Neural Networks and Sign Language

An Investigation Into
Real-Time Sign Language Translating

Shoubhit Abhin
Centre Number – 51123
Candidate Number – 8000

AQA Computer Science NEA

Contents

1 Analysis

1.1 Problem Definition

This investigation targets a specific problem area – communication for individuals who rely on sign language as their primary method of expression. This can be challenging in real-time scenarios where others may not understand sign language. This lack of accessibility affects millions of people globally, creating barriers in education, employment, and social settings. This investigation aims to address this problem.

The investigation is centred around the creation of a software tool capable of interpreting American Sign Language (ASL) gestures in real-time, translating each letter into the written English equivalent. The program will be able to recognise individual hand shapes representing letters and output corresponding letters on the screen, as well as preserving letters which would allow readable words to be formed by sequential gestures. By using computer vision and machine learning techniques, this solution will facilitate communication and make sign language more accessible in real-world situations.

This investigation targets:

- Primary Users: Individuals who communicate in ASL and wish to bridge the communication gap with non-ASL users.
- General Users: This may benefit people unfamiliar with sign language by providing an accessible tool to interpret ASL.
- Educational Institutions: This could serve as a learning aid for both deaf and hearing individuals, fostering inclusivity in educational settings.

To understand the investigative problem and the solution being proposed, a few background concepts are essential:

- Sign Language and Its Structure: Sign language uses hand shapes, movements, and facial expressions to convey meaning, although this project and model will focus specifically on hand shapes on movements rather than facial expressions. The ASL alphabet consists of hand gestures corresponding to each letter. Unlike spoken language, it lacks direct audio output. This means that for sign language users to communicate with non-signers, an interpreter or translation device is needed.
- Computer Vision and Machine Learning for Gesture Recognition: Gesture recognition (in this context) relies on capturing images from a camera, analysing hand shapes, and matching them to a specific, pre-determined list of letters. This process requires either a pre-trained machine learning model or custom-coded algorithms to map visual input (the camera feed) to language output (text).
- Limitations in Existing Technology: Current applications often rely on high-level libraries and pre-trained models, such as PyTorch. These are effective but are resource intensive, therefore reducing their accessibility.

1 Analysis

1.2 Background Research

Primarily, my background research involved exploring existing sign language recognition systems, revealing a heavy reliance on pre-trained machine learning models or advanced libraries (such as TensorFlow or PyTorch) for gesture recognition. These methods can be effective, although they are resource intensive and therefore not computationally viable on all machines, reducing accessibility. It also revealed a low accuracy rate for most systems, especially in foreign conditions, 'foreign' in this context meaning conditions different to the ones the model was trained on. This became the primary focus of my project – an investigation into a system with high accuracy and low dependency on external libraries and files.

Consequently, two approaches were immediately considered, and the advantages and limitations of each approach were weighted and compared. The first approach involves not using any machine learning techniques – this approach will be referred to as the 'non-ML' approach throughout this document from this point forward. The second approach involves using machine learning techniques – this approach will be referred to as the 'ML' approach throughout this document from this point forward.

The advantages and disadvantages of each approach are summarised in *Table 1* below.

Approach	Pros	Cons
ML	Higher accuracy	Larger size
	Real-time performance	Heavier library dependency
	Scalability	Training the model is expensive
non-ML	Smaller file size	Lower accuracy
	No need to train	Limited scalability
	Easier to deploy	

Table 1 – A table comparing the advantages and disadvantages of each approach.

After evaluating the advantages and limitations of both, a non-ML approach was selected for the initial phase due to its lower complexity, reduced dependency on large datasets, and greater portability for a preliminary version of the software.

1 Analysis

1.3 Investigative Discussion

At the beginning of this project, two possible approaches for real-time sign language recognition were considered: a non-machine learning (non-ML) approach and a machine learning (ML) approach. The non-ML approach was initially explored due to its lower complexity, reduced dependency on large datasets, and greater portability. Since non-ML methods rely on predefined rules and manually developed mathematical functions rather than trained models, they seemed more accessible and computationally efficient for deployment on standard hardware.

The primary objective was to extract hand landmarks from a live webcam feed and classify static hand gestures into letters A to Z (excluding J and Z due to their motion requirements). This classification would be achieved by analysing key geometric features such as:

- Contour Detection: Extracting curves to represent hand shapes.
- Background Subtraction: Removing the background to isolate the hand.
- Angle Measurements: Calculating angles between fingers to differentiate gestures.

The expectation was that by developing explicit, rule-based algorithms, gesture recognition could be achieved without requiring a complex training phase, making the system lightweight and portable.

A prototype was developed using OpenCV for real-time image processing and MediaPipe for efficient hand landmark detection. The extracted landmark coordinates were processed using custom mathematical functions to classify gestures. However, despite initial progress, several critical limitations became evident. These included:

1. Inconsistent Recognition: Changes in the environment (such as lighting) affected contour detection, making it difficult to extract stable features.
2. Lack Of Robustness Against Variety: Users with different hand sizes and skin tones experienced varied success rates. Background subtraction was difficult, especially for users who wore skin-coloured clothes.
3. Limited Scalability: Since the system worked on manually defined rules, it had very low accuracy for complicated gestures.
4. Performance Bottlenecks: The need for complex mathematical calculations for every frame made the system computationally expensive and could not maintain a usable frame rate for real-time predictions.

These challenges led to the conclusion that a purely non-ML approach could not deliver the required accuracy and reliability. While theoretically possible, it became evident that the technical skill required to refine the system for robust recognition would exceed the complexity of implementing an ML-based model.

Given the limitations of the non-ML approach, the investigation shifted towards a machine learning-based solution to achieve higher accuracy, better generalisation, and improved real-time performance.

Both approaches were compared again following my initial research and development of the non-ML system. My findings are summarised in *Table 2* below.

Factor	non-ML	ML
Accuracy	Low (30-50%)	Higher
Scalability	Limited due to pre-defined rules	Easily expandable based on the data the model is trained on
Processing Speed	Slower to due to complex calculations being processed	Faster with optimised architecture
Robustness	Highly sensitive to changes in the environment	More adaptable to variation in the environment
Development Effort	High effort	High effort

Table 2 – A table outlining the high-level state transition diagram for a general solution of this investigation.

One of the key driving factors for choosing ML was meeting the client's requirement for high accuracy. Through discussions with stakeholders, it was established that the system had to recognise gestures consistently, reliably and had to work in real-time without significant lag

Since the non-ML approach failed to meet these criteria, the focus of the investigation shifted to creating and optimising a ML implementation of this system. As a result, the following sections – *2 Design*, *3 Development*, and *4 Testing* – will only focus on the ML approach. Brief detail around the non-ML approach can be found in *Section 5 non-ML Approach*. The model for design, outlined in *Section 1.4 Modelling For Design* is relevant for both approaches since it discusses a high-level, conceptual design. Requirements for the ML implementation are outlined in *Section 1.5 Requirements*. A full comparison of both approaches can be found in *Section 4 Testing* and *Section 6 Evaluation*.

1 Analysis

1.4 Modelling For Design

The model for design provides a high-level overview of a general system structure, highlighting the key functionality that would be required from any general sign language to text translator, regardless of the specific technical requirements. The following diagrams illustrate key components and interactions of the system. This model is used as the foundation and built on using the exact system requirements discussed in *Section 1.5 Requirements* to create a high-level system overview for the developed system – see *Section 2 Design* for design details.

The two keys models for design are:

1. State Diagram: This diagram shows the key states of the program (e.g., idle, detecting gestures, translating gestures) and transitions between them.
2. Flowchart: A flowchart of the entire process, from capturing the camera feed to displaying recognised letters on the screen.

The state diagram (1) is outlined in *Table 3*

Current State	Input (Event)	Next State
Idle	Input detected	Detecting Gesture
Detecting Gesture	Features extracted	Translating To Text
Translating To Text	Confirmation of successful translation	Output Text
Output Text	Confirmation of successful output	Waiting For Next Gesture
Waiting For Next Gesture	No new input for X seconds	Idle
Waiting For Next Gesture	New input detected	Detecting Gesture

Table 3 – A table outlining the high-level state transition diagram for a general solution of this investigation.

The state diagram can be represented visually, as shown in *Figure 1*.

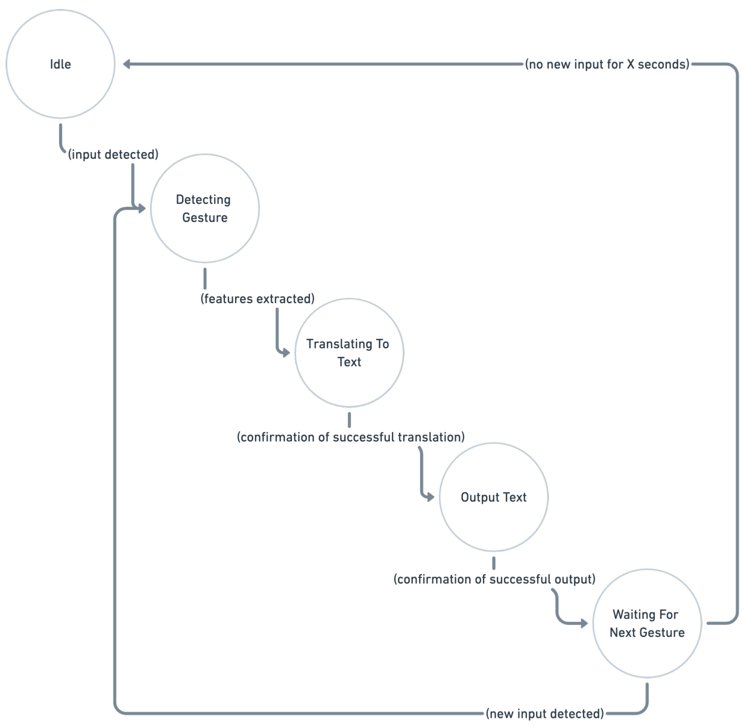


Figure 1 – A visual representation of the state transition diagram outlined in Table 1.

There are no accepting states since this is a continuous process, the only real accepting state being 'program ends' which is trivial and therefore has not been included.

The flowchart (2) is outlined below:

1. Program Executed: The program is executed.
2. Capture Input from Camera: A camera is used to capture a live feed.
3. Pre-Process Image: The input from the camera is pre-processed.
4. Feature Extraction: Key features are extracted.
5. Gesture Recognition: Extracted features are used to recognise the letter being signed.
6. Output Recognised Letter: The identified letter is displayed on the screen.

The flowchart can be represented visually, as shown in *Figure 2*.

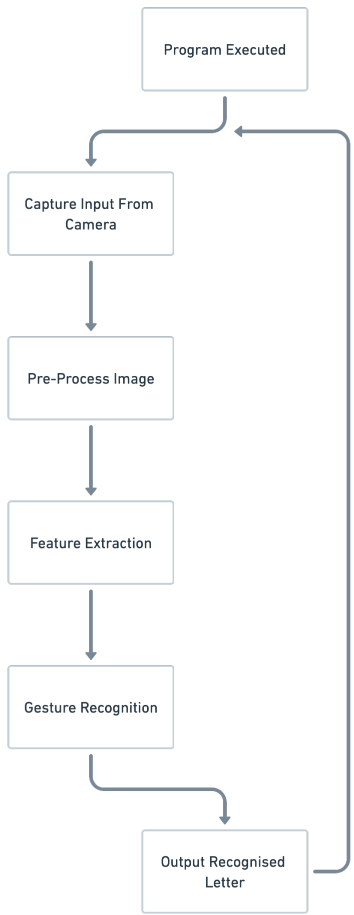


Figure 2 – A visual representation of the written flowchart (2).

1 Analysis

1.5 Data Volumes

For the ML practical implementation of this investigation, a substantial volume of image data is required to effectively train a convolutional neural network (CNN) for hand gesture recognition. To ensure the model is both accurate and reliable, I determined an appropriate dataset size based on research into existing image classification systems and machine learning dataset recommendations.

The total dataset for this project consists of 100,000 images, with 20,000 images per letter class. Each image is stored in JPG format at a resolution of 300x300 pixels. The dataset is divided into separate folders for each letter class within a `savedData/trainOnThese` directory, which is further split into `train` and `val` (validation) subdirectories, evident in the project directory introduced in *Section 2 Design*. This separation allows for consistent model training and performance evaluation on unseen data.

The chosen data volume was informed by research into the relationship between dataset size and classification accuracy in CNNs. Sources recommend that at least 10,000 images per class are typically required for reliable performance in image recognition tasks, particularly when working with limited variability and relatively simple images. To improve model robustness and account for possible image variability (including changes in hand angle, lighting, and background), I doubled this recommendation to 20,000 images per class.

Images were collected manually using a Python-based image capture script via a webcam (discussed further in *Section 2 Design)*, ensuring consistent image size and structure. Data augmentation was then applied separately using a custom script (also discussed further in *Section 2 Design)* to artificially expand the dataset's diversity, improving the model's ability to generalise to new images during testing.

The final data volumes were determined based on:

- Research into machine learning dataset size guidelines.
- Practical limitations of available hardware (a 2020 M1 MacBook Pro without a dedicated GPU).
- Time constraints for dataset collection, processing, and model training within the NEA schedule.

1 Analysis

1.6 Requirements

The requirements specification shown in *Table 4* below contains the main user requirements. The requirements specification shown in *Table* 5 contains the predictive model requirements. Since the focus of the investigation is now the ML approach, the predictive model requirements are for the ML approach only. Comments on the non-ML approach can be found in *Section 5 non-ML Approach*.

To effectively understand and convey the requirements specification, this investigation has been split into three phases, each of which is outlined below. The number before the phase name is the phaseNum (1,2 or 3), and the phaseNum will be used to identify the phase in question. All further analysis, design, development, testing, and evaluation will refer to the following phases and use their respective phaseNum's.

- <u>1 – Data Collection & Augmentation</u> – This phase describes the process behind collecting the data used by the machine learning model, as well as how the data is augmented and the effects this has on the model. Elements from this phase are not of concern to the end user. phaseNum = 1.
- <u>2 – Model Creation & Training</u> – This phase describes the process behind creating the model and using the data collected and augmented in phase 1 to train the model, producing a model that is usable. Elements from this phase are also not of concern to the end user. phaseNum = 2.
- <u>3 – Final Product</u> – This phase describes the process behind the final product, as well as instructions on how to install and operate the system (installation and user guide). Elements from this phase are of concern to the end user. phaseNum = 3.

