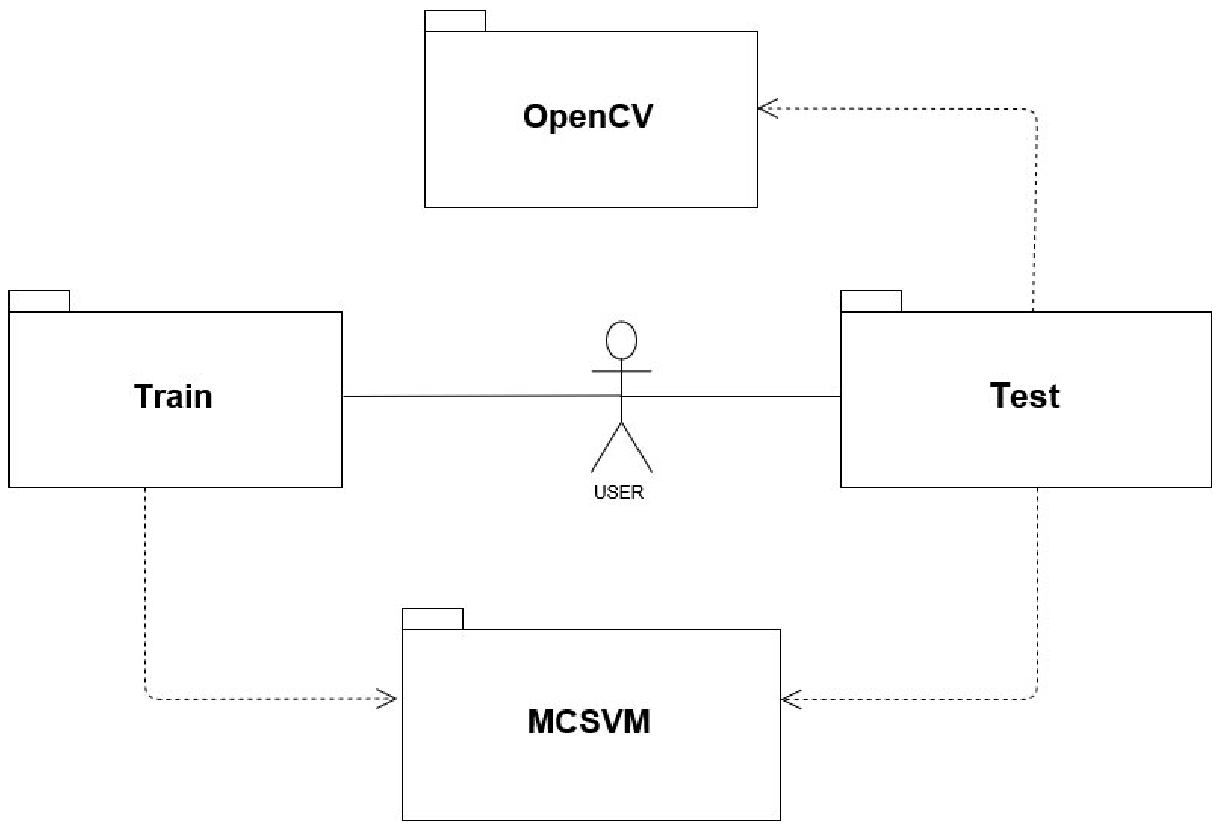


Figure 12 | System Package UML Diagram

3.7 Security and Reliability Features

There is only two data access points during the running of the application. At first, is the storage of training data, and their access to build the model. In addition, there is the creation / update of the model file, and its access during testing. None of the data access instances is in direct interaction with the user, and the user has no control over this data apart from adding more training samples, which is also secure as this process is executed by the application. Even if there is corruption of the model, or it is deleted, the application still won't suffer any problems, as it checks the existence of the model before allowing access to translation.

IV. Implementation

4.1 Necessary Environment Parameters for Execution

The application can run in any operating system that can interpret python 2.7 or above. In addition, required Python packages for import are the following:

- **OpenCV**
  OpenCV is a package for image and video processing in Python. OpenCV is needed for three processes. First we need video stream, and that is done through OpenCV cv2.cam() method. Next, we need object detection in order to detect hand gesture in the marked box, so we can pass them into the SVM predictor. Last, and most important, we need to access HOG descriptor module from OpenCV, that will allow us to implement our **calc_HOG()** method.

- **NumPy**
  Our algorithm, as described above, makes heavy use of algebraic operations and manipulations of large matrices. NumPy is one of the best packages that deals with all these functions, enhancing the usage of matrices also known as **np.array**-s.

- **Tkinter**
  In order to display the frames produced by OpenCV and add some extra user-friendly functionality, we need a way of creating Windows Forms. This functionality in Python is imported by Tkinter. We therefore, use Tkinter for all our three windows i.e. Home, Translate and Train.

- **CVXOPT**
  We closely observed the analytical solution of the SVM classification, in the Software Design section above. In the end of that analytical solution, we are left with a quadratic problem, i.e. a dual problem in Lagrange shape. To solve that system of multiple equations and get our support vectors, we make use of the quadratic solver available at CVXOPT found as **solvers.qp** (Quadratic Programming with Python and CVXOPT). CVXOPT specializes in optimization problems, therefore it is a reliable package to use.

The application does not require internet connection to run. As observed from the memory usage at runtime, 256 MB of RAM memory must be available on the system. In addition, Web Camera is required.

4.2 Code Snippets

The most important part of our implementation in the solution presented, is the implementation of the Support Vector Machine and Multiclass Support Vector Machine.

4.2.1 SVM.fit (x,y) (Quadratic Programming with Python and CVXOPT)

```
def fit(self, X, y):
    n_samples, n_features = X.shape
    n_samples = int(n_samples)
    n_features = int(n_features)

    K = np.zeros((n_samples, n_samples))
    for i in range(n_samples):
        for j in range(n_samples):
            K[i, j] = self.kernel(X[i], X[j])

    P = cvxopt.matrix(np.outer(y, y) * K)
    q = cvxopt.matrix(np.ones(n_samples) * -1)

    A = cvxopt.matrix(y, (1, n_samples))
    b = cvxopt.matrix(0.0)

    if self.C is None:
        G =cvxopt.matrix(np.diag(np.ones(n_samples) * -1))
        h = cvxopt.matrix(np.zeros(n_samples))
    else:
        tmp1 = np.diag(np.ones(n_samples) * -1)
        tmp2 = np.identity(n_samples)
        G = cvxopt.matrix(np.vstack((tmp1, tmp2)))
        tmp1 = np.zeros(n_samples)
        tmp2 = np.ones(n_samples) * self.C
        h = cvxopt.matrix(np.hstack((tmp1, tmp2)))

    # solve Lagrange
    solution = cvxopt.solvers.qp(P, q, G, h, A, b)

    # Lagrange multipliers
    a = np.ravel(solution['x'])

    # SV != 0
    sv = a > 1e-5
    ind = np.arange(len(a))[sv]
    self.a = a[sv]
    self.sv = X[sv]
    self.sv_y = y[sv]
    print("%d support vectors out of %d points" % (len(self.a), n_samples))

    # Intercept
    self.b = 0
    for n in range(len(self.a)):
        self.b += self.sv_y[n]
        self.b -= np.sum(self.a * self.sv_y * K[ind[n], sv])
    self.b /= len(self.a)

    # Weight vector
    if self.kernel == linear_kernel:
        self.w = np.zeros(n_features)
        for n in range(len(self.a)):
            self.w += self.a[n] * self.sv_y[n] * self.sv[n]
    else:
        self.w = None
```

The predict method above accepts as arguments a complete list of samples for the two classes, and a complete list of labels corresponding to each sample, respectively *X and y*. Then, we extract the size of the sample matrix, so we can initialize an empty matrix of the same size, so we can fill it later with the products of all pairs of samples. Note that:

```
for i in range(n_samples):
        for j in range(n_samples):
            K[i, j] = self.kernel(X[i], X[j])
```

corresponds to $\sum \boldsymbol{x_i x_j}$, while

`P = cvxopt.matrix(np.outer(y, y) * K)` corresponds to $\sum \boldsymbol{y_i y_j x_i x_j}$.

4.2.2 SVM.predict(x)

```
def project(self, X):
    if self.w is not None:
        return np.dot(X, self.w) + self.b
    else:
        y_predict = np.zeros(len(X))
        for i in range(len(X)):
            s = 0
            for a, sv_y, sv in zip(self.a, self.sv_y, self.sv):
                s += a * sv_y * self.kernel(X[i], sv)
            y_predict[i] = s
        return y_predict + self.b

def predict(self, X):
    return np.sign(self.project(X))
```

The above code is pretty simple to understand, we simply plug in the coordinates of out training sample into the SVM hyperplane that has been built on our model.