Individual user requirements are given in *Table 4* below:

It is key to note, requirements highlighted in dark grey are not original user requirements. These are requirements as a consequence of investigation – requirements set after 'mini-investigations' were carried out (more information in *Section 2.4 Log Of Investigative Stages)*. For all development, any requirements in *Table 4* will be treated as user requirements, regardless of whether they are requirements as a consequence of investigation.

ID	Phase	Requirement
REQ1	1	When run, the data collection script should open a new window which displays the camera feed.
REQ2	1	When run, the data collection script should capture the hand image and display this image cropped in a small, separate window.
REQ3	1	When run, the data collection script should highlight the landmarks on a hand when the hand is completely visible to the camera.
REQ4	1	The data collection script should be able to capture a still image triggered by a specific key press.
REQ5	1	The data collection script should be able to save an image captured according to the requirement *REQ4*, in a designated folder, with each image having a unique identifier based on the date and time it was captured.
REQ6	1	The data augmentation script should define a path to a data directory and, when run, ensure this is a valid path (i.e. the file folder exists).
REQ7	1	The data augmentation script should augment data based on appropriate parameters, such as zooming and rotation. It should not change the size of the image or the aspect ratio.
REQ8	1	The data augmentation script should be able to save the augmented images to a specified folder, and if the folder does not exist, create a folder with the same identifier in the location specified
REQ9	3	The app script should have functionality regarding a trained model, defined by the following sub-requirements: • *REQ9a* – Ask the user which model they would like to load from a selection of models • *REQ9b* – Return an appropriate error message if the requested trained model cannot be loaded
REQ10	3	The app script should define the class labels.

REQ11	3	When run, the app script should have functionality regarding the webcam, defined by the following sub-requirements: • *REQ11a* – Access and display a working and connected webcam *REQ11b* – Return an appropriate error message if a working webcam cannot be found or accessed properly
REQ12	3	The app script should include the main functionality of the app, defined by the following sub-requirements: • *REQ12a* – Display an appropriate title • *REQ12b* – Display an appropriate quit message • *REQ12c* – Display the predicted labels and probabilities of each label • *REQ12d* – Display the gesture queue • *REQ12e* – Display a status message (for operations outlines in *REQ18*) • *REQ12f* – Update the displayed information • *REQ12g* – Process Predicted Letter • *REQ12h* – Save gesture queue to file • *REQ12g* – Display the confidence threshold
REQ13	3	The app script should handle the functionality of the gesture queue, defined by the following sub-requirements: • *REQ13a* – Initialise a queue • *REQ13b* – Enqueue • *REQ13c* – Dequeue • *REQ13d* – Clear the queue • *REQ13e* – Return the whole queue • *REQ13f* – Ensure the queue stores no more than 10 letters at a time
REQ14	3	The app script should have logic to take a specific frame and preprocess it to change the dimensions while not skewing the image to ensure input consistency.
REQ15	3	The app script should be able to use the trained model and the pre-processed frame to make a prediction for that specific frame.
REQ16	3	The app script should include a confidence threshold, which is a set limit and any predictions under this limit are not considered as accurate enough to be a definite prediction.
REQ17	3	The app script should be able to return the (definite) prediction made to meet *REQ12c* and *REQ12f*.
REQ18	3	The app script should be able to predict accurately[1] the first five letters of the English alphabet, these being 'A', 'B', 'C', 'D' and 'E'.

[1]accurately is loosely defined as a success rate of 80%+, although a full, expanded definition with examples is given in Section 4 - Testing

REQ19	3	The app script should include an 'Unknown' label, which is classified for all gestures that are not A,B,C,D or E, including random 'noise' which does not mean anything.
REQ20	3	The app script should have functionality that upon a key press, the queue can be cleared or saved to a file. An appropriate message must be displayed to meet *REQ12e*.
REQ21	3	The confidence threshold defined in *REQ16* should be able to be adjusted manually using a key press (+ to increase by 0.05 and – to decrease by 0.05). The updated confidence threshold should be reflected in the displayed confidence threshold according to *REQ12g*.

Table 4 – A table outlining the user requirements for the implementation of this project.

Individual predictive model requirements are given in *Table 5* below:

ID	Phase	Requirement
REQ22	2	The model definition script should define a convolutional neural network (CNN), which can be trained to classify landmark patterns corresponding to the position of the hand.
REQ23	2	The model definition script should include multiple convolutional layers while also meeting requirement *REQ22*.
REQ24	2	The model definition script should include a flattening layer while also meeting requirement *REQ22*.
REQ25	2	The model definition script should include a dense layer using an appropriate activation function while also meeting requirement *REQ22*.
REQ26	2	The model definition script should include a fully connected layer with dropout to reduce overfitting while also meeting requirement *REQ22*.
REQ27	2	The model definition script should include an output layer using an appropriate activation function while also meeting requirement *REQ22*.
REQ28	2	The model definition script should include a compile statement with an appropriate optimiser and loss function while also meeting requirement *REQ22*.
REQ29	2	The model training script should define a CNN model architecture which can be used to train an appropriate model.
REQ30	2	The model training script should validate input dimensions and reject incompatible data.
REQ31	2	The model training script should process the data according to a pre-determined train-test split.
REQ32	2	The model training script should evaluate appropriate class weights based on the input data given and output class indices to verify correct encoding.
REQ33	2	The model training script should have an early stopping function based on the validation accuracy of an epoch.
REQ34	2	The model training script should, when run, train the model based on a set of input data and the model training script and save the trained model in a specified location.

Table 5 – A table outlining the predictive model requirements for the implementation of this project.

Please Note:

The following sections –

2 Design
3 Development
4 Testing

are relevant to the ML approach of this project. For further details on the non-machine learning approach, see *Section 5 non-ML Approach.*

2 Design

2.1 System Design

The system design has been split according to the phases discussed above: *1 Data Collection & Augmentation, 2 Model Creation & Training,* and *3 Final Product.*

To understand the project, a project directory has been included.

```
├── CNNModels
  └── model1.h5
  └── model1.keras
  └── model2.h5
  └── model2.keras
  └── model3.h5
  └── model3.keras
  └── model4.h5
  └── model4.keras
  └── model5.h5
  └── model5.keras
  └── model6.h5
  └── model6.keras
├── dataCollectionAndAugmentation
  └── dataAugmentation.py
  └── dataCollection.py
├── extraFiles
  └── linkedListGestureQueue.py
  └── saveAsh5.py
├── filesForTesting
  └── confusionMatrix.py
  └── fileSavingUnitTest.py
  └── gestureProcessingUnitTest.py
  └── queueOperationUnitTest.py
  └── UIDisplayUnitTest.py
├── finalProduct
  └── app.py
├── modelCreationAndTraining
  └── model.py
  └── train.py
├── useThisFolderToRunTheProgram
  └── app.py
  └── CNNModels
    └── model1.keras
    └── model2.keras
    └── model3.keras
    └── model4.keras
    └── model5.keras
    └── model6.keras
├── dataForAllModels
  └── model1
    └── augmented
    └── notAugmented
    └── trainOnThese
      └── evaluation
        └── A
        └── B
        └── C
        └── D
        └── E
        └── Unknown
      └── train
        └── A
        └── B
        └── C
        └── D
        └── E
        └── Unknown
  └── model2
    └── ... (same as model1)
  └── model3
    └── ... (same as model1)
  └── model4
    └── ... (same as model1)
  └── model5
    └── ... (same as model1)
  └── model6
    └── ... (same as model1)
 ├── requirements.txt
```

2 Design

2.1 System Design

2.1.1 Data Collection & Augmentation Phase

The system for data collection and augmentation can be broken down using the following hierarchy:

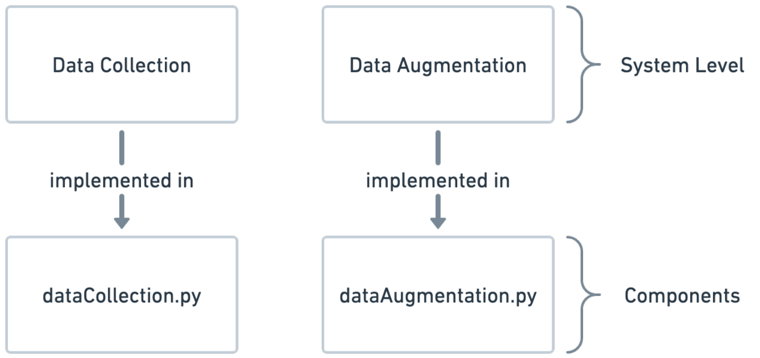


Figure 3 – A high-level structure chart of the data collection and augmentation phase

Exploring the system flow for each implementation of the system level structures shown above, we arrive at the following diagrams:

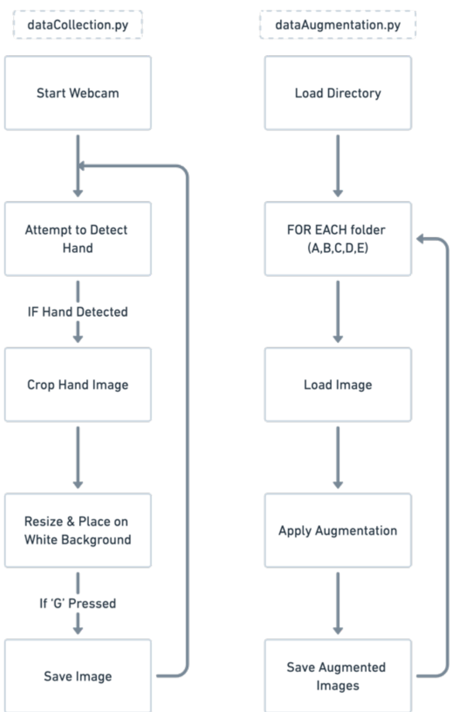


Figure 4 – A structure diagram of the `dataCollection.py` *and* `dataAugmentation.py` *files*

Both the `dataCollection.py` and `dataAugmentation.py` scripts employ the procedural techniques of sequencing, selection and iteration:

- Sequencing: Both scripts follow a clear sequence of actions where each step depends on the successful completion of the previous one. In `datacollection.py`, the sequence involves initialising the webcam, detecting hands, cropping and resizing, and saving images. Similarly, in `dataAugmentation.py`, the sequence is to read the original images, apply augmentations, and save the modified images.

- Selection: There is logic-based selection in both scripts. In `datacollection.py`, it checks if a hand is detected before proceeding with the image processing steps. In `dataAugmentation.py`, the code checks for valid image files and processes only those.

- Iteration: Both scripts make use of iteration:
    - `datacollection.py` continuously captures images in a loop, allowing the user to take multiple photos.
    - `dataAugmentation.py` iterates over all the images in each letter folder, applying augmentation to each.

The `dataCollection.py` script also employ modular coding practices through subroutines, including:

- `initialiseCamera` (*Pseudocode Snippet 1*)
- `captureHandImage` (*Pseudocode Snippet 2*)
- `resizeImage`
- `saveImage`
- `main`

The *Pseudocode Snippets* are shown below:

```
FUNCTION initialiseCamera():
          cap = cv2.VideoCapture(defaultCamera)
          detector = HandDetector(maxHands=1)
          RETURN cap, detector
ENDFUNCTION

#Pseudocode Snippet 1
```

```
FUNCTION captureHandImage(camera, detector, imageSize, offset):

          # Capture an image frame from the webcam
          success, img = camera.read()

          # If the frame capture fails
          IF not successful:
                    PRINT "Fail"
                    RETURN none, none
          ENDIF

          # Use the HandDetector to find the hands in the frame
          hands, img = HandDetector.findHands(img)

          # If no hands are detected
          IF hands is empty
                    RETURN none, img
          ENDIF

          # If hands are detected, process the first hand only
          hand = hands[0]

          # Get the bounding box coordinates of the detected hand
          x, y, w, h = hand["bbox"]

          # Try to crop the hand region from the image

          TRY:
          imgCrop = Crop image from img using coordinates (x, y,
w, h) with Offset padding

          # Return the cropped hand image and the full image
          RETURN imgCrop, img

          CATCH any error:

          PRINT "Error while cropping"
          RETURN None, img
ENDFUNCTION

#Pseudocode Snippet 2
```

2 Design

2.1 System Design

2.1.2 Model Creation & Training Phase

The system for model creation & training can be broken down using the following hierarchy:

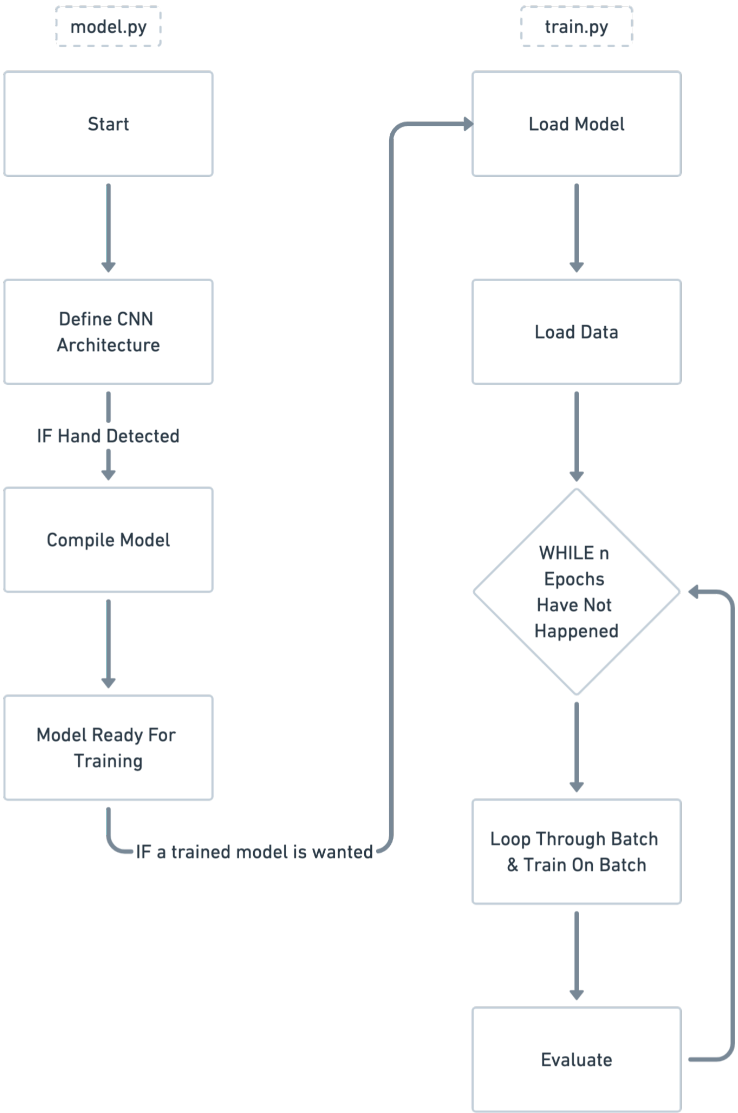


Figure 5 – A high-level structure chart of the model creation and training phase

Both the `model.py` and `train.py` scripts employ the procedural techniques of sequencing, selection and iteration:

- <u>Sequencing:</u> In `model.py`, the model is built sequentially using layers like `Conv2D`, `MaxPooling2D`, `Flatten`, and `Dense` to create a CNN for classification. After the model is created, the compile method is called to prepare it for training. In `train.py`, the training data is loaded and pre-processed in sequence. The model is trained using a loop over the number of epochs. After each epoch, the model's performance (accuracy, loss) is evaluated using validation data.

- <u>Selection:</u> In both `model.py` and `train.py`, selections are made regarding:
  - <u>Layer Choices:</u> The type of layers (convolution, pooling, dense) used in the model is chosen based on the desired architecture.
  - <u>Hyperparameters:</u> Hyperparameters like learning rate, number of epochs, batch size, optimizer, and loss function are selected in `train.py`.
  - <u>Data Directories:</u> Based on the folder structure (`trainOnThese`), selection is made on how data is accessed for training and validation.

- <u>Iteration:</u> The model is trained in iterations (epochs), where each epoch represents one complete pass through the training data. After each forward pass, the weights are updated iteratively based on the backpropagation algorithm. In `train.py`, the system iterates over the train and val data folders to load batches of images for training and validation.

The `model.py` and `train.py` scripts also employ modular coding practices through subroutines, including:

In `model.py`:
- `buildModel()`: This function creates the model by sequentially adding layers, such as convolutional layers, max-pooling layers, flattening, and dense layers.
- `compileModel()`: This function prepares the model for training by selecting an optimizer and loss function.

In `train.py`:
- `trainModel()`: This subroutine trains the model using the training data and validation data. It iterates through the data in batches and updates weights based on backpropagation.
- `evaluateModel()`: This subroutine checks the model's performance on the validation data at the end of each epoch.

2 Design

2.1 System Design

2.1.3 Final Product Phase

The system for the final product is implemented using the `app.py` script, which forms the main execution layer of the system. It coordinates model selection, video capture, and display of results via a custom GUI. The `app.py` script follows an object-oriented approach, utilising class structure. It also incorporates control structures such as sequencing, selection and iteration, which increases readability, reusability and maintainability.

The system uses has three main classes used to separate three distinct functions of the system. The `modelSelector` class and the `signLanguageTranslatorApp` class will be discussed in this section, while the `gestureQueue` class will be discussed in *Section 2.2 Data Structures & Algorithms*.

It is key to note that this running this project requires certain pieces of hardware, notably a webcam. An appropriate webcam <u>must</u> be available to the machine running the `app.py` script, otherwise no gestures will be recognised. The script will also run best on a machine with a dedicated, high-performance GPU or a high-performance CPU capable of intensive graphical and mathematical calculation. This is because the CNN is computationally expensive and requires large amounts of processing power, hence the program may be slow on older or less capable machines.

Class: `modelSelector`

- Purpose: Responsible for listing available trained CNN models from a specified directory, prompting the user to select one, and then loading the selected model for use.
- System Design: Encapsulates all logic relating to the model selection and loading, ensuring that model management is separate, modular and isolated from the main program.

The `modelSelector` class can be represented using a class diagram:

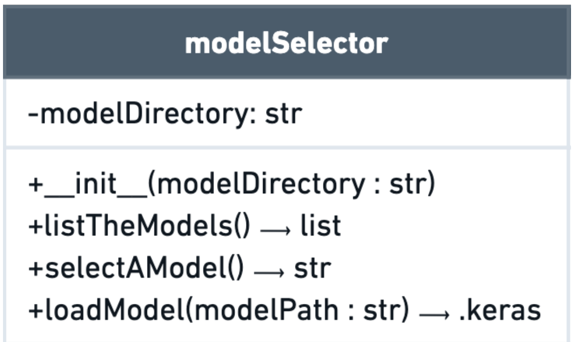


Figure 6 – A class diagram for the `modelSelector` *class.*

The `modelSelector` class has the following attributes:

- `modelDirectory`: a string path to the directory storing the models

The `modelSelector` class has the following methods:

- `__init__(modelDirectory)`: constructor method
- `listTheModels()`: lists the available `.keras` models.
- `selectAModel()`: prompts the user to select a model and returns the appropriate path if the model exists.
- `loadModel()`: loads the selected model using TensorFlow.

Class: `signLanguageTranslatorApp`

- Purpose: Manages the UI display elements, handling predictions, maintaining a gesture queue (although the queue's operations are under the `gestureQueue` class, covered in *Section 2.2 Data Structures & Algorithms*), real-time feedback, status messaging and confidence threshold management.
- System Design: Represents the core control of the application, abstracting away display management and prediction handing from the main procedural flow.

The `signLanguageTranslatorApp` class can be represented using a class diagram:

Figure 7 – A class diagram for the `signLanguageTranslatorApp` class.

The `signLanguageTranslatorApp` class has attributes which track the program state. It also has methods which handle UI updates, prediction processing, queue saving, and displaying messages.

The final product script also employs procedural elements, such as sequencing, selection, and iteration.

<u>Sequencing</u>: The final product script uses sequencing by having a clear order of execution, ensuring operations are executed logically and predictably. This can be represented by the flowchart below:

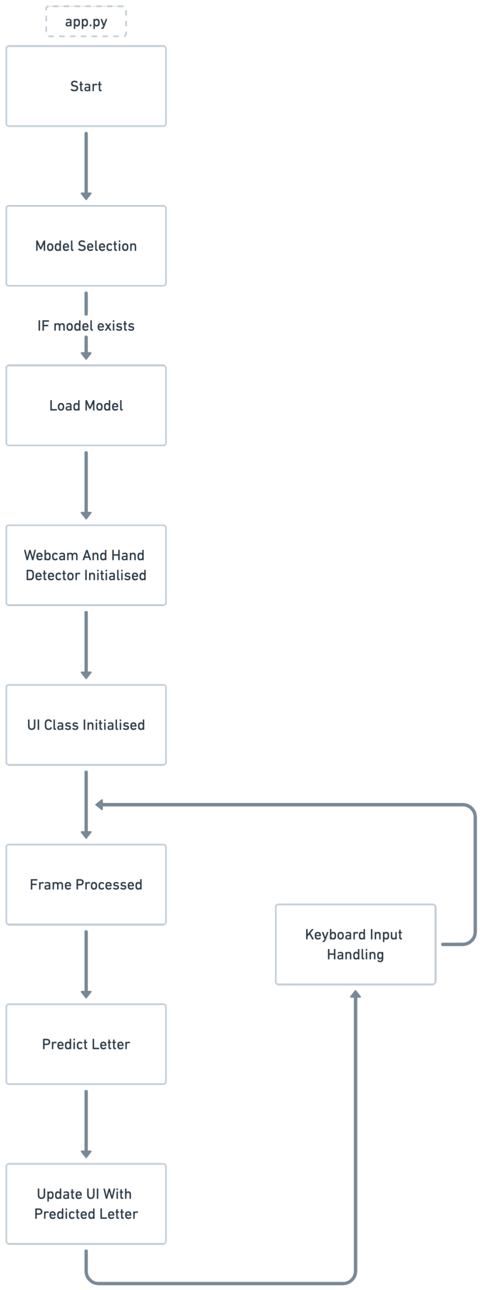


Figure 8 – Main program flow for the `app.py` *script, evidence of sequencing.*

Selection: The final product script uses selection structures to make decisions at several points. Examples of decisions made include:

- Check whether any hands are detected
- Decide whether a prediction is above a confidence threshold (*Pseudocode Snippet 3)*
- Decide what to enqueue based on the predicted label
- Handle different keyboard inputs for system control
- Error handling (*Pseudocode Snippet 4*)

```
IF predictionConfidence < confidenceThreshold THEN
    prediction ← "Unknown"
ELSE
    prediction ← classLabel
ENDIF

#Pseudocode Snippet 3
```

```
TRY
    initialisingTheCamera
EXCEPT error
    OUTPUT "Error"
    STOP initialisingTheCamera
END TRYEXCEPT

#Pseudocode Snippet 4
```

Iteration: The primary iteration is the continuous while loop, shown in *Code Snippet 1.* This iteration enables real-time, continuous processing.

```
while True:
    success, img = cap.read()
    …

#Code Snippet 1
```

The `app.py` script also utilises files ready for direct access:

<u>Files for Direct Access:</u>

- The pre-trained model is stored as a file (as a `.keras` file) on disk, and the path to this model file is passed into the `loadModel()` method.
- The app interacts with the model and UI, so these files are accessed directly when the app is running.

<u>File Access Flow:</u>

- Model File: The model is loaded directly into memory through `modelSelectorInstance.loadModel(selectedModelPath)`

The `app.py` script also employs exception handling in the form of try-except statements. These are demonstrated in the following Pseudocode Snippets – *Pseudocode Snippet 5* and *Pseudocode Snippet 6*.

```
TRY
    loadingTheModel
EXCEPT error
    OUTPUT error
    STOP loadingTheModel
END TRYEXCEPT

#Pseudocode Snippet 5
```

```
TRY
    initialisingTheCamera
EXCEPT error
    OUTPUT error
    STOP initialisingTheCamera
END TRYEXCEPT

#Pseudocode Snippet 6
```

The exception handling demonstrated above ensures the system can detect, handle and recover from unexpected errors gracefully, without crashing or demonstrating unintended behaviours. This improves the robustness of the system and consequently the user experience.

The system also allows dynamic adjustment of the confidence threshold, modifying the strictness of the gesture recognition. This enhances system flexibility, giving the user control over recognition sensitivity during runtime. This functionality is demonstrated in *Pseudocode Snippet 7*.

```
IF keypress = '+'
    confidenceThreshold = confidenceThreshold + 0.05
ELSEIF keypress = '-'
    confidenceThreshold = confidenceThreshold - 0.05
ENDIF

#Pseudocode Snippet 7
```

The final product script also relies on some external libraries, which are outlined below. To install these libraries, use the `requirements.txt` file, with more details on how to run the program on your own machine in *Section 7*.

The external dependencies are as follows:

- `OpenCV`: Enables webcam access for video capture and used to create a simple GUI.
- `TensorFlow`: Used to load and run the CNN model.
- `cvzone.HandTrackingModule`: Used to detect hands in the video frame.
- `datetime, time, os`: Used to time-stamp files, delays and file operations.

The final product script can also be represented using a finite state diagram, defined as follows:

The finite set of states:
- Start – The application starts but hasn't initialized any components.
- UI Initialized – The OpenCV window is created.
- Camera Activated – Webcam starts capturing frames.
- Hand Detected – Hand is successfully detected in the frame.
- Hand Processed – The region of interest (ROI) is extracted and pre-processed.
- Prediction Made – The model predicts a sign language letter.
- UI Updated – The predicted letter is displayed on the UI.
- Exit State – The application closes when the user presses 'Q'.

The transition alphabet:
- start() → Start application
- initUi() → Initialize UI
- activateCamera() → Start webcam
- detectHand() → Detect a hand in the frame
- processHand() → Extract and preprocess the hand image
- predictLetter() → Pass the image to the model and get a prediction
- updateUi() → Update the displayed UI with predictions
- pressQ() → Quit the application

We must redefine our states to create a state transition diagram:
States:
- S0: Initial State (GUI not initialized, no model loaded)
- S1: GUI Initialized
- S2: Model Loaded
- S3: Webcam Feed Captured
- S4: Hand Detected
- S5: Hand Processed (Preprocessing Complete)
- S6: Prediction Displayed
- S7: Exit State

Alphabet of Transitions:
- a: Initialize GUI
- b: Load Model
- c: Capture Webcam Feed
- d: Detect Hand
- e: Process Hand Image
- f: Predict Gesture
- g: Update Display
- h: Quit

This creates the following state transition functions, represented as a finite state machine below:

δ (S0, a) = (S1,⊠, R)   (Initializing GUI)
δ (S1, b) = (S2, ⊠, R)   (Loading model)
δ (S2, c) = (S3, ⊠, R)   (Capturing webcam feed)
δ (S3, d) = (S4, ⊠, R)   (Detecting hand)
δ (S4, e) = (S5, ⊠, R)   (Processing hand image)
δ (S5, f) = (S6, ⊠, R)   (Predicting gesture)
δ (S6, g) = (S6, ⊠, R)   (Updating display, loop continues)
δ (S6, h) = (S7, ⊠, R)   (Exit)

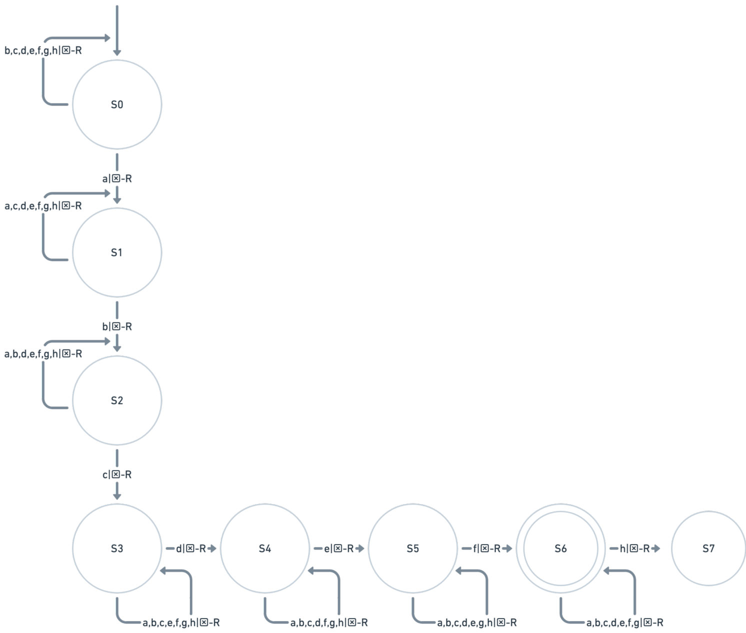


Figure 9 – A state transition diagram for the final product phase, where ⊠ represents a blank and R represents a move one place to the right

2 Design

2.2 Data Structures & Algorithms

2.2.1 Data Collection & Augmentation Phase

The data collection & augmentation phase employs a varied selection of data structures and algorithms.