4.2.3 MultiClass SVM Class

4.2.3.1 MCSVM Fit()

```
def multiClassFit(self):

    features = defaultdict(list)
    w = {}
    b = {}

    clf = SVM.SVM()

    for root, dirs, files in os.walk('./Train Data'):
        for sample in files:
            filename = os.path.join(sample)
            path = 'Train Data/' + filename

            im = cv2.imread(path)
            im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
            hog = utils.calc_HOG(im)
            features[filename[0]].append(hog.transpose(1, 0)[0])

    labels = list(features.keys())

    for i in range(0, len(labels)):
        for j in range(i + 1, len(labels)):
            print labels[i]+labels[j]

    for i in range(0, len(labels)):
        for j in range(i + 1, len(labels)):
            neg = np.full((len(features[labels[i]])), -1.0)
            pos = np.full(( len(features[labels[j]])), 1.0)
            tempLabel = np.concatenate((neg, pos), axis=0)
            tempFeatures =
np.concatenate((np.array(features[labels[i]]),
np.array(features[labels[j]])), axis=0)

            clf.fit(tempFeatures, tempLabel);
            self.w[labels[i] + labels[j]] = clf.w
            self.b[labels[i] + labels[j]] = clf.b

    np.savez('storedSVM\\model', w=self.w, b=self.b)
```

This algorithm was described in pseudocode in the software design sector. The importance of this snippet relies on the use of aggregation, i.e. delegation of the SVM class in MCSVM class at: ***clf = SVM.SVM().***

In addition, we should also note the use of **np.savez()** method from NumPy, to save our model into an **.npz** file or numpy's compressed array format. We load that model later through **np.load()** in the predict method following.

4.2.3.2 MCSVM predict()

This code was also presented in software design as a pseudocode, so it will be helpful to look at it for a better understanding.

```python
def multiClassPredict(self, HOG):

    votes = defaultdict(int)
    data = np.load('storedSVM/model.npz')
    w = data['w'].item()
    b = data['b'].item()

    labels =  list(w.keys())

    svm = SVM.SVM()


    for i in range(0, len(labels)):

        svm.w = w[labels[i]]
        svm.b = b[labels[i]]



        if svm.predict(HOG) < 0 :
            votes[labels[i][0]] = votes[labels[i][0]] + 1
        else:
            votes[labels[i][1]] = votes[labels[i][1]] + 1



    return max(votes, key=votes.get)
```

4.2.4 Calc_HOG method

```python
def calc_HOG (gesture_frame):
    winSize = (224, 256)
    blockSize = (16, 16)
    blockStride = (8, 8)
    cellSize = (8, 8)
    nbins = 9
    derivAperture = 1
    winSigma = 4.
    histogramNormType = 2
    L2HysThreshold = 2.0000000000000001e-01
    gammaCorrection = 0
    nlevels = 64
    hog = cv2.HOGDescriptor(winSize, blockSize, blockStride,
cellSize, nbins, derivAperture, winSigma,histogramNormType,
L2HysThreshold, gammaCorrection, nlevels)
    winStride = (1, 1)
    hist = hog.compute(gesture_frame, winStride);
    return hist;
```

This implementation of the **calc_HOG()** illustrates the HOG parameters we make use of to extract features from our gestures. Our window size is *224 x 256px*, blocks are *16 x16px*, and each of them contains 4 cells of *8 x 8px*. The number of orientation bins is set to 9, that is we are grouping angles into 20 degree-groups. The total of features based on these HOG parameters will be *31*27*9*4 = 30132 features!*

V. Results and Conclusions

5.1 Resolution to Starting Goals

Overall, I am excited to write without hesitation that the initial goals of the project purpose were met for the greatest part, and lead to a fulfillment of my purposes. This is reflected on the running application able to predict all hand gestures corresponding to the 5 first letters of the alphabet. Also, besides the completion of a working program, as far as I am concerned, the major outcome of this project, was the introduction, research and technical application of machine learning classification methods unknown to me before. Not only, I could realize the power and usability of machine learning, but through building the “Sign Language Translator”, I saw in many ways, how new technologies can continue to contribute in improving our quality of life, particularly referring here to people with disadvantages.

5.2 Incomplete Features, Problems, and Limitations

During the implementation of the problem solution, several problems were faced, as expected. At first, implementation of HOG algorithm was working, but slow, about 300% slower than the new one implemented using HOGDescriptor from OpenCV. For that reason, it was replaced, however it was used to build the HOGVisual.py script, that helped us visualize the feature extraction process, in real samples. In addition, mixing OpenCV frames, with Tkinter Windows forms was tricky too, and it took enough time. The solution given to that problem, is a working and efficient one, with no limitations, apart from a medium – to high coupling between the Tkinter forms and OpenCV frames, in Testing and Training packages. This is not a major problem neither in functionality, nor in understandability, but it might concern mostly the design part, and extensibility of the program.

5.2.1 Number of Samples

The initial goal on the very beginning was to implement the full alphabet of 26 letters. However, it did not take time to realize that this would be very time consuming considering the 3-month period in disposal for both designing, implementing and training the translator. Therefore, the application was trained for 5 letters with around 40 samples on each. Increasing the number of characters to be predicted, would increase dramatically the number of samples required as well, since SVM would need much more information on each class in order to separate them correctly. Adding to that, the use of CVXOPT allows for a total of 128 samples on each class, so in case we would like to extend classes or train to become more precise we will need to replace our quadratic problem solution approach.

5.2.2 Statistics

Gesture / Character	Success Rate in Prediction	Average Prediction Time
A	> 97 %	~ 1 second
B	> 95%	~ 1.5 seconds
C	> 95%	~ 1 second
D	> 94%	~ 1 second
E	> 92.5%	~ 3 seconds

5.3 Screenshots

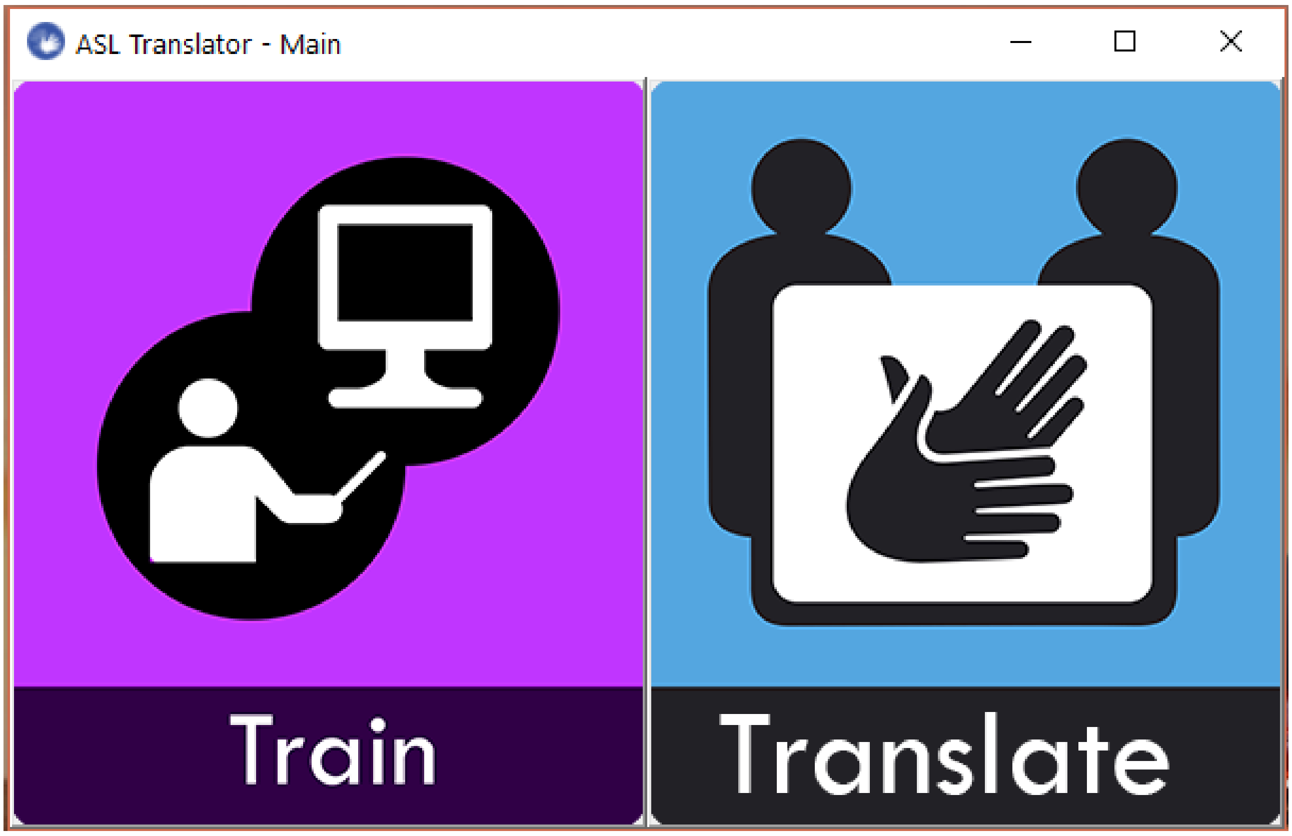


Figure 13 | Home Menu – Training has been Completed, both buttons are active

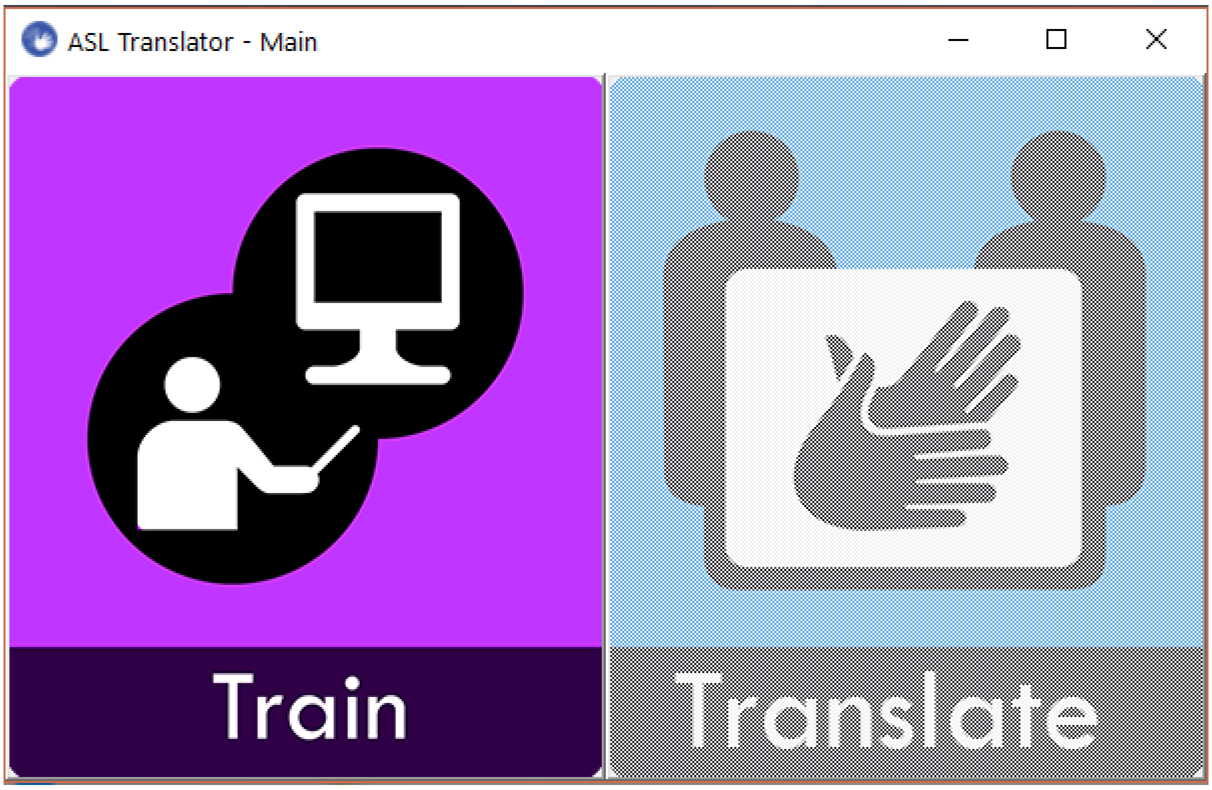


Figure 14 | Home Menu – No training has occurred, translate button is inactive

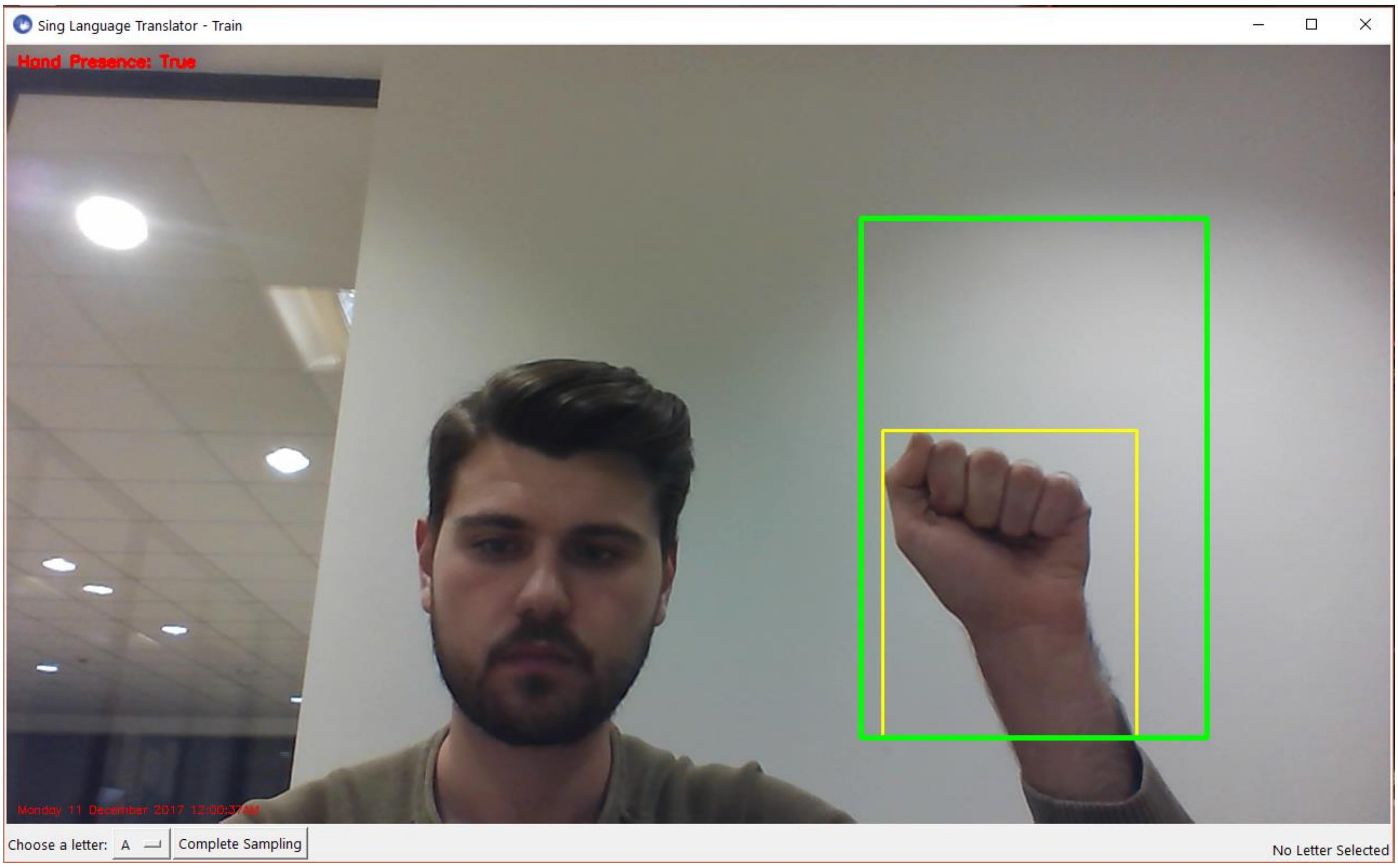