Firstly, two scripts are involved in this phase:

- `dataCollection.py` : The data collection script, which collects images of hand signs and stores them in a structured dataset.
- `dataAugmentation.py` : The data augmentation script, which applies transformations to the collected images to increase dataset diversity.

Files have also been organised for direct access, shown below through a subset of the directory introduced earlier.

```
├── dataForAllModels
  └── model1
    └── augmented
    └── notAugmented
    └── trainOnThese
      └── evaluation
        └── A
        └── B
        └── C
        └── D
        └── E
        └── Unknown
      └── train
        └── A
        └── B
        └── C
        └── D
        └── E
        └── Unknown
  └── model2
    └── ... (same as model1)
  └── model3
    └── ... (same as model1)
  └── model4
    └── ... (same as model1)
  └── model5
    └── ... (same as model1)
  └── model6
    └── ... (same as model1)
```

While multiple data structures and algorithms are used throughout the data collection & augmentation scripts, this section focuses on one representative data structure and one key algorithm from each file. This approach allows for clarity and depth of explanation without duplicating similar patterns already demonstrated elsewhere in the project.

A key data structure in the data collection script is the 3D NumPy Array.

In the data collection script, images are loaded and manipulated using NumPy 3D arrays, which represent the image in a structured format:

- The first dimension represents the images' height
- The second dimension represents the images' width
- The third dimension represents the three colour channels: red, green and blue.

This structure makes it efficient to manipulate pixel values directly and perform operations like cropping, resizing, and adding white backgrounds.

A key algorithm in the data collection script is the aspect-ratio-preserving algorithm. After detecting a hand in a video frame, the program crops around the hand's bounding box with an offset to give some padding. It then resizes the cropped image to fit within a 300×300 pixel square, while preserving the original image's aspect ratio (to avoid distortion). Any empty space is filled with white pixels. This ensures that all training images are consistent in size, improving model performance while retaining the shape of the gesture signed.

This algorithm can be visually represented, as shown below in *Figure 10*.

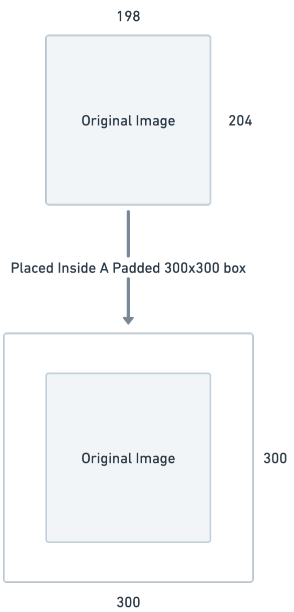


Figure 10 – An illustration of the aspect-ratio preserving algorithm.

This algorithm is demonstrated in *Pseudocode Snippet 8*.

```
FUNCTION cropAndResizeImage(image, boundingBox, imageSize):

    offset = 20

    // Get coordinates from bounding box
    SET x, y, w, h TO boundingBox values

    // Crop the region of interest with padding
    croppedImage ( image[y - offset : y + h + offset, x - 
offset : x + w + offset]

    // Get cropped image height and width
    height ( height of croppedImage
    width ( width of croppedImage

    // Calculate aspect ratio
    aspectRatio ( height / width

    // Create a white square canvas of size imageSize × 
imageSize
    CREATE whiteCanvas filled with white pixels of shape 
(imageSize, imageSize, 3)

    IF aspectRatio > 1 THEN:
        // Image is taller than it is wide
        scaleFactor ( imageSize / height
        newWidth ( round(scaleFactor × width)
        RESIZE croppedImage TO (newWidth, imageSize)
        xOffset ( (imageSize - newWidth) DIV 2
        PLACE resized croppedImage onto whiteCanvas at 
position (0, xOffset)

    ELSE:
        // Image is wider than it is tall
        scaleFactor ( imageSize / width
        newHeight ( round(scaleFactor × height)
        RESIZE croppedImage TO (imageSize, newHeight)
        yOffset ( (imageSize - newHeight) DIV 2
        PLACE resized croppedImage onto whiteCanvas at 
position (yOffset, 0)
    ENDIF

    RETURN whiteCanvas

#Pseudocode Snippet 8
```

A key data structure in the data augmentation script is the list.

During data augmentation, multiple new images are generated from each original image by applying transformations like flipping, rotating, and shifting. These new images are stored temporarily in the `augmentedImages` list.

A list is an ideal choice here because:
- Lists can dynamically grow as new images are added.
- The order of images is preserved.
- Lists allows easy iteration when saving images to disk.

A key algorithm in the data augmentation script is the shifting (translation) algorithm. The shifting (translation) algorithm moves an image horizontally and/or vertically by a small, random amount. This increases positional variety in the training data and helps the model learn to recognize gestures regardless of position.

It uses a transformation matrix to calculate the new pixel positions for the image. This is shown below:

$$\begin{pmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \end{pmatrix}$$

$where\ \Delta x, \Delta y\ is\ the\ horizontal\ and\ vertical\ shift\ values\ respectively.$

2 Design

2.2 Data Structures & Algorithms

2.2.2 Model Creation & Training Phase

The main model is is structured using the TensorFlow Sequential API, making this a key data structure.

```
model = Sequential([
    Conv2D(32, (3,3), activation='relu', input_shape=(300, 300,
3)),
    MaxPooling2D(pool_size=(2,2)),
    Conv2D(64, (3,3), activation='relu'),
    MaxPooling2D(pool_size=(2,2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(len(class_labels), activation='softmax')
])

#Code Snippet 2
```

As per *Code Snippet 2*, we can understand what the CNN actually does:

- Conv2D: Extracts features from images.
- MaxPooling2D: Reduces dimensionality.
- Flatten: Converts the feature maps into a 1D vector.
- Dense Layers: Fully connected layers for classification.

The TensorFlow Sequential API is an essential data structure since it makes building and managing complex networks modular and readable. The sequential data structure holds the model's architecture, weight matrices, and configuration as an ordered list of layers, each with its own properties.

For more information on neural networks, convolutional neural networks or the TensorFlow Sequential API, click on the relevant underlined phrase.

There are also two major algorithms in the model script: the forwards propagation and backwards propagation algorithms.

Forward Propagation: Inputs are passed through each layer, applying weight multiplications, activation functions and ultimately producing a prediction.

Backward Propagation: The prediction is compared to the actual value via the loss function. The network then calculates the error gradient and propagates it backwards, adjusting the weights and biases to reduce error via an optimiser.

These algorithms are essential and at the core at how a CNN learns over epochs. To learn more about either the Forward Propagation Algorithm or the Backward Propagation Algorithm, click on the relevant underline phrase.

Another essential 'data structure' are the hyperparameters. Hyperparameters are pre-set configuration values that control the learning process but are not learned by the model itself. Choosing the right hyperparameters directly affects how fast the model learns and its' accuracy when attempting to generalise. Initial values were chosen based on common CNN conventions and small-scale testing to balance accuracy and speed. Examples of hyperparameters from the model script are shown below in *Table 6*.

Hyperparameter	Example In Code	What It Controls
Number of Filters	`Conv2D(32, (3,3))`	Number of feature detectors applied per convolution.
Kernel Size	`(3,3)`	The size of the filter window moving across the image.
Activation Function	`"relu", "softmax"`	The non-linear transformation applied at each layer.
Dropout Rate	`Dropout(0.5)`	The percentage of nodes randomly deactivated to prevent overfitting.
Number Of Dense Units	`Dense(256)`	Number of neurons in the fully connected layer.

The training script features many data structures and algorithms.

A key data structure is the `ImageDataGenerator`. This is a data pipeline object that generates batches of images for training and validation. It efficiently loads images from directories, applies preprocessing, and converts them into `NumPy` arrays ready for the model. It is important since it allows for real-time image processing during training without loading the entire dataset into memory — ideal for large image sets.
This data structure is shown in *Code Snippet 3*.

```
train_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(...)

#Code Snippet 3
```

A key algorithm is the Early Stopping algorithm. Early Stopping is an optimisation algorithm used to prevent overfitting. It monitors a performance metric (like validation accuracy) during training, and if it stops improving after a set number of epochs (known as the patience), the training halts and the model reverts to its best-performing state. This makes it important since it reduces wasted computation, prevents overfitting, and ensures you keep the best-performing model, improving the overall efficiency and effectiveness of training.
This algorithm is shown in *Code Snippet 4*.

```
early_stopping = EarlyStopping(monitor='val_accuracy',
patience=5, restore_best_weights=True)

#Code Snippet 4
```

2 Design

2.2 Data Structures & Algorithms

2.2.3 Final Product Phase

The final product phase utilises many data structures and algorithms, the key data structure being the gesture queue, contained within the `gestureQueue` class.

The `gestureQueue` class is a classic example of the queue linear data structure, it being responsible for managing a queue of predicted sign language letters. It temporarily stores recognised gestures in the order they are detected, allowing the user to build up a sequence of letters (a "word" or "phrase") in real time.

It provides operations to:

- Add new letters (enqueue) to the queue.
- Remove letters (dequeue) if needed.
- Clear the entire queue.
- Retrieve the current queue contents for display in the UI.

This ensures a structured, reliable way to record and manage sequential gesture predictions in the final application.

The `gestureQueue` class can be represented using a class diagram:

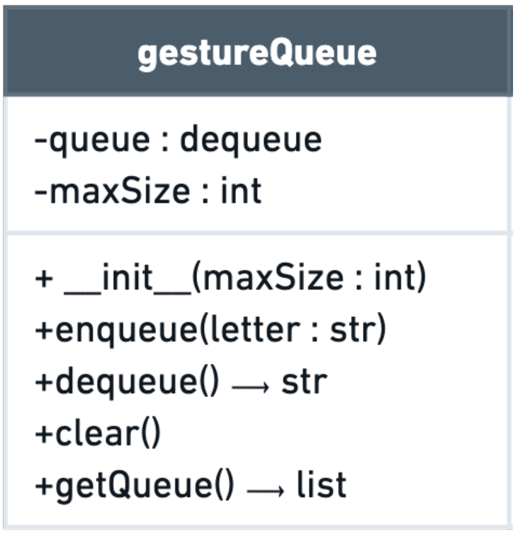


Figure 11 – A class diagram for the `gestureQueue` *class.*

It is key to note that earlier designs for the queue included a linked list implementation of the queue. The implementation for this design can be found in the `linkedListGestureQueue.py` file in the `extraFiles` folder, evident in the directory of the project. However, this implementation was discarded since the added time complexity and space complexity of the pointers, despite being O(1), were still greater than the time complexity of the basic queue used. Since the priority of this phase of the project was efficiency over impressive solutions, the linked list implementation was discarded. However, the implementation is included below in *Code Snippet 5.*

```
class Node:
    def __init__(self, gesture):
        self.gesture = gesture
        self.next = None

class GestureQueue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def enqueue(self, gesture):
        new_node = Node(gesture)
        if not self.tail:
            self.head = self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self.size += 1

    def dequeue(self):
        if self.head is None:
            return None
        gesture = self.head.gesture
        self.head = self.head.next
        if not self.head:
            self.tail = None
        self.size -= 1
        return gesture

    def clear(self):
        self.head = self.tail = None
        self.size = 0

    def getQueue(self):
        current = self.head
        gestures = []
        while current:
            gestures.append(current.gesture)
            current = current.next
        return gestures

    def isEmpty(self):
        return self.size == 0

#Code Snippet 5
```

The final product phase also contains custom algorithms, some of which are discussed below:

<u>Saving the Queue *(Pseudocode Snippet 9)*:</u>

- <u>Goal</u>: To save the sequence of gestures (letters) in the queue to a text file.
- Algorithm:
  1. When the user presses 'S', the current queue is saved to a text file.
  2. The filename is timestamped to ensure unique file names. The datetime module is used to generate the timestamp.
  3. The queue is written to the file as a sequence of letters.
  4. A status message is displayed on the screen indicating that the queue has been saved.

```
FUNCTION saveQueueToFile(self)

timestamp = getCurrentTimestamp()
fileName = "sequenceOfLetters " + timestamp + "txt"

# Open the file in write mode
    OpenFileForWriting(filename):
        # Join all the items in the gesture queue into a single
string
        joined_queue =
JoinQueueItemsIntoString(self.gesture_queue)

        # Write the joined string to the file
        WriteToFile(joined_queue)

    # Print a confirmation message with the filename
    Print("Queue saved to " + filename)

    # Update the status message to show the queue has been saved
    UpdateStatusMessage("Queue saved as " + filename)

    # Set the timestamp for the status message to display for 2
seconds
    SetStatusMessageTimestamp()

ENDFUNCTION

#Pseudocode Snippet 9
```

Clearing the Queue *(Pseudocode Snippet 10)*:

- Goal: To allow the user to clear the gesture queue.
- Algorithm:
  1. When the user presses 'C', the clear method of the `gestureQueue` is called, which clears all elements in the queue.
  2. A status message is displayed on the screen indicating that the queue has been cleared.

```
FUNCTION clear(self)
          self = self.removeAllItemsFromTheQueue()
          RETURN self
ENDFUNCTION

#Pseudocode Snippet 10
```

Displaying the Status Message:

- Goal: To show feedback to the user when actions like saving or clearing the queue are performed.
- Algorithm:
  1. When the user saves the queue, clears the queue, or performs another relevant action, a status message is generated.
  2. The message is displayed for 2 seconds before disappearing. This is achieved by setting a timestamp when the message is first shown and checking the elapsed time during each frame.