Figure 15 | Training Window - hand detected, no letter selected

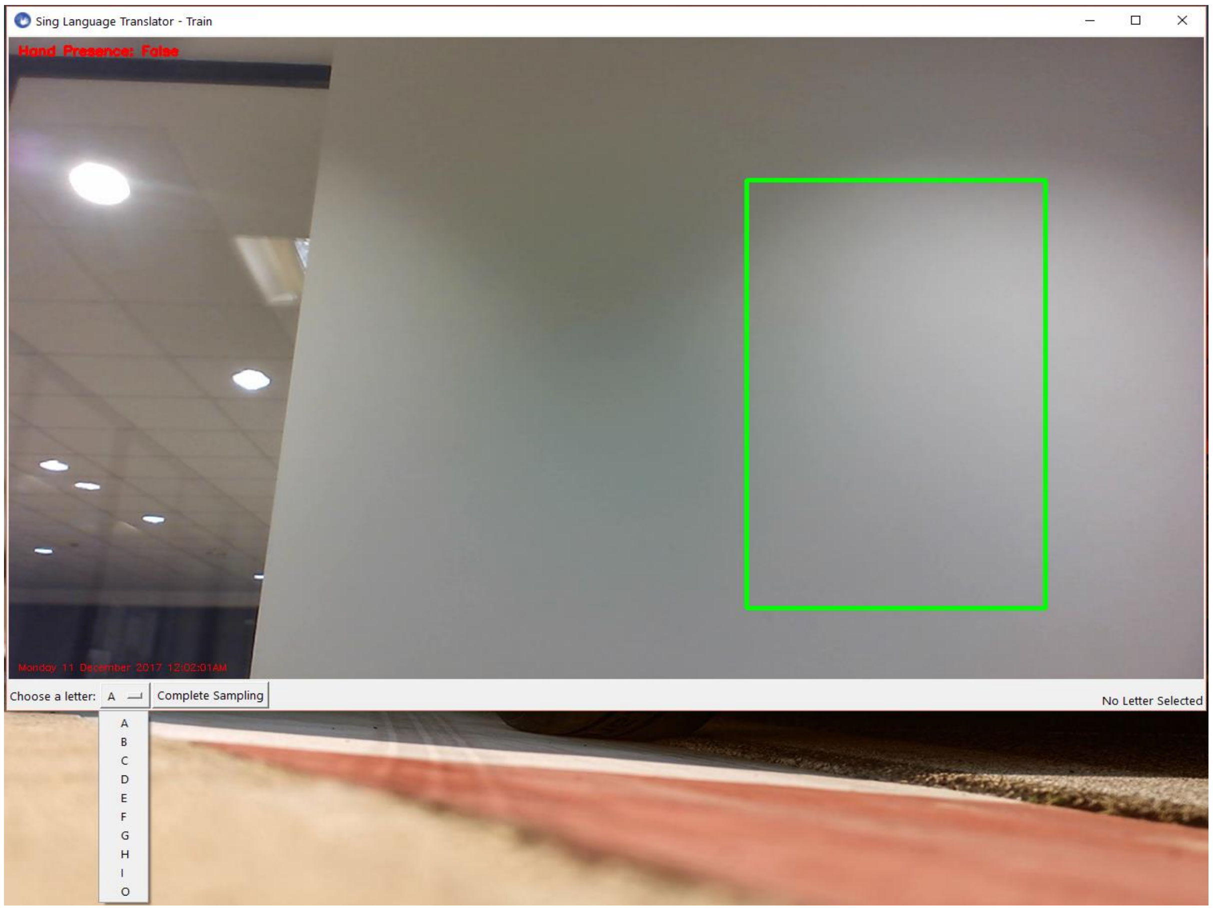


Figure 16 |

Training Window - Dropdown list of character choices

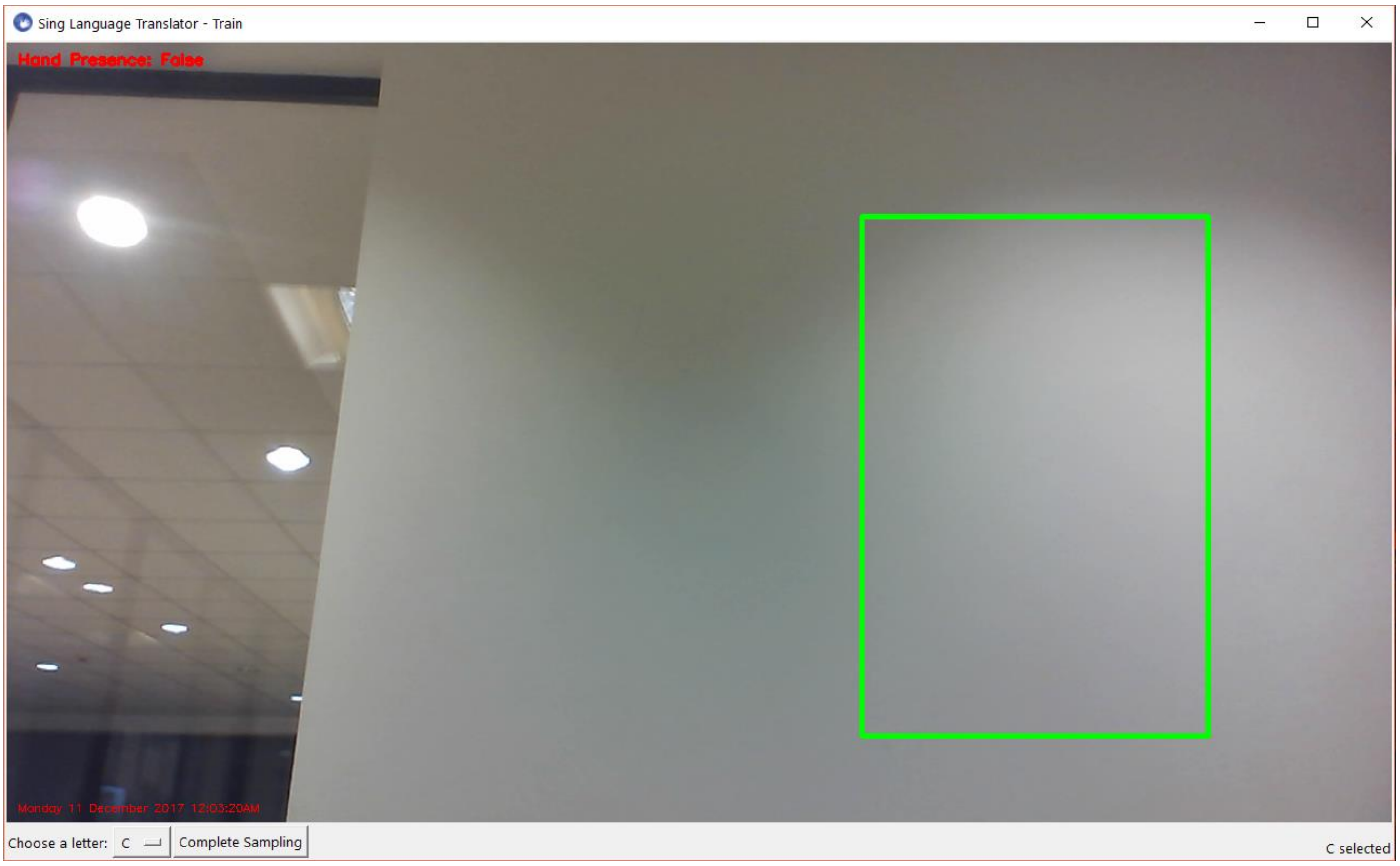

Sing Language Translator - Train
Hand Presence: False
Monday 11 December 2017 12:03:20AM
Choose a letter: C
Complete Sampling
C selected

Figure 17 | Training Window, C selected from the dropdown

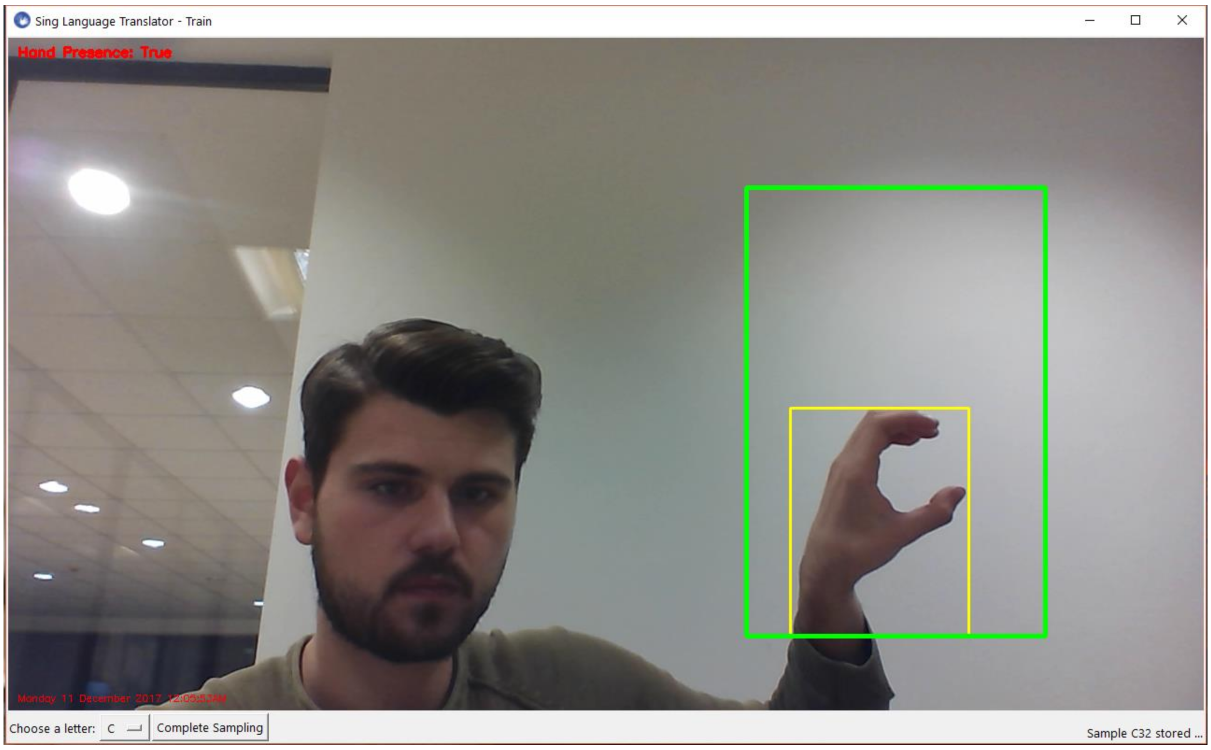


Figure 18 | Training Window, "C32" Stored

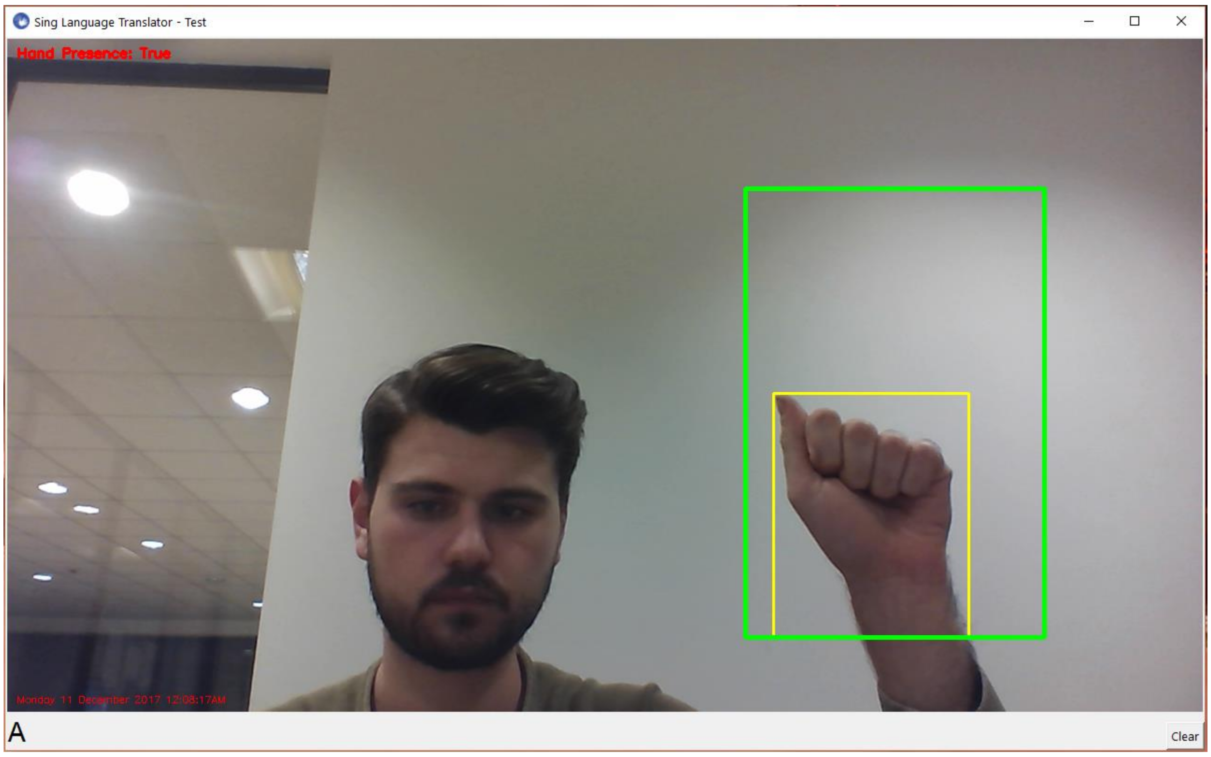

Sing Language Translator - Test
Hand Presence: True
Monday 11 December 2017 12:08:17AM
A
Clear

Figure 19 | Translating Window, "A" Detected

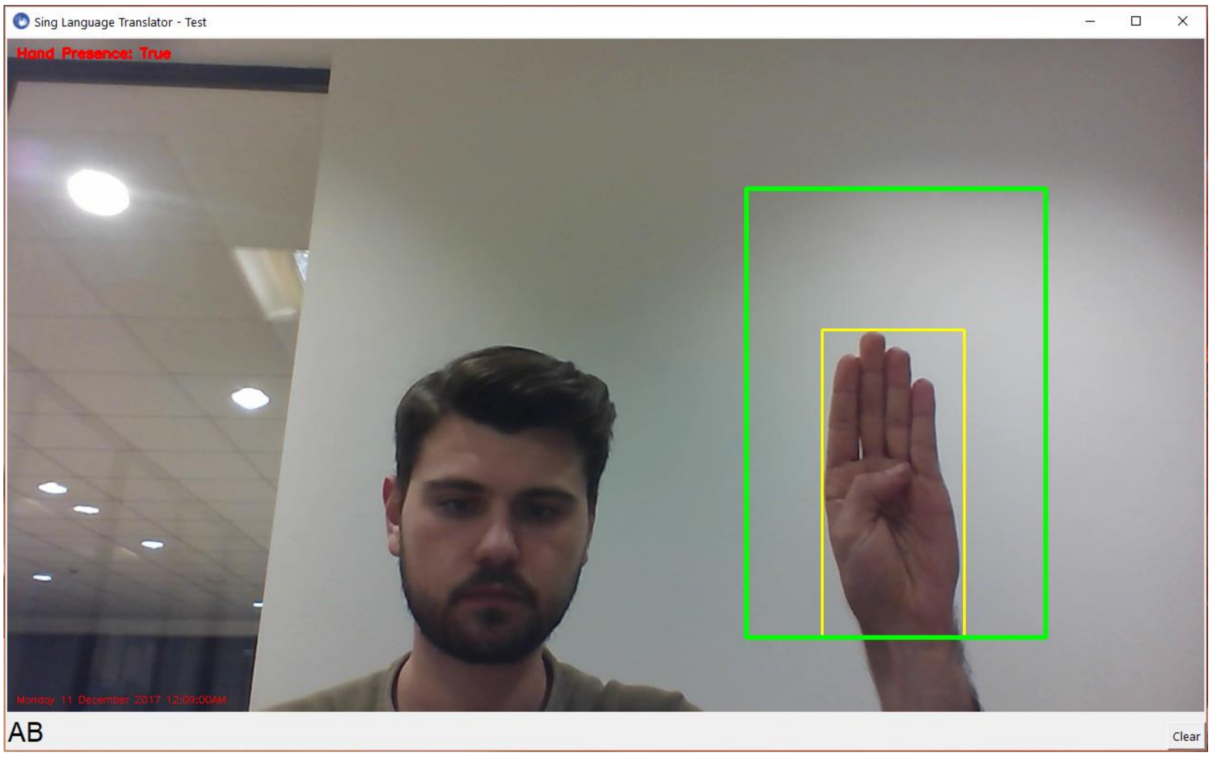

Sing Language Translator - Test
Hand Presence: True
Monday 11 December 2017 12:09:00AM
AB
Clear