Other key algorithms include:

- `showTitle`: displays the program title at the top of the UI window.
- `showQuitMessage`: displays quit instructions at the bottom of the UI window.
- `showPredictions`: displays the class labels and predictions confidences on screen.
- `showGestureQueue`: displays the current gesture queue contents at the bottom of the screen.
- `showStatusMessage`: displays a temporary status message.
- `showError`: displays an error message until error rectified.
- `updateDisplay`: updates the OpenCV window.

2 Design

2.3 User Interface Design

The user interface is completely handled in the final product phase of the project, hence the code is found fully in the `app.py` file, specifically in the `signLanguageTranslatorApp` class.

The reasons behind designing the UI like this are outlined below:

1. Single Responsibility:
The class has a single responsibility: it handles the UI interactions and displays information to the user. This separation ensures the code is modular, and any changes to the UI won't interfere with other parts of the program (like the model or gesture recognition logic). It also makes the code more maintainable, and any future changes (like adjusting the layout, adding new UI components, or tweaking appearance) can be done independently without impacting the core logic.

2. Dynamic Display Update:
The method `updateDisplay()` is key to updating the UI in real time. It allows the application to dynamically reflect changes like:

- The predicted label of the sign language gesture.
- The probabilities of different predictions.
- UI elements like the quit message and title.

These dynamic updates are central to providing feedback to the user in real-time as they perform sign language gestures.

3. Efficient Text Placement:
The method `showPredictions()` manages how predictions are displayed on the screen. The `y_offset` and iterative placement of prediction text help ensure that the UI remains clean and organized, with each class label and probability appearing in its own space.

4. User Guidance:
The UI shows a quit message at the bottom of the window. This is useful for providing instructions to the user, guiding them on how to exit the program. It's an essential usability feature, ensuring the program is user-friendly.

5. Feedback for Real-Time Interactions:
The `signLanguageTranslatorApp` class allows for real-time feedback during gesture recognition. It immediately reflects predictions made by the model, providing the user with an interactive experience. This helps in ensuring that the translation process is seamless and intuitive.

6. Customizability:
The class is designed to be customisable, as seen with parameters like `windowTitle` and `quitMessage`. You can easily modify these to suit different languages, preferences, or themes, making the application flexible for different use cases.

7. Error Handling & Visual Feedback:
Though the `signLanguageTranslatorApp` class itself doesn't handle errors directly, the display can be adapted to show useful information if an error occurs (like if the hands are outside the camera frame, caught using `try, except` statements elsewhere in `app.py`). This enhances the user experience by providing feedback on errors.

8. User-Centric Features:
By showing predictions and probabilities, this class makes the app feel interactive and responsive. The predictions update as the user gestures, allowing for a dynamic experience, as well as an easy testing tool during development. The real-time UI features such as displaying gesture letters and status messages help the user to feel engaged, a purposeful UX feature.

Ensuring the UI is simple and easy to use was the top priority, as can be seen through the wireframe, initial design and initial deployed version of the UI below.

Figure 12 – A wireframe of the UI design for the final product phase

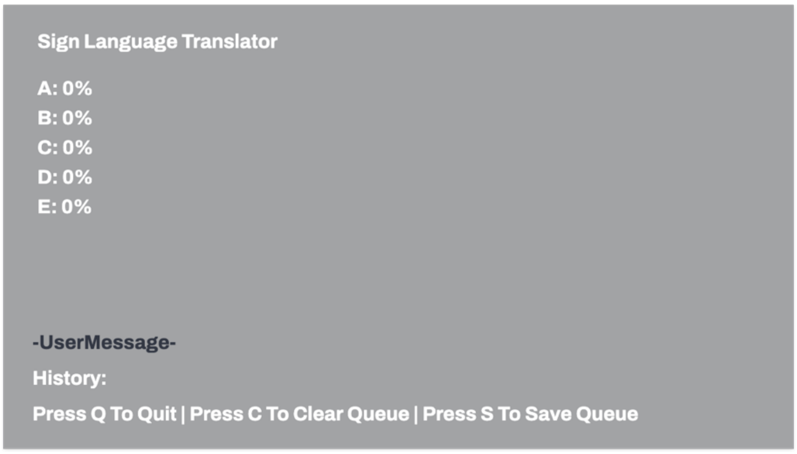


Figure 13 – An initial design of the UI for the final product phase

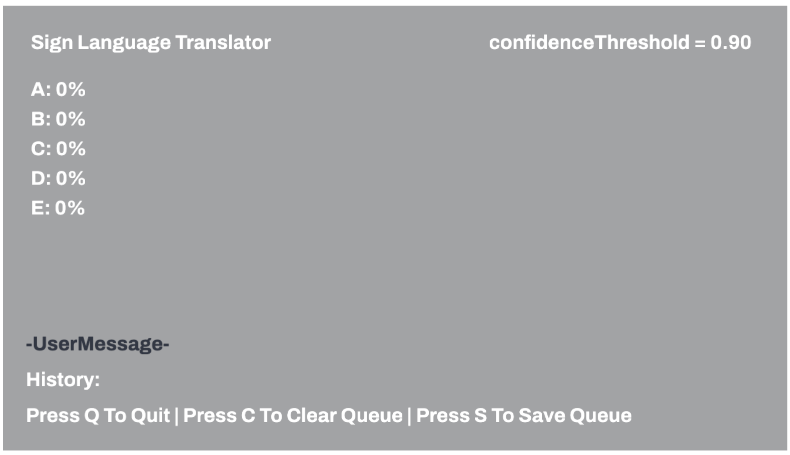


Figure 14 – The initial deployed version of the UI, showing the confidence threshold.

2.4 Log Of Investigative Stages

During this investigation, there were many stages where a mini-investigation into a specific subsection was conducted, and the findings from these mini-investigations then effected the next steps of the wider investigation. Examples of a selection of mini-investigations are shown in *Table 7*.

Date	Mini-Investigation	Approach	Outcome	Next Steps
10/12/24	Tested feasibility of using custom (manual) hand-tracking algorithms instead of OpenCV's Hand Tracking Module	Attempted to manually track hand landmarks using pixel thresholds and contour detection; then trialled OpenCV's built-in module	Proved unreliable under varying lighting and hand angles, whereas OpenCV was more consistent.	Used OpenCv's Hand Tracking module for consistent landmark detection
05/01/25	Compared different image resolutions (300x300 vs 200x200)	Collected datasets at both resolutions & trained identical CNNs	300x300 images achieved 5% higher accuracy on a validation set	Standardised data collection and augmentation to 300x300
12/01/25	Compared RGB vs Grayscale images for CNN training	Converted datasets to grayscale and trained identical CNNs	RGB images improved gesture clarity, particularly for A & E, which are very similar	Used RGB images for training
18/01/25	Investigated CNN architecture depth (2 vs 3 vs 4 layers)	Designed and tested 3 CNN (Models 2, 3 and 4)	Model 3 performed best since Model 2 had too few layers and Model 4 suffered from overfitting	Selected Model 3 as the base (evidence in *Section 7* where the models are labelled), although all models are provided and the user can choose which to use

03/03/25	Implemented the queue data structure as a linked-list	Implemented versions with and without the linked-list structure	Linked list added unnecessary complexity for no gain in time or space complexity (all O(n))	Replaced the linked-list with a standard queue structure
27/03/25	Investigated different UI layouts for model selection, displaying the webcam feed and and predictions	Experimented with various positions, font sizes and window sizes	Layout with top-left populated gave clearest visibility for the majority of users since the majority are right-handed. Additionally model selection was limited to a command line interface due to time restrictions.	This layout was implemented, with the command line interface for model selection being implemented as a separate class to make future changes to a possible GUI easier
01/04/25	Investigated methods of handling Type-2 errors	Introduced the Confidence Threshold and 'Unknown' class.	Reduced Type 2 errors by 12%	Integrated the 'Unknown' class (*REQ19)* and the Confidence Threshold (*REQ16*, *REQ17*, *REQ21*)

Table 7 – A table detailing an investigative log for this investigation.

3 Development

3.1 Tools & Technologies

The technical solution for this project was developed using a combination of Python libraries and external dependencies, all of which are listed in the `requirements.txt` file included in the project repository. These tools were chosen to support efficient computer vision processing, machine learning model management, and user interface development.

The following libraries were used in the final implementation:

- `cvzone`: A high-level OpenCV wrapper designed to simplify real-time computer vision applications, particularly useful for integrating hand tracking and webcam-based UI features.
- `OpenCV (cv2)`: Utilised internally by `cvzone` and directly for additional image processing and display functionality.
- `TensorFlow`: The primary library for defining, training, saving, and loading the Convolutional Neural Network (CNN) used for sign language recognition.
- `scikit-learn`: Used to generate classification reports and performance metrics, including confusion matrices, after model training.
- `NumPy`: Provides efficient handling of large numerical arrays, especially for image data manipulation and matrix operations during data processing and prediction.
- `Matplotlib & Seaborn`: Employed for data visualisation, particularly in plotting model performance metrics such as accuracy, loss curves, and confusion matrices.
- `Pillow`: A Python imaging library used for image manipulation during the data augmentation phase.
- `Mediapipe`: Integrated via `cvzone` for hand tracking and landmark detection from the webcam feed.

<u>Development Environment:</u>

- <u>Python version</u>: 3.11.4
- <u>Development platform</u>: macOS (Apple M1 MacBook Pro, 2020)
- The dependencies are fully defined in the `requirements.txt` file to allow for reproducibility on other systems.

3 Development

3.2 Implementation

The final implementation of this project was split across multiple program files to improve modularity, maintainability, and ease of testing. Each file is responsible for a distinct part of the overall system, allowing for a clear separation between data collection, data augmentation, model creation, training, and application execution.

<u>File Structure:</u>
The project was modularised across multiple Python files to maintain clarity and ease of testing:

- `dataCollection.py`: Captures images from the webcam for dataset creation.
- `dataAugmentation.py`: Performs augmentation on existing images to improve model generalisation.
- `model.py`: Defines the Convolutional Neural Network (CNN) architecture.
- `train.py`: Handles model training, validation, and performance evaluation.
- `app.py`: Runs the main application with the user interface, webcam integration, and real-time prediction display.

The modular file structure and choice of libraries were made to ensure the system could handle real-time hand gesture detection efficiently while maintaining cross-platform compatibility and ease of maintenance.

The file structure described above is shown below:

```
├── dataCollectionAndAugmentation
  └── dataAugmentation.py
  └── dataCollection.py
├── finalProduct
  └── app.py
├── modelCreationAndTraining
  └── model.py
  └── train.py
```

The code includes inline comments throughout to explain the purpose and function of individual lines and blocks of code. In addition to these comments, this section presents annotated program listings of the key program files and algorithms used in the final solution. Where appropriate, significant algorithms and complex program logic are explained in greater detail.

A copy of the entire codebase, including all scripts not covered under *Section 3 Development* (testing, models etc.), is available on GitHub. Click here to access the repository.

3 Development

3.2 Implementation

3.2.1 Data Collection

The script used for data collection is `dataCollection.py`.

```
import cv2
from cvzone.HandTrackingModule import HandDetector
import numpy as np
import math
import time

# Initialise webcam and hand detector
def initialiseCamera():
    cap = cv2.VideoCapture(0)
    detector = HandDetector(maxHands=1)
    return cap, detector

# Capture and process hand image
def captureHandImage(cap, detector, offset, imgSize):
    success, image = cap.read()
    if not success:
        return None, None

    hands, image = detector.findHands(image)
    if hands:
        hand = hands[0]
        x, y, w, h = hand["bbox"]
        imageCrop = image[max(0, y - offset):
min(image.shape[0], y + h + offset + 5),
                          max(0, x - offset):
min(image.shape[1], x + w + offset + 5)]
        return imageCrop, image
    return None, image

# Resize the image to fit the required size
def resizeImage(imageCrop, imageSize, h, w):
    croppedImageWithWhiteBg = np.ones((imageSize,
imageSize, 3), np.uint8) * 255
    aspectRatio = h / w
    if aspectRatio > 1:
        k = imageSize / h
        wCal = math.ceil(k * w)
        imgResize = cv2.resize(imageCrop, (wCal,
imageSize))
        wGap = math.ceil((imageSize - wCal) / 2)
        croppedImageWithWhiteBg[:, wGap:wGap + wCal] =
imgResize
    else:
```

```
        k = imageSize / w
        hCal = math.ceil(k * h)
        imgResize = cv2.resize(imageCrop, (imageSize,
hCal))
        hGap = math.ceil((imageSize - hCal) / 2)
        croppedImageWithWhiteBg[hGap:hGap + hCal, :] =
imgResize
    return croppedImageWithWhiteBg

# Save the image
def saveImage(croppedImageWithWhiteBg, folder, counter):
    try:
        cv2.imwrite(f"{folder}/Image_{time.time()}.jpg",
croppedImageWithWhiteBg)
        counter += 1
        print(f"Image number {counter} saved")
    except Exception as e:
        print(f"Error saving image: {e}")

# Main function
def main():
    cap, detector = initialiseCamera()
    offset = 30
    imageSize = 300
    counter = 0
    folder = 'savedDataModel5/notAugmented/Unknown'

    while True:
        imageCrop, image = captureHandImage(cap,
detector, offset, imageSize)
        if imageCrop is not None:
            imgWhite = resizeImage(imageCrop, imageSize,
*imageCrop.shape[:2])
            cv2.imshow("Image White", imgWhite)

        cv2.imshow("Image", image)
        key = cv2.waitKey(1)
        if key == ord("g"):
            saveImage(imgWhite, folder, counter)
            counter += 1
        if key == ord("q"):
            break

    cap.release()
    cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```

3 Development

3.2 Implementation

3.2.2 Data Augmentation

The script used for data augmentation is `dataAugmentation.py`.