Figure 20 | Translating Window, "B" Detected

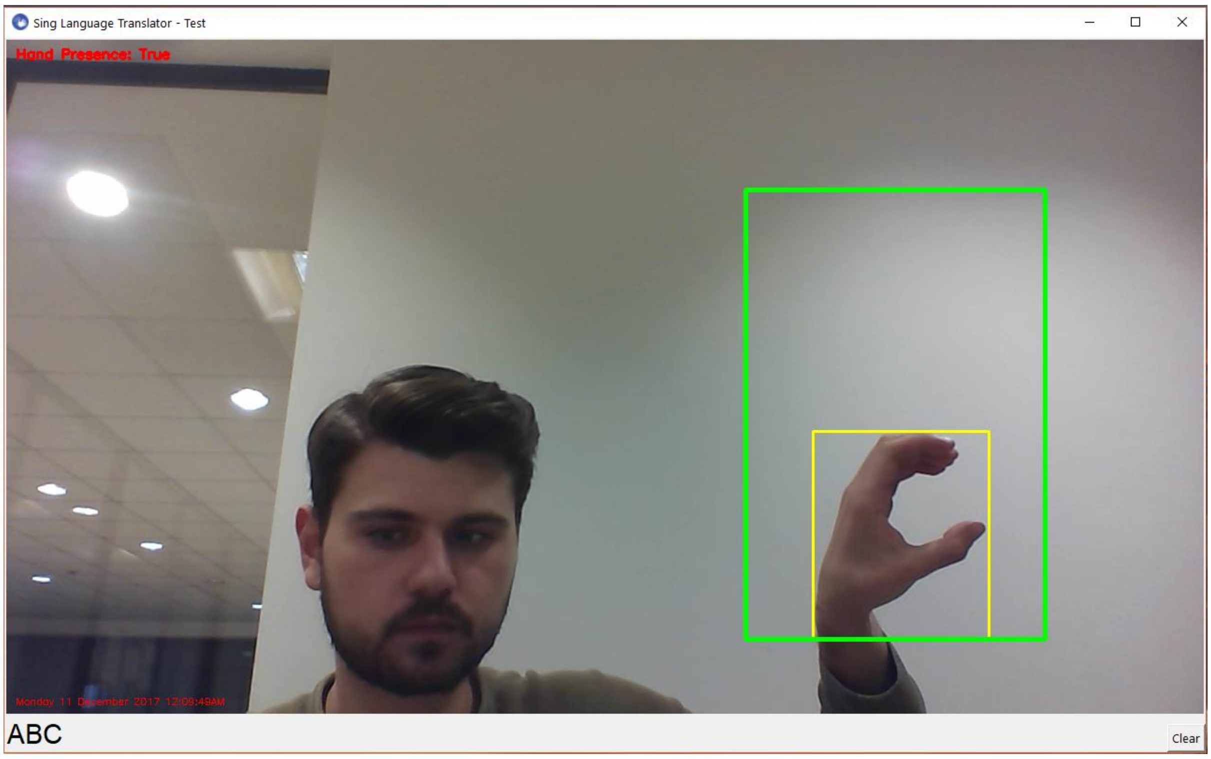
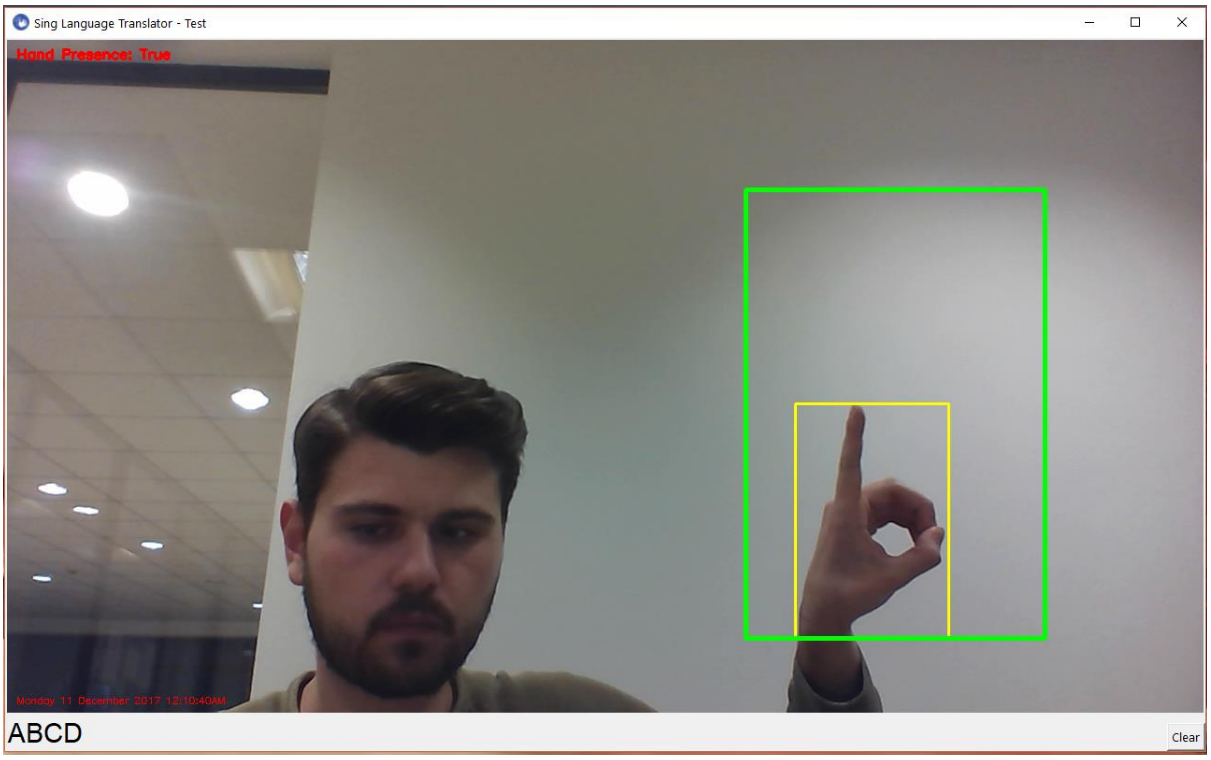
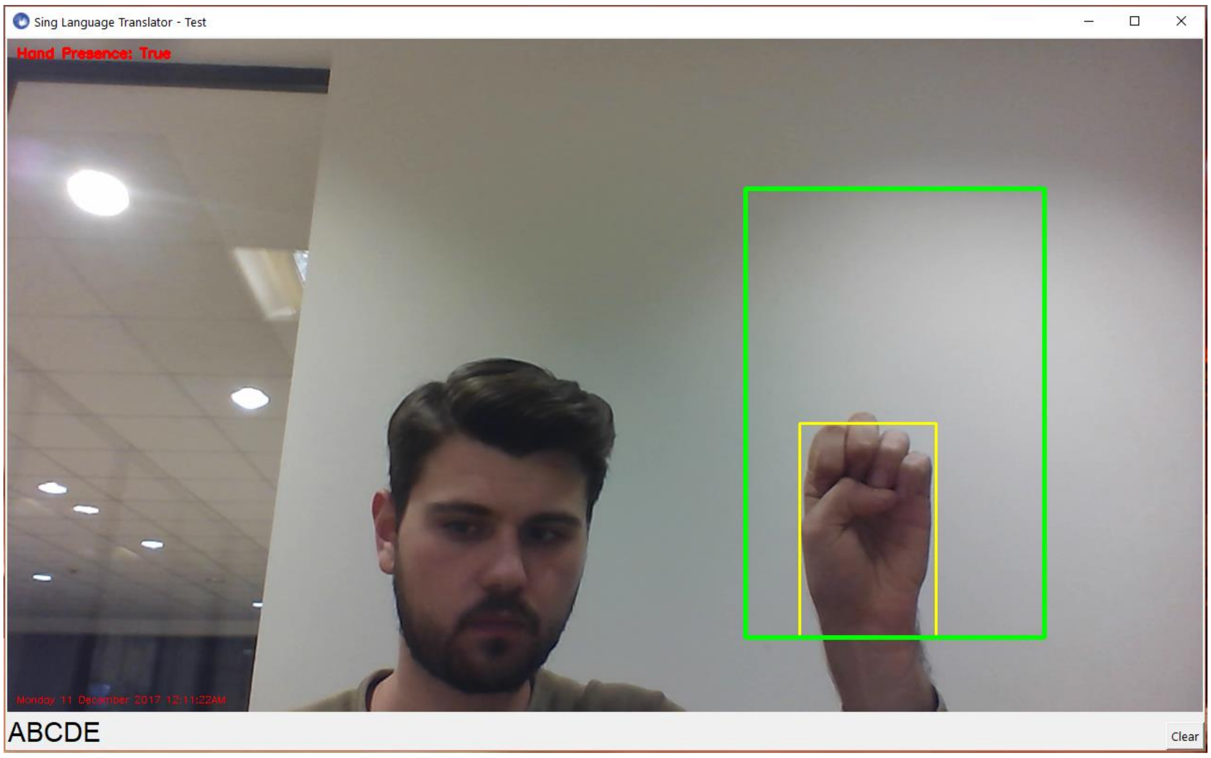