```
# dataAugmentation.py is a Python script that reads
images from a folder, applies data augmentation
techniques, and saves the augmented images to a new
folder.

import os
import cv2
import numpy as np
import time
import math

# Define the path to the data that isn't augmented
dataDir = '/Users/shoubhitabhin/Documents/VSCode
Projects/ALevel/ALevelNEA/JanMMLV3/savedDataModel5/notAu
gmented'

# Ensure augmented data folder exists
augmentedDataDir = os.path.join(dataDir, 'augmented')

if not os.path.exists(augmentedDataDir):
    os.makedirs(augmentedDataDir)

# Loop through each letter folder in the saved data
for letterFolder in os.listdir(dataDir):
    letterPath = os.path.join(dataDir, letterFolder)

    # Skip files like .DS_Store (ERROR ENCOUNTERED WHEN
TESTING)
    if not os.path.isdir(letterPath):
        continue

    # Just for me
    print(f"Processing images for letter:
{letterFolder}")

    for imageName in os.listdir(letterPath):
        imagePath = os.path.join(letterPath, imageName)

        # Skip non-image files
        if not imagePath.lower().endswith(('.png',
'.jpg', '.jpeg')):
            continue
```

```
        # Read the image
        image = cv2.imread(imagePath)
        if image is None:
            continue

        augmentedImages = []

        # Flipping the images
        flippedImage = cv2.flip(image, 1)
        augmentedImages.append(flippedImage)

        # Rotation
        rows, cols = image.shape[:2]
        angle = np.random.uniform(-15,15) # Means the
image is rotated anywhere between 15 degrees and 15
degrees
        M = cv2.getRotationMatrix2D((cols / 2, rows /
2), angle, 1)
        rotatedImage = cv2.warpAffine(image, M, (cols,
rows))
        augmentedImages.append(rotatedImage)

        zoomFactor = np.random.uniform(0.9, 1.1)
        zoomedImage = cv2.resize(image, None,
fx=zoomFactor, fy=zoomFactor)
        zoomedImage = cv2.resize(image, None,
fx=zoomFactor, fy=zoomFactor)
        h, w = zoomedImage.shape[:2]
        zoomedImageThatIsNowPadded = np.zeros((300, 300,
3), dtype=np.uint8)
        xOffset = (300 - w) // 2
        yOffset = (300 - h) // 2
        if xOffset >= 0 and yOffset >= 0:
            zoomedImageThatIsNowPadded[yOffset:yOffset +
h, xOffset:xOffset + w] = zoomedImage
        else:
            # If the zoomed image is bigger than
300x300, resize it down since that is what the model is
expecting
            zoomedImage = cv2.resize(zoomedImage, (300,
300))

augmentedImages.append(zoomedImageThatIsNowPadded)

        # Applies the transformation matrix to the image
using affine warping, which is a special type of
transformation matrix
        horizontalTranslation = np.random.randint(-10,
10)
        verticalTranslation = np.random.randint(-10, 10)
```

```
        shiftingMatrix = np.float32([[1, 0,
horizontalTranslation], [0, 1, verticalTranslation]])
        shiftedImage = cv2.warpAffine(image,
shiftingMatrix, (cols, rows))
        augmentedImages.append(shiftedImage)

        noise = np.random.normal(0, 10,
image.shape).astype(np.uint8)
        noisyImage = cv2.add(image, noise)
        augmentedImages.append(noisyImage)

        """
        Possible reason for failure of B and D since
they are rectangualr in shape, solution from ChatGPT
below
        # Resize to a new size (e.g., 64x64)
        img_resize = cv2.resize(img, (64, 64))
        augmented_images.append(img_resize)
        """

        #  --- The code below is from ChatGPT ---

        # Resize while maintaining aspect ratio and
padding to 300x300
        h, w = image.shape[:2]
        scale = 300 / max(h, w)
        new_w, new_h = int(w * scale), int(h * scale)
        resized_img = cv2.resize(image, (new_w, new_h))

        # Create a blank black image (300x300) and
center the resized image
        padded_img = np.zeros((300, 300, 3),
dtype=np.uint8)
        xOffset = (300 - new_w) // 2
        yOffset = (300 - new_h) // 2
        padded_img[yOffset:yOffset + new_h,
xOffset:xOffset + new_w] = resized_img

        augmentedImages.append(padded_img)

        # --- End of ChatGPT code ---

        # Save augmented images
        for idx, augmentedImage in
enumerate(augmentedImages):
            augmentedImageName =
f"{letterFolder}_{imageName.split('.')[0]}_aug_{idx+1}.j
pg"
            augmentedImagePath =
os.path.join(augmentedDataDir, letterFolder,
augmentedImageName)
```

```
            if not
os.path.exists(os.path.join(augmentedDataDir,
letterFolder)):

os.makedirs(os.path.join(augmentedDataDir,
letterFolder))

            cv2.imwrite(augmentedImagePath,
augmentedImage)

        print(f"Augmented {imageName} and saved
{len(augmentedImages)} new images.")

print("Data augmentation completed!")
```

3 Development

3.2 Implementation

3.2.3 The Model

The script used to define the model is `model.py`.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D,
MaxPooling2D, Flatten, Dense, Dropout

def build_model(input_shape=(300, 300, 3),
num_classes=6): # NOW 6 CLASSES SINCE 'OTHER'
    """
    Builds a CNN model for sign language recognition.

    Improvements (from the one that was used to train
models 1,2,3 and 4) include:
    - Corrected num_classes to match the actual dataset
(A, B, C, D, E → 5 classes). # note now 6 since Unknown
label added
    - Added additional convolutional layers for better
feature extraction.
    - Tuned dropout rates to reduce overfitting.
    - Added batch normalization to stabilize training.
    - Verified softmax activation for multi-class
classification.
    """
    model = Sequential()

    # First convolutional layer: Extract low-level
features
    model.add(Conv2D(32, (3, 3), activation='relu',
input_shape=input_shape))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Second convolutional layer: Deeper feature
extraction
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Third convolutional layer: More complex features
    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Fourth convolutional layer: Increasing feature
complexity
    model.add(Conv2D(256, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
    # Fifth convolutional layer : slightly taller than
is wide so should pick up vertical shapes better
    model.add(Conv2D(32, (5, 3), activation='relu',
input_shape=input_shape))

    # Flatten and Dense layers for classification
    model.add(Flatten())
    model.add(Dense(256, activation='relu'))

    # Fully connected layer with dropout to reduce
overfitting
    model.add(Dropout(0.5))

    # Output layer with softmax for classification
    model.add(Dense(num_classes, activation='softmax'))

    # Compile the model with an appropriate optimizer
and loss function
    model.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy']) #
CHANGED OPTIMISER TO BRITISH SPELLING MAY CAUSE ERROR

    return model
```

3 Development

3.2 Implementation

3.2.4 Training The Model

The script used to train the model is `train.py`.

```
# run with this script always -- python3 -m
modelCreationAndTraining.train

import os
import tensorflow as tf
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping
from modelCreationAndTraining.model import buildModel
from sklearn.utils.class_weight import
compute_class_weight
import numpy as np

class signLanguageTrainer:
    def __init__(self, data_dir, batch_size=32,
epochs=5):
        self.data_dir = data_dir
        self.batch_size = batch_size
        self.epochs = epochs

    def train(self):
        # Only rescales data (no augmentation since this
script uses the augmented data )
        train_datagen =
ImageDataGenerator(rescale=1./255)

        validation_datagen =
ImageDataGenerator(rescale=1./255)

        train_generator =
train_datagen.flow_from_directory(
            os.path.join(self.data_dir, 'train'),
            target_size=(300, 300),
            batch_size=self.batch_size,
            class_mode='categorical'
        )

        validation_generator =
validation_datagen.flow_from_directory(
            os.path.join(self.data_dir, 'evaluation'),
            target_size=(300, 300),
            batch_size=self.batch_size,
            class_mode='categorical'
```

```
        )

        # Print class indices to verify correct encoding
(for personal testing only)
        print("Class indices:",
train_generator.class_indices)

        # Compute class weights to handle potential
class imbalance
        class_weights = compute_class_weight(
            class_weight='balanced',
            classes=np.unique(train_generator.classes),
            y=train_generator.classes
        )
        class_weight_dict =
dict(enumerate(class_weights))

        # Build and compile the model
        num_classes = len(train_generator.class_indices)
        model = buildModel(inputShape=(300, 300, 3),
numClasses=num_classes)

        # Early stopping based on validation accuracy
        early_stopping =
EarlyStopping(monitor='val_accuracy', patience=5,
restore_best_weights=True)

        # Train the model
        model.fit(
            train_generator,
            steps_per_epoch=max(1,
train_generator.samples // self.batch_size),  # Prevent
steps = 0
            epochs=self.epochs,
            validation_data=validation_generator,
            validation_steps=max(1,
validation_generator.samples // self.batch_size),  #
Prevent steps = 0
            callbacks=[early_stopping],
            class_weight=class_weight_dict
        )

        # Save the trained model
        model.save('CHANGETHENAMEOFTHIS.keras') # ADD
MODEL NAME HERE MAKE SURE IT IS DIFFERENT FROM PREVIOUS
MODEL NAMES OR CRASH BANG POW!
        print("Model training complete and saved.")

if __name__ == "__main__":
```

```
    data_dir = '/Users/shoubhitabhin/Documents/VSCode
Projects/ALevel/ALevelNEA/JanMMLV3/savedDataModelWithUnk
nown2/trainOnThese'
    trainer = signLanguageTrainer(data_dir)
    trainer.train()
```