Sing Language Translator - Test
Hand Presence: True
Monday 11 December 2017 12:09:48AM
ABC
Clear

Figure 21 | Translating Window, "C" Detected


Sing Language Translator - Test
Hand Presence: True
Monday 11 December 2017 12:10:40AM
Clear

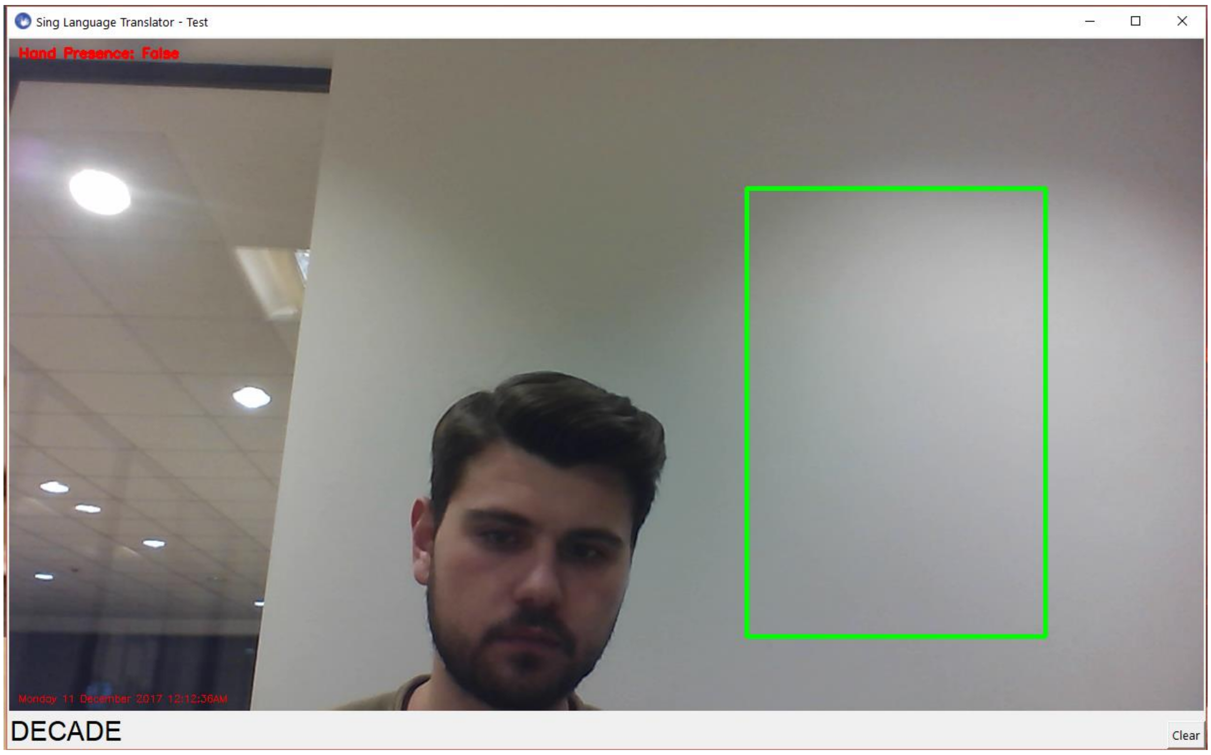
ABCD

Figure 22 | Translating Window, "D" Detected


Sing Language Translator - Test
Hand Presence: True
Monday 11 December 2017 12:11:22AM
Clear

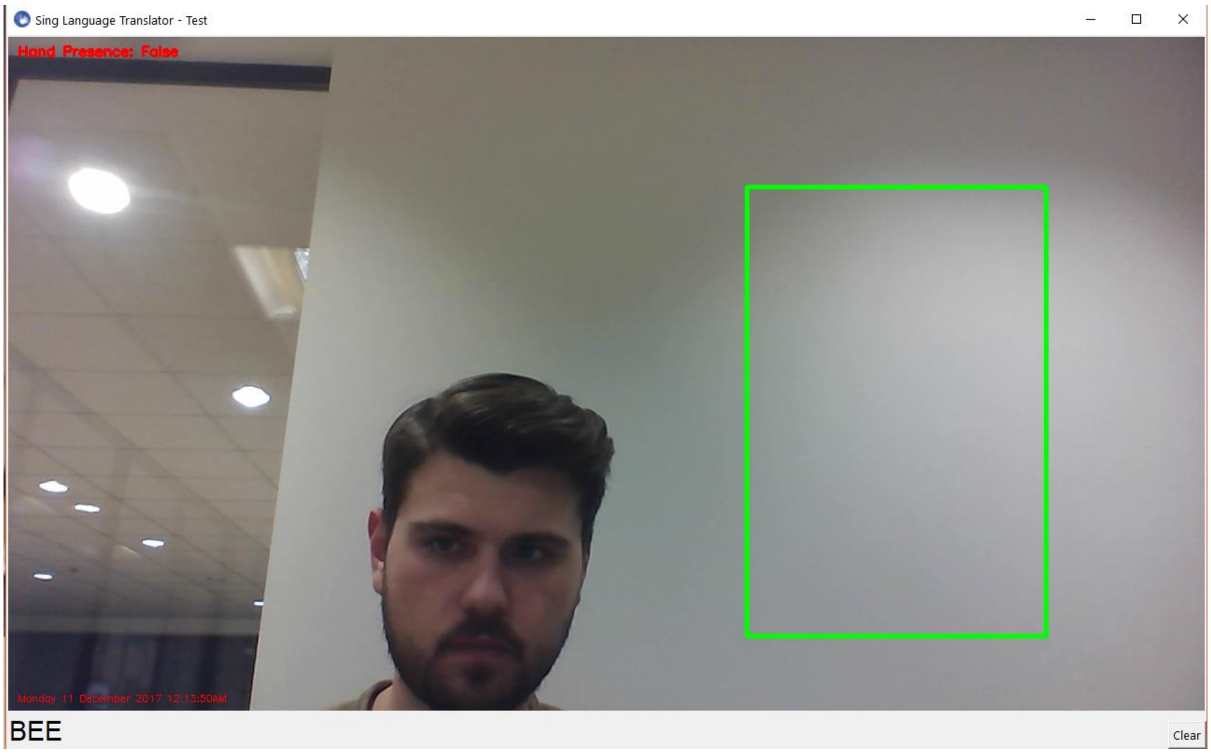
ABCDE

Figure 23 | Translating Window, "E" Detected


Sing Language Translator - Test
Hand Presence: False
Monday 11 December 2017 12:12:36AM
Clear

DECADE

Figure 24 | Translating Window, "Decade" written using ASL


Sing Language Translator - Test
Hand Presence: False
Monday 11 December 2017 12:15:50AM
Clear

BEE

Figure 25 | Translating Window, "BEE" was written using ASL

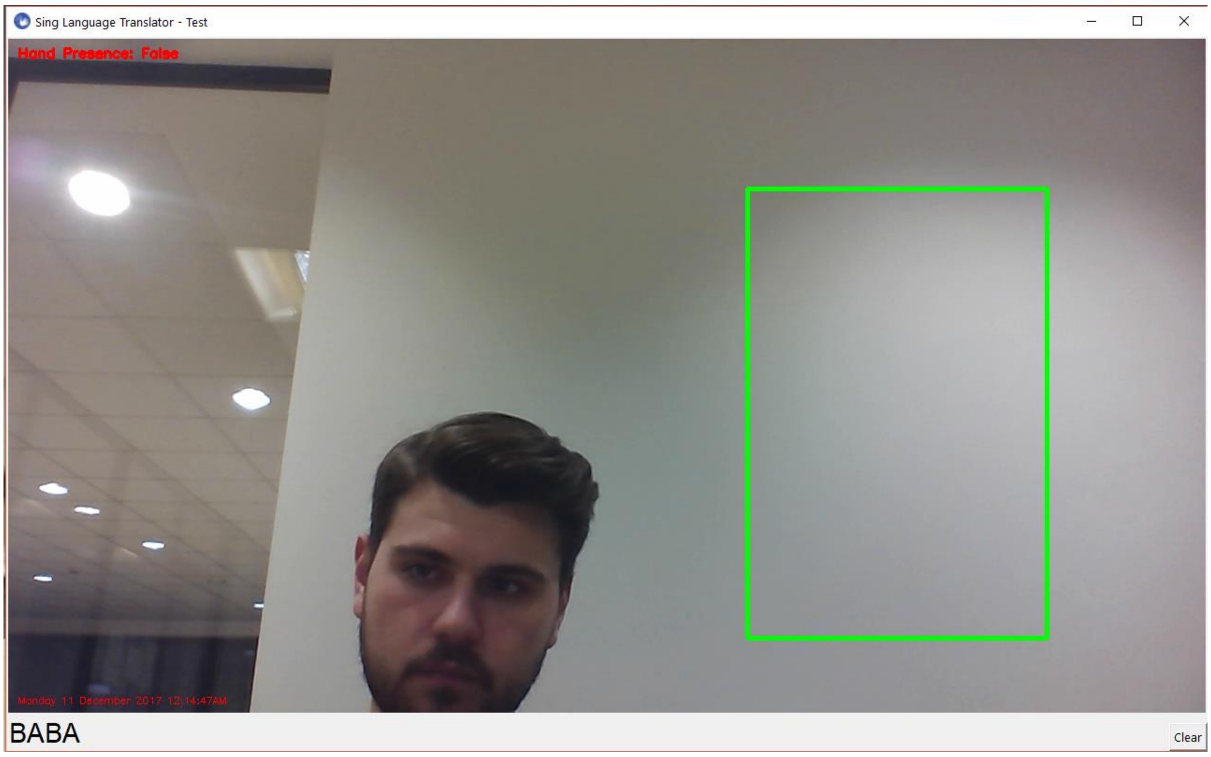

Sing Language Translator - Test
Hand Presence: False
Monday 11 December 2017 12:14:47AM
BABA
Clear

Figure 26 | Translating Window, "BABA" written using ASL

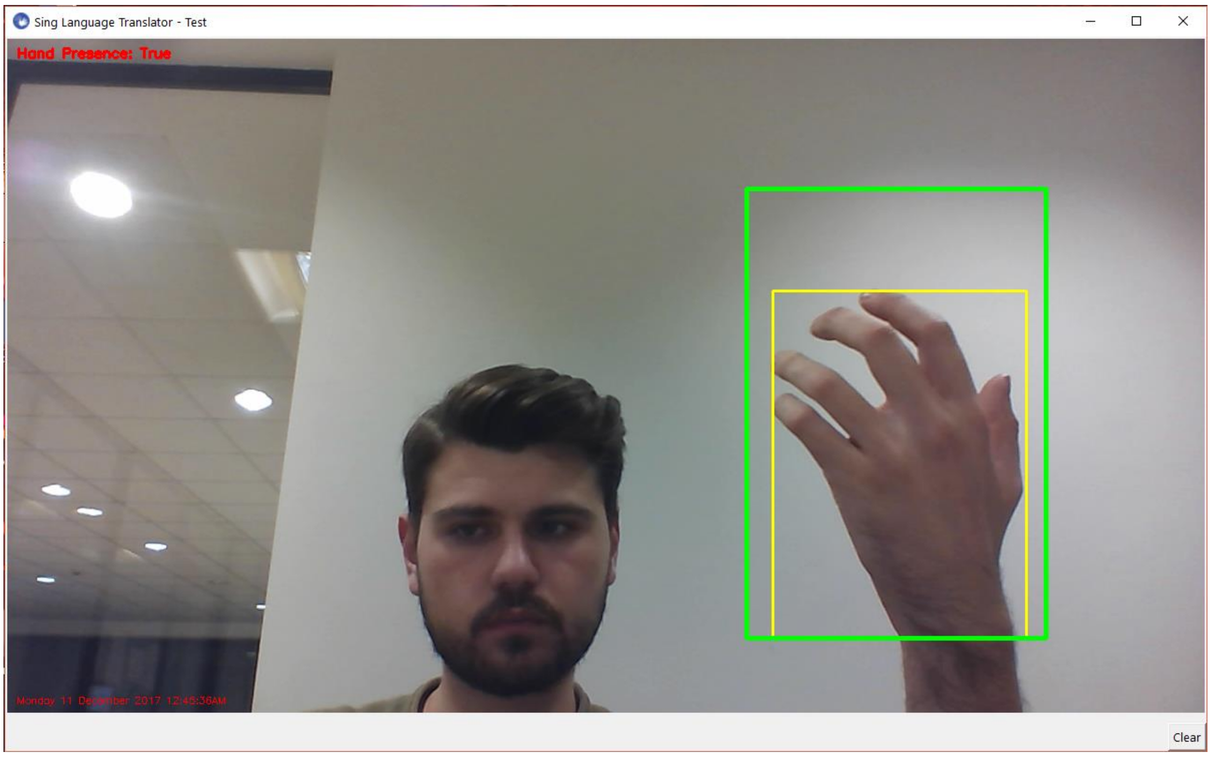

Sing Language Translator - Test
Hand Presence: True
Monday 11 December 2017 12:45:36AM
Clear

Figure 27 | Translating Window, Invalid gesture – no letter printed

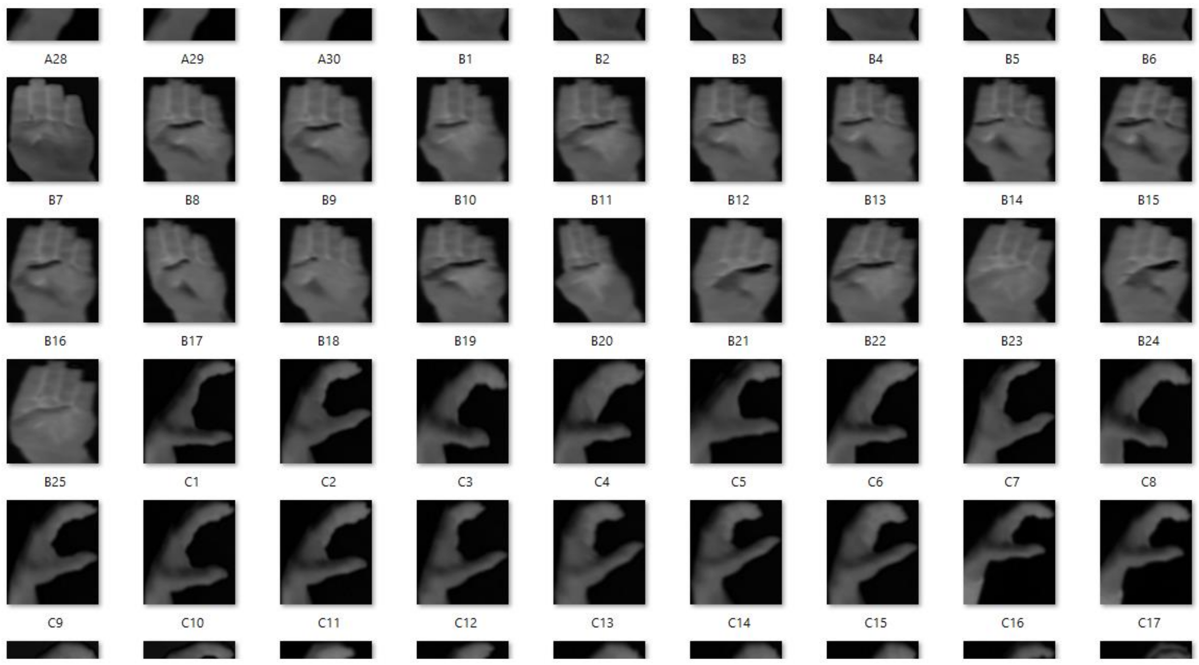


Figure 28| Training Data

VI. References

1. Viswanathan, D. M., & Idicula, S. M. (2014). Recognition of hand gestures of English alphabets using HOG method. *2014 International Conference on Data Science & Engineering (ICDSE)*. doi:10.1109/icdse.2014.6974641

2. Quadratic Programming with Python and CVXOPT. (n.d.). *MIT Open CourseWare,* 1-4. Retrieved December 11, 2017, from https://courses.csail.mit.edu/6.867/wiki/images/a/a7/Qp-cvxopt.pdf.

3. Lin , C. (n.d.). *Optimization,Support Vector Machines, and Machine Learning* [Talk at DIS, University of Rome and IASI , CNR, September, 2005]. National Taiwan University.

4. Math ∩ Computer Science. (2017, July 11). Formulating the Support Vector Machine Optimization Problem. Retrieved December 10, 2017, from https://jeremykun.com/2017/06/05/formulating-the-support-vector-machine-optimization-problem/

5. Stewart, J. (2014). *Stewart Calculus*

6. Primitives for Intel® Architecture Developer Reference. (2017, November 11). Histogram of Oriented Gradients (HOG) Descriptor. Retrieved December 10, 2017, from https://software.intel.com/en-us/ipp-dev-reference-histogram-of-oriented-gradients-hog-descriptor

7. Cone Programming¶. (n.d.). Retrieved December 10, 2017, from http://cvxopt.org/userguide/coneprog.html