3.2.5 Main Program

The script used to run the main program is `app.py`.

```
import cv2
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import load_model
from cvzone.HandTrackingModule import HandDetector
from PIL import Image
from collections import deque  # For the queue
functionality
import datetime  # For timestamping the file name
import time  # For managing the delay
import os # Allows choosing the model


class modelSelector:
    def __init__(self, modelDirectory):
        self.modelDirectory = modelDirectory

    def listTheModels(self):
        """Retursn a list of available model filenames
from the directory"""
        return [f for f in
os.listdir(self.modelDirectory) if f.endswith(".keras")]

    def selectAModel(self):
        """Prompts the user to select a model and return
its' full path"""
        models = self.listTheModels()
        if not models:
            raise Exception("No .keras models found in
the specified directory.")
        for i, model in enumerate(models):
            print(f"{i+1}. {model}")
        while True:
            try:
                choice = int(input("Select model number:
")) - 1
                if 0 <= choice < len(models):
                    return
os.path.join(self.modelDirectory, models[choice])
                else:
                    print("Invalid selection. Try
again.")
            except ValueError:
```

```
                print("Please enter a valid number.")

    def loadModel(self, modelPath):
        """Load and return the selected model"""
        return load_model(modelPath)

# Initialize model selector and load model
try:
    """bug fix, assign an instance of the modelSelector
class to the variable modelSelectorInstance to make it
clear that we must have an instance of the class named
differently to the class itself"""
    modelSelectorInstance = modelSelector("CNNModels")
    selectedModelPath =
modelSelectorInstance.selectAModel()
    model =
modelSelectorInstance.loadModel(selectedModelPath)
    print(f"Loaded model: {selectedModelPath}")
except Exception as e:
    print(f"Error loading model: {e}")
    exit(1)

# Define class labels (ensure these match the training
class order, so when more letters are added make sure
they match)
classLabels = ['A', 'B', 'C', 'D', 'E', 'Unknown']

class gestureQueue:
    def __init__(self, maxSize: int):
        self.queue = deque(maxlen=maxSize)  # Use deque
for efficient FIFO operations
        self.maxSize = maxSize

    def enqueue(self, letter: str):
        """Add a letter to the queue"""
        self.queue.append(letter)

    def dequeue(self):
        """Remove a letter from the queue"""
        if self.queue:
            return self.queue.popleft()
        return None

    def clear(self):
        """Clear the entire queue"""
        self.queue.clear()

    def getQueue(self):
        """Return the list of letters in the queue"""
        return list(self.queue)
```

```
# Define SignLanguageTranslatorUI class
class signLanguageTranslatorApp:
    def __init__(self, windowTitle: str, quitMessage:
str):
        self.windowTitle = windowTitle
        self.quitMessage = quitMessage
        self.classLabels = classLabels
        self.predicted_label = ""
        self.predictions = []
        self.gestureQueue = gestureQueue(maxSize=10)  #
Initialise the gesture queue with a max size of 10
        self.lastPredictedLabel = None  # Tracks the
last predicted letter added to the queue
        self.stableFrameCount = 0  # Counter to track
how many frames the prediction has been stable
        self.stableFrameThreshold = 10  # Number of
frames the prediction must remain stable before being
enqueued
        self.statusMessage = ""  # For displaying status
messages on the screen
        self.statusTimestamp = None  # Timestamp when
the status message is shown
        self.confidenceThreshold = 0.90 # Initialises
the confidence threshold

        # Initialise the OpenCV window
        cv2.namedWindow(self.windowTitle)

    def showTitle(self, img):
        """Show the title text at the top of the
window"""
        cv2.putText(img, self.windowTitle, (50, 50),
cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2,
cv2.LINE_AA)

    def showQuitMessage(self, img):
        """Show the instructions to quit the program"""
        cv2.putText(img, self.quitMessage, (50,
img.shape[0] - 50), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255,
255, 255), 2, cv2.LINE_AA)

    def showPredictions(self, img):
        """Show the predicted label and probabilities on
the image"""
        y_offset = 100  # Starting point for displaying
predictions
        for i, (label, prob) in
enumerate(zip(self.classLabels, self.predictions)):
            cv2.putText(img, f"{label}:
{prob*100:.2f}%", (50, y_offset + i * 30),
```

```
cv2.FONT_HERSHEY_SIMPLEX, 0.8, (255, 255, 255), 2,
cv2.LINE_AA)

    def showGestureQueue(self, img):
        """Display the gesture queue on the screen"""
        y_offset = img.shape[0] - 100  # Starting point
for displaying the queue
        queue_str = "Queue: " +
"".join(self.gestureQueue.getQueue())  # Get all letters
in the queue
        cv2.putText(img, queue_str, (50, y_offset),
cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2,
cv2.LINE_AA)

    def showStatusMessage(self, img):
        """Display a status message on the screen for 2
seconds"""
        yOffset = img.shape[0] - 150  # Starting point
for displaying the message
        if self.statusMessage and self.statusTimestamp:
            elapsed_time = time.time() -
self.statusTimestamp  # Calculate elapsed time
            if elapsed_time < 2:
                cv2.putText(img, self.statusMessage,
(50, yOffset), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 0),
2, cv2.LINE_AA)
            else:
                self.statusMessage = ""  # Clear message
after 2 seconds
                self.statusTimestamp = None  # Reset
timestamp

    def showError(self, img, message="Error occurred"):
        """Displays a temporary error message on the
screen for 1 second"""
        # Draw the error message in red
        cv2.putText(img, message, (50, 100),
cv2.FONT_HERSHEY_SIMPLEX,
                    1, (0, 0, 255), 2, cv2.LINE_AA)

        # Show the frame with the error message
        cv2.imshow("Sign Language Translator", img)

        # Wait for however long I feel like rn
        cv2.waitKey(10)

    def updateDisplay(self, predicted_label,
predictions, img):
        """Update the displayed information"""
        self.predicted_label = predicted_label
        self.predictions = predictions
```

```
        self.showTitle(img)
        self.showPredictions(img)
        self.showQuitMessage(img)
        self.showGestureQueue(img)  # Display the
gesture queue
        self.showStatusMessage(img)  # Display status
message

    def processPredictedLetter(self, predicted_label):
        """Add predicted letter to the queue if stable
for the defined number of frames"""
        if predicted_label != self.lastPredictedLabel:
            self.stableFrameCount = 0  # Reset counter
if the predicted label changes
            self.lastPredictedLabel = predicted_label
        else:
            self.stableFrameCount += 1

        if self.stableFrameCount >=
self.stableFrameThreshold:
            if predicted_label != 'Unknown':

self.gestureQueue.enqueue(predicted_label)
            else: # If it's Unknown, it will add a space
in the queue
                self.gestureQueue.enqueue('#')
            self.stableFrameCount = 0  # Reset after
enqueuing

    def saveQueueToFile(self):
        """Save the current gesture queue to a text file
with a timestamp"""
        timestamp =
datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
        filename = f"sequenceOfLetters{timestamp}.txt"
        with open(filename, 'w') as f:
            # Join the queue elements into a single
string and write it to the file

f.write("".join(self.gestureQueue.getQueue()))
        print(f"Queue saved to {filename}")
        self.statusMessage = f"Queue saved as
{filename}"  # Update the status message
        self.statusTimestamp = time.time()  # Set the
timestamp for the message


"""Start of the main program logic"""
# Initialise webcam and hand detector

try:
```

```
    cap = cv2.VideoCapture(0)
    detector = HandDetector(maxHands=1)
    if not cap.isOpened():
        raise Exception("Webcam not accessible. Please
check your connection.")
except Exception as e:
    print(f"Error initializing webcam: {e}")
    exit(1)  # Exit if the webcam cannot be accessed

# Initialize the UI class
ui = signLanguageTranslatorApp("Sign Language
Translator", "Press Q to Quit | Press C to Clear Queue |
Press S to Save Queue")

while True:
    success, img = cap.read()
    if not success:
        break

    hands, img = detector.findHands(img, draw=True)

    if hands:
        hand = hands[0]
        x, y, w, h = hand['bbox']

        try:
            # Extract and preprocess the hand region
            hand_img = img[y:y+h, x:x+w]
            hand_img = cv2.resize(hand_img, (300, 300))
            hand_img = cv2.cvtColor(hand_img,
cv2.COLOR_BGR2RGB)  # Convert to RGB
            hand_img = hand_img / 255.0  # Normalize
            hand_img = np.expand_dims(hand_img, axis=0)

            # Predict the sign language letter
            prediction = model.predict(hand_img)
            predicted_class = np.argmax(prediction)
            confidence = np.max(prediction)  # Get the
maximum confidence for the predicted class

            if confidence < ui.confidenceThreshold:
                predicted_label = 'Unknown'  # If
confidence is too low, classify as Unknown
            else:
                predicted_label =
classLabels[predicted_class]  # Otherwise, classify
normally

            probabilities = prediction[0]  # Get
probabilities for all classes
```

```
            # Update the UI with predictions and process
the predicted letter
            ui.processPredictedLetter(predicted_label)
# Check if letter should be enqueued
            ui.updateDisplay(predicted_label,
probabilities, img)
        except Exception as e:
            print(f"Bring your hand in the frame: {e}")
            ui.showError(img, "Bring your hand in the
frame")

    # Get the image width and height
    height, width, _ = img.shape

    # Define the position for the top right (adjust 20
pixels from the right edge)
    x_pos = width - 500  # You can adjust the 250 value
to fit the text nicely
    y_pos = 50  # Keep it at the top of the window

    # Place the Confidence Threshold text at the top
right
    cv2.putText(img, f"Confidence Threshold:
{ui.confidenceThreshold:.2f}",
                (x_pos, y_pos),
cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2,
cv2.LINE_AA)

    # Keyboard controls for adjusting confidence
threshold
    key = cv2.waitKey(1) & 0xFF
    if key == ord('='):  # Increase threshold - PRESS
THE +/= key but CV2 recognises it as the = key
        ui.confidenceThreshold = min(1.0,
ui.confidenceThreshold + 0.05)  # Max threshold is 1.0
        print(f"Threshold increased:
{ui.confidenceThreshold}")
    elif key == ord('-'):  # Decrease threshold
        ui.confidenceThreshold = max(0.0,
ui.confidenceThreshold - 0.05)  # Min threshold is 0.0
        print(f"Threshold decreased:
{ui.confidenceThreshold}")

    # Show the image with updated UI
    cv2.imshow("Sign Language Recognition", img)

    # Exit on 'q' key press
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

    # Clear the queue on 'c' key press
```

```
    if cv2.waitKey(1) & 0xFF == ord('c'):
        ui.gestureQueue.clear()
        ui.statusMessage = "Queue cleared"  # Display
message on screen
        ui.statusTimestamp = time.time()  # Set
timestamp for the message

    # Save the queue to a text file on 's' key press
    if cv2.waitKey(1) & 0xFF == ord('s'):
        ui.saveQueueToFile()
        ui.statusMessage = "File saved"  # Display
message on screen
        ui.statusTimestamp = time.time()

cap.release()
cv2.destroyAllWindows()
```

The annotated program listings presented in this section represent the complete, final implementation of the sign language recognition system. The code has been fully tested, commented, and modularised to support clarity, maintainability, and potential future improvements. While the current implementation meets the specified requirements, opportunities for further enhancement - such as optimising model performance or expanding the system to support additional hand gestures - have been identified and are discussed later in this report. A version-controlled copy of the complete codebase is available via the provided GitHub repository link.

Please Note:

The following sections –

4 Testing

will be a part of the final documentation, but since testing has not been completed yet, it is currently not part of the documentation. The same goes for *Section 6* and *Section 7*.

For further details on the non-machine learning approach, see *Section 5 non-ML Approach.*

5 non-ML Approach

5.1 non-ML Design

While the focus of the investigation is the ML approach, design documents were produced for the non-ML approach and are included below.

The non-machine learning approach will involve using traditional image processing techniques to segment the hand from the background, identify basic geometric features, and classify hand gestures based on predefined rules. The system will classify hand gestures using geometric analysis, such as the number of fingers shown or hand shapes like open, closed, or pointing.

<u>High-Level Architecture:</u>

The system contains the following high-level structure:

- <u>Input Layer</u>: Captures an image from the source path provided, or a frame from the webcam.
- <u>Preprocessing Layer</u>: Converts the image to an appropriate RGB format.
- <u>Hand Landmark Detection</u>: Mediapipe's hand tracking module is responsible for identifying landmarks.
- <u>Feature Extraction & Geometric Analysis</u>: Extracts landmarks and calculates geometric relationships between the landmarks, such as distances, angles and the dot product.
- <u>Gesture Classification</u>: Uses the data from the feature extraction & geometric analysis to classify the image as a letter from the ASL alphabet.
- <u>Visualisation & UI</u>: Displays the landmarks and result using OpenCV.

<u>Data Flow:</u>

Image → Landmark Detection → Landmark Extraction → Distance & Angle Calculations → Finger Extension Classification

There are two primary data structures employed by this program – lists and tuples.

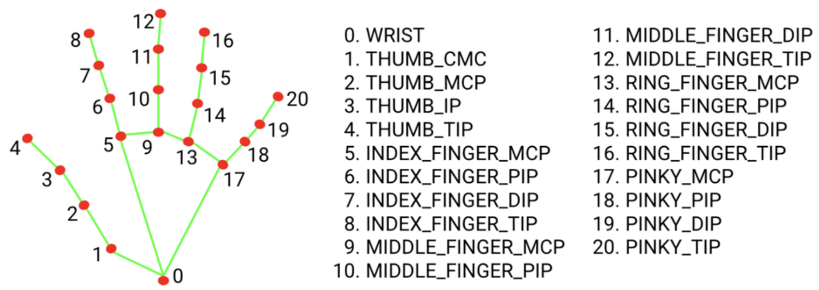
Each landmark is represented as a tuple of floating-point numbers (x,y), indicating normalised coordinates - x and y are normalised to [0.0, 1.0] by the image width and height.
This is shown in *non-ML Code Snippet 1*.

```
landmarks = [(x1, y1), (x2, y2), ..., (x21, y21)]

# The code below is an example of real landmark values of the
first two tuples in the list of landmarks:
[(0.5029321908950806, 0.7681965231895447), (0.554777979850769,
0.6923765540122986) … ]

#non-ML Code Snippet 1
```

A list also contains all the dedicated landmarks for each hand.
This is shown in *non-ML Code Snippet 2*.

```
results = hands.process(image_rgb)
results.multi_hand_landmarks

#non-ML Code Snippet 2
```

The lists are ordered according to the Mediapipe guidelines, in the following order:



non-ML Figure 1 – The standard landmark locations used by the Mediapipe library.

Floats are used for angle and distance calculations, as well as to store the normalised landmark values indicated in *non-ML Code Snippet 1*.

Both geometric and mathematical algorithms have been used in this system.

The Euclidean distance formula is used to measure the distance between two points.
This is shown in *non-ML Pseudocode Snippet 1* and *non-ML Code Snippet 3*.

$$d(p,q) = \sqrt{\sum_{i=1}^{n} \left(q_i - p_q\right)^2}$$

$$\textit{where } p, q \textit{ are two points in Euclidean space}$$
$$q_i, p_i \textit{ are two Euclidean vectors from the origin to points } p \,\&\, q \textit{ respectively}$$
$$n \textit{ is a the } n \textit{ space}$$

```
FUNCTION calculate_distance(point 1, point 2)
     RETURN sqrt((point2.x - point1.x)^2 +   (point2.y -
point1.y) ^2)

#non-ML Pseudocode Snippet 1
```

```
def calculate_distance(point1, point2):
     return math.sqrt((point2[0] - point1[0]) ** 2 + (point2[1] -
point1[1]) ** 2)

#non-ML Code Snippet 3
```

5 non-ML Approach

5.2 non-ML Development

While the focus of the investigation is the ML approach, the non-ML implementation was coded, and is included below.

```
import cv2
import mediapipe as mp
import math

# Initialize Mediapipe Hand Detection
mp_hands = mp.solutions.hands
hands = mp_hands.Hands(static_image_mode=True,
max_num_hands=1)
mp_draw = mp.solutions.drawing_utils

def preprocess_image(image_path):
    """Load and preprocess the image for landmark
extraction."""
    image = cv2.imread(image_path)
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = hands.process(image_rgb)
    return image, results

def extract_landmarks(results):
    """Extract hand landmarks and return a list of
landmark points."""
    if results.multi_hand_landmarks:
        landmarks = []
        for hand_landmarks in
results.multi_hand_landmarks:
            for point in hand_landmarks.landmark:
                landmarks.append((point.x, point.y))
        return landmarks
    return None

def calculate_distance(point1, point2):
    """Calculate Euclidean distance between two
points."""
    return math.sqrt((point2[0] - point1[0]) ** 2 +
(point2[1] - point1[1]) ** 2)

def is_finger_extended(landmarks, tip_index, dip_index,
pip_index):
    """
    Check if a finger is extended using relative y-
coordinates.
```

```
    The finger is extended if the tip is above both DIP
and PIP.
    """
    return (landmarks[tip_index][1] <
landmarks[dip_index][1] < landmarks[pip_index][1])

def is_thumb_extended(landmarks):
    """
    Enhanced thumb detection by measuring both
horizontal spread and proximity to the palm.
    """
    thumb_tip = landmarks[4]
    thumb_ip = landmarks[3]
    thumb_mcp = landmarks[2]
    wrist = landmarks[0]
    index_base = landmarks[5]

    tip_mcp_distance = calculate_distance(thumb_tip,
thumb_mcp)
    tip_index_distance = calculate_distance(thumb_tip,
index_base)

    def calculate_angle(a, b, c):
        ab = (a[0] - b[0], a[1] - b[1])
        cb = (c[0] - b[0], c[1] - b[1])
        dot_product = ab[0] * cb[0] + ab[1] * cb[1]
        mag_ab = math.sqrt(ab[0]**2 + ab[1]**2)
        mag_cb = math.sqrt(cb[0]**2 + cb[1]**2)
        angle = math.acos(dot_product / (mag_ab * mag_cb
+ 1e-6))
        return math.degrees(angle)

    thumb_angle = calculate_angle(wrist, thumb_mcp,
thumb_tip)

    if tip_mcp_distance > 0.2 and tip_index_distance >
0.2 and thumb_angle > 30:
        return True
    else:
        return False

def count_extended_fingers(landmarks):
    """Count the number of extended fingers excluding
the thumb."""
    return sum([
        is_finger_extended(landmarks, 8, 6, 5),
        is_finger_extended(landmarks, 12, 10, 9),
        is_finger_extended(landmarks, 16, 14, 13),
        is_finger_extended(landmarks, 20, 18, 17)
    ])
```

```
def identify_letter(landmarks):
    """Identify ASL letters based on optimized geometric
rules."""

    print("Processing landmarks:", landmarks)
    thumb_extended = is_thumb_extended(landmarks)
    extended_fingers = count_extended_fingers(landmarks)

    # A: Thumb extended slightly, all fingers folded
    if thumb_extended and extended_fingers == 0:
        return "A"
    # B: All fingers extended straight up, thumb folded
across palm
    elif not thumb_extended and extended_fingers == 4:
        return "B"
    # C: All fingers curved forming a 'C' shape,
measured using thumb to index distance
    elif thumb_extended and extended_fingers == 4:
        index_thumb_distance =
calculate_distance(landmarks[4], landmarks[8])
        if index_thumb_distance > 0.1:
            return "C"
    # D: Index finger extended, thumb touching index
forming circle
    elif extended_fingers == 1 and
calculate_distance(landmarks[4], landmarks[8]) < 0.05:
        return "D"
    # E: All fingers folded into the palm with thumb
folded
    elif not thumb_extended and extended_fingers == 0:
        return "E"
    # F: Thumb and index touching, other fingers
extended
    elif extended_fingers == 3 and
calculate_distance(landmarks[4], landmarks[8]) < 0.05:
        return "F"
    # G: Thumb and index extended horizontally
    elif extended_fingers == 1 and thumb_extended and
landmarks[8][1] > landmarks[6][1]:
        return "G"
    # H: Index and middle extended horizontally, thumb
folded
    elif extended_fingers == 2 and thumb_extended and
landmarks[8][1] > landmarks[6][1]:
        return "H"
    # I: Only pinky extended
    elif extended_fingers == 1 and landmarks[20][1] <
landmarks[18][1]:
        return "I"
    # J: Same as I, but motion-tracing (J motion not
detectable here)
```

```
    elif extended_fingers == 1 and landmarks[20][1] <
landmarks[18][1]:
        return "J"
    # K: Index and middle extended in a V shape with
thumb extended
    elif extended_fingers == 2 and thumb_extended:
        return "K"
    # L: Thumb and index forming an L shape
    elif extended_fingers == 1 and thumb_extended:
        return "L"
    # M: Thumb under three folded fingers
    elif extended_fingers == 0 and
calculate_distance(landmarks[4], landmarks[8]) < 0.05:
        return "M"
    # N: Thumb under two folded fingers
    elif extended_fingers == 0 and
calculate_distance(landmarks[4], landmarks[8]) < 0.05:
        return "N"
    # O: Thumb and index forming a circular shape
    elif extended_fingers == 0 and
calculate_distance(landmarks[4], landmarks[8]) < 0.05:
        return "O"
    # P: Thumb and index touching, index pointing
downwards
    elif extended_fingers == 1 and thumb_extended:
        return "P"
    # Q: Similar to G but pointing downwards
    elif extended_fingers == 1 and thumb_extended and
landmarks[8][1] > landmarks[6][1]:
        return "Q"
    # R: Index and middle crossed
    elif extended_fingers == 2 and
calculate_distance(landmarks[8], landmarks[12]) < 0.05:
        return "R"
    # S: Fist with thumb across fingers
    elif extended_fingers == 0 and not thumb_extended:
        return "S"
    # T: Thumb between index and middle
    elif extended_fingers == 0:
        return "T"
    # U: Index and middle extended together
    elif extended_fingers == 2 and
calculate_distance(landmarks[8], landmarks[12]) > 0.05:
        return "U"
    # V: Index and middle forming a V shape
    elif extended_fingers == 2 and
calculate_distance(landmarks[8], landmarks[12]) < 0.1:
        return "V"
    # W: Three fingers extended
    elif extended_fingers == 3:
        return "W"
```

```
    # X: Index bent, others folded
    elif extended_fingers == 1 and landmarks[8][1] >
landmarks[6][1]:
        return "X"
    # Y: Thumb and pinky extended
    elif extended_fingers == 1 and landmarks[20][1] <
landmarks[18][1]:
        return "Y"
    # Z: Index tracing a Z shape (motion not detected)
    return "Unknown"

def main(image_path):
    image, results = preprocess_image(image_path)
    landmarks = extract_landmarks(results)


    if landmarks:
        extended_fingers =
count_extended_fingers(landmarks)
        thumb_extended = is_thumb_extended(landmarks)
        predicted_letter = identify_letter(landmarks)

        cv2.putText(image, f"Extended Fingers:
{extended_fingers}", (10, 50),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (255,
0, 0), 2)
        cv2.putText(image, f"Thumb Extended: {'Yes' if
thumb_extended else 'No'}", (10, 100),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (255,
0, 0), 2)
        cv2.putText(image, f"Predicted Letter:
{predicted_letter}", (10, 150),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0,
0), 2)

        for lm in results.multi_hand_landmarks:
            mp_draw.draw_landmarks(image, lm,
mp_hands.HAND_CONNECTIONS)
        cv2.imshow("Hand Detection", image)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
    else:
        print("No hand detected.")

if __name__ == "__main__":
    image_path = "/Users/shoubhitabhin/Downloads/ASL/B-
ASL.jpg"
    main(image_path)
```

5 non-ML Approach

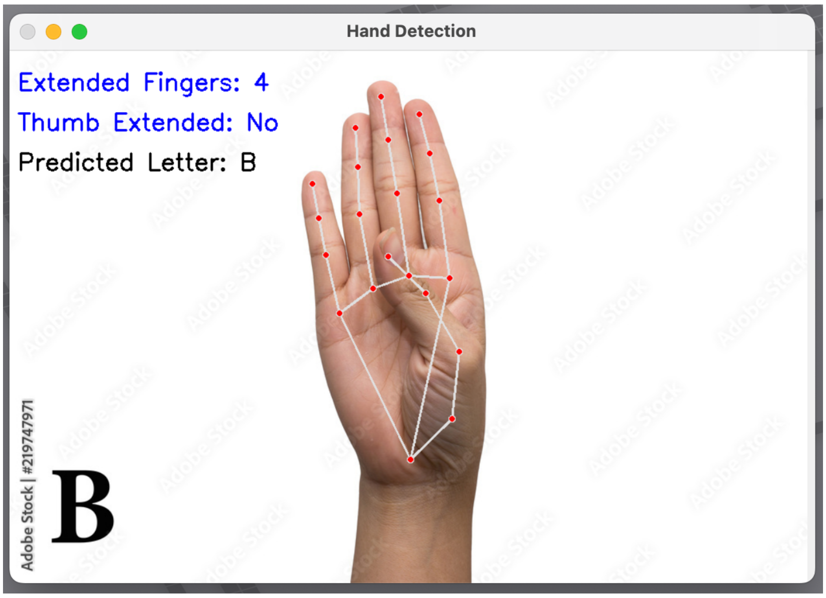
5.3 non-ML Testing

While the focus of the investigation is the ML approach, the non-ML implementation was tested, and evidence is included below.

Evidence of the script being run:

```
(venv) shoubhitabhin@MacBookPro JanMMLV3 % python TEMPORARYno-ML/noMLProc.py
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1744940865.213166  994608 gl_context.cc:369] GL version: 2.1 (2.1 Metal - 89.3), renderer: Apple M1
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
W0000 00:00:1744940865.231013  994737 inference_feedback_manager.cc:114] Feedback manager requires a model with a single signature inference. Disabling supp
ort for feedback tensors.
W0000 00:00:1744940865.239946  994737 inference_feedback_manager.cc:114] Feedback manager requires a model with a single signature inference. Disabling supp
ort for feedback tensors.
W0000 00:00:1744940865.264792  994737 landmark_projection_calculator.cc:186] Using NORM_RECT without IMAGE_DIMENSIONS is only supported for the square ROI.
Provide IMAGE_DIMENSIONS or use PROJECTION_MATRIX.
Processing landmarks: [(0.5029321908950806, 0.7681965231895447), (0.554777979850769, 0.6923765540122986), (0.5632495880126953, 0.566041111946106), (0.521628
4990310669, 0.45701542496681213), (0.4744774401187897, 0.38791003823280334), (0.5514802932739258, 0.4283495545387268), (0.5389792323112488, 0.28201684355735
78), (0.5263333320617676, 0.1948993057012558), (0.5139045119285583, 0.12021832168102264), (0.5002976655960083, 0.42355218529701233), (0.4857034981250763, 0.
2692089080810547), (0.47457507252693176, 0.1687663048505783), (0.4658416211605072, 0.08716291934251785), (0.45544669032096863, 0.44751933217048645), (0.4387
403130531311, 0.30878686904907227), (0.4363771677017212, 0.21963828802108765), (0.43373510241508484, 0.14647021889686584), (0.4136923849582672, 0.4935213923
4542847), (0.3964064121246338, 0.38510310649871826), (0.387382447719574, 0.31504836678504944), (0.37963682413101196, 0.2508634626865387)]
2025-04-18 02:47:45.650 Python[13198:994608] +[IMKClient subclass]: chose IMKClient_Modern
2025-04-18 02:47:45.650 Python[13198:994608] +[IMKInputSession subclass]: chose IMKInputSession_Modern
```

Evidence of output:



The unit test that was run:

```python
import unittest
from noMLProc import identify_letter,
calculate_distance, is_thumb_extended,
count_extended_fingers
# Should be the letter ‘B’
class TestASLFunctions(unittest.TestCase):

    def setUp(self):
        self.mock_landmarks_B = [
            (0.5, 0.8), #Wrist
            (0.45, 0.7),  # Thumb CMC
            (0.45, 0.6), #Thumb MCP:
            (0.45, 0.5), # Thumb IP
            (0.45, 0.4),  # Thumb tip
            (0.55, 0.7),   # Index MCP
            (0.55, 0.5),  # Index PIP
            (0.55, 0.3),  # Index DIP
            (0.55, 0.2),  # Index tip
            (0.6, 0.7), # Middle MCP
            (0.6, 0.5),  # Middle PIP
            (0.6, 0.3),  # Middle DIP
            (0.6, 0.2),   # Middle tip
            (0.65, 0.7),   # Ring MCP
            (0.65, 0.5),  # Ring PIP
            (0.65, 0.3),  # Ring DIP
            (0.65, 0.2),  # Ring tip
            (0.7, 0.7), # Pinky MCP
            (0.7, 0.5),  # Pinky PIP
            (0.7, 0.3),   # Pinky DIP
            (0.7, 0.2),   # Pinky tip

        ]

    def test_identify_letter_B(self):
        """Test the identification of the letter 'B'."""
        print("Test landmarks:", self.mock_landmarks_B)
        result = identify_letter(self.mock_landmarks_B)
        print("Result:", result)
        self.assertEqual(result, 'B', f"Expected 'B',
but got {result}")


if __name__ == '__main__':
    unittest.main()
```

```
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1744941126.411483 1003649 gl_context.cc:369] GL version: 2.1 (2.1 Metal - 89.3), renderer: Apple M1
Test landmarks: [(0.5, 0.8), (0.45, 0.7), (0.45, 0.6), (0.45, 0.5), (0.45, 0.4), (0.55, 0.7), (0.55, 0.5), (0.55, 0.3), (0.55, 0.2),
 (0.6, 0.7), (0.6, 0.5), (0.6, 0.3), (0.6, 0.2), (0.65, 0.7), (0.65, 0.5), (0.65, 0.3), (0.65, 0.2), (0.7, 0.7), (0.7, 0.5), (0.7, 0
.3), (0.7, 0.2)]
Processing landmarks: [(0.5, 0.8), (0.45, 0.7), (0.45, 0.6), (0.45, 0.5), (0.45, 0.4), (0.55, 0.7), (0.55, 0.5), (0.55, 0.3), (0.55,
 0.2), (0.6, 0.7), (0.6, 0.5), (0.6, 0.3), (0.6, 0.2), (0.65, 0.7), (0.65, 0.5), (0.65, 0.3), (0.65, 0.2), (0.7, 0.7), (0.7, 0.5), (
0.7, 0.3), (0.7, 0.2)]
Result: B
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

The second unit test that was run:

```
import unittest
from noMLProc import identify_letter,
calculate_distance, is_thumb_extended,
count_extended_fingers
# Should be the letter ‘E’
class TestASLFunctions(unittest.TestCase):

    def setUp(self):
        self.mock_landmarks_E = [
            (0.5, 0.7),   # Wrist
            (0.58, 0.58), # Thumb CMC
            (0.6, 0.4), # Thumb MCP
            (0.5, 0.3), # Thumb IP
            (0.42, 0.28), # Thumb Tip
            (0.6, 0.3), # Index MCP
            (0.6, 0.1), # Index PIP
            (0.58, 0.1), # Index DIP
            (0.57, 0.25), # Index Tip
            (0.5, 0.28),    # Middle MCP
            (0.5, 0.08),    # Middle PIP
            (0.6, 0.1),    # Middle DIP
            (0.5, 0.23),    # Middle Tip
            (0.45, 0.3),  # Ring MCP
            (0.43, 0.1),  # Ring PIP
            (0.44, 0.13),  # Ring DIP
            (0.45, 0.23),  # Ring Tip
            (0.39, 0.3),   # Pinky MCP
            (0.37, 0.17),   # Pinky PIP
            (0.4, 0.17),    # Pinky DIP
            (0.4, 0.24)     # Pinky Tip
        ]


    def test_identify_letter_E(self):
        """Test the identification of the letter 'E'."""
        print("Test landmarks:", self.mock_landmarks_E)
        result = identify_letter(self.mock_landmarks_E)
        print("Result:", result)
        self.assertEqual(result, 'E', f"Expected 'E',
but got {result}")


if __name__ == '__main__':
    unittest.main()
```

```
(venv) shoubhitabhin@MacBookPro JanMMLV3 % python TEMPORARYno-ML/unitTestLetterE.py
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1744941295.980450 1007815 gl_context.cc:369] GL version: 2.1 (2.1 Metal - 89.3), renderer: Apple M1
Test landmarks: [(0.5, 0.7), (0.58, 0.58), (0.6, 0.4), (0.5, 0.3), (0.42, 0.28), (0.6, 0.3), (0.6, 0.1), (0.58, 0.1), (0.57, 0.25),
(0.5, 0.28), (0.5, 0.08), (0.6, 0.1), (0.5, 0.23), (0.45, 0.3), (0.43, 0.1), (0.44, 0.13), (0.45, 0.23), (0.39, 0.3), (0.37, 0.17),
(0.4, 0.17), (0.4, 0.24)]
Processing landmarks: [(0.5, 0.7), (0.58, 0.58), (0.6, 0.4), (0.5, 0.3), (0.42, 0.28), (0.6, 0.3), (0.6, 0.1), (0.58, 0.1), (0.57, 0
.25), (0.5, 0.28), (0.5, 0.08), (0.6, 0.1), (0.5, 0.23), (0.45, 0.3), (0.43, 0.1), (0.44, 0.13), (0.45, 0.23), (0.39, 0.3), (0.37, 0
.17), (0.4, 0.17), (0.4, 0.24)]
Result: E
```