UNIVERSITY OF CAMBRIDGE

Ronan Ragavoodoo
rdr32

Sign Language Recognition

Computer Science Tripos – Part II

Homerton College

May 12, 2023

Declaration

I, Ronan Ragavoodoo of Homerton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT.

I, Ronan Ragavoodoo of Homerton College, am content for my dissertation to be made available to the students and staff of the University.

Ronan Ragavoodoo	May 12, 2023
Signed	Date

Acknowledgements

Writing this dissertation required not only hard work and dedication but also help and encouragement from many individuals. First of all, I would like to express my gratitude to Mr. Filip Svoboda for supervising the project and pointing me to all the right resources. His encouragement and academic excellence have helped greatly whilst writing my dissertation. I am also grateful to Dr. John Fawcett for providing feedback on drafts of this dissertation and for helping me grow both academically and personally throughout my years at University. Lastly, I would like to thank my friends and family for reading drafts of this dissertation and for supporting my studies in Cambridge.

Proforma

Candidate Number:	**2405E**
College:	**Homerton College**
Project Title:	**Sign Language Recognition**
Examination:	**Computer Science Tripos – Part II, May 2023**
Word Count:	**9772**[1]
Code Line Count:	**1120**[2]
Project Originator:	The Candidate
Supervisor:	Mr. Filip Svoboda

Original Aims of the Project

The original aim of this project was to create a model that can help recognize British Sign Language (BSL), specifically isolated signs. This model would be able to accurately interpret and translate BSL gestures and movements into written or spoken language, making communication with individuals who use BSL more seamless and accessible. The goal was to develop a program that can assist individuals who are deaf or hard of hearing, as well as improve overall accessibility in society. The project would be focused on utilizing machine learning and computer vision techniques to train the model to recognize BSL gestures with high accuracy.

Work Completed

I have successfully implemented a model that can recognise with high-accuracy a variety of isolated BSL expressions in real-time. Different models have been evaluated to compare performances and determine which is more adequate for real-time recognition.

Special Difficulties

None.

[1]This word count was computed using texcount -inc rdr32.tex

[2]This line count was computed using GitHub's in-built line count and summing the individual files

Contents

Chapter 1

Introduction

British Sign Language (BSL) is a visual language that is used by around 150,000 people in the UK[1], many of whom are deaf or hard of hearing. With the advent of technology and accessibility features being introduced, the need for reliable systems to aid individuals who rely on sign language as a means of communication is clear. As the field of sign language recognition is still very much an area of open-research, there are not many systems which are optimised for sign language use. Sign language recognition can be grouped into two distinct groups: isolated sign language and continuous sign language. While isolated sign language involves distinct gestures being performed one at a time, continuous signing involves many gestures performed in quick succession, often merging some signs together.

Work has been conducted on the BOBSL[1] dataset, which is the largest continuous BSL dataset. This data has been scraped from existing footage of programs from the British Broadcasting Corporation (BBC), specifically those using sign language gestures. Despite having the BOBSL dataset, BSL does not have many suitable isolated sign language datasets.

Outside of BSL, there exists other isolated sign language datasets. The MS-ASL dataset [2] is one of the largest datasets for isolated sign language recognition. The study which yielded it also proposed I3D as a novel way of classifying sign gestures. The LSA64 dataset [3] is another dataset which has been used for evaluating models such as SPOTER [4].

However, existing datasets are often unsuited for specific models. Lack of enough training data for certain labels is common, due to the sheer number of them present in each different variation of sign language. This is true for the dataset used in this project, BSLDict [5]. As such, a model which performs efficiently using low amounts of training data is desired. The model should also be optimised to perform in real-time, as such a use case would more practical for sign language users. Some models are also optimised for specific variations of sign language, due to inherent features such as static signing compared to moving ones, and the number of hands involved. The possibility of using other datasets is explored in this project. A basic model used for basic human-action recognition serves as the base case, and sign language recognition is an extension of it.

1.1 Project Overview and Contributions

The motivation behind this project is to improve communication accessibility for individuals who use BSL. Despite its prevalence, BSL is not widely understood or recognized by the general

[1] https://bda.org.uk/help-resources/

population, making communication with individuals who use BSL difficult and often frustrating. Additionally, the project aims to raise awareness and understanding of BSL, and to demonstrate the power of technology to promote accessibility and inclusivity.

The task at hand can be described as follows: Given an input of data representing a sign language gesture in a format such as videos or coordinates, output the most probable sign that could have been performed. We wish to provide a pipeline which can achieve this using a low amount of training data and that can be used in real-time detection.

In this project, I aim to provide a solution by contributing the following:

- Implementing a reliable sign language recognition model which can accurately recognise different sign language gestures in real-time, specifically isolated gestures.
- Choosing reliable datasets to conduct train the implemented model.
- Verify the accuracy of the model and conduct evaluation of its performance on a range of testing data under different conditions.
- Obtaining feedback from users about the experience of using the model to recognise performed sign gestures.
- Explore different possible extensions that could lead to further work, including possible use cases for continuous sign language and more efficient algorithms.

Chapter 2

Preparation

This chapter gives a background of current research on various sign-language recognition based models, as well as existing datasets, libraries, and techniques. This forms part of the research conducted prior to writing the implementation code for the project, but will also include further research that has been done in response to challenges encountered during the implementation.

2.1 Requirements Analysis

The task at hand requires a sign language gesture as an input, represented in a suitable format such a video or coordinates. The output should be a prediction of which sign gesture was used as an input.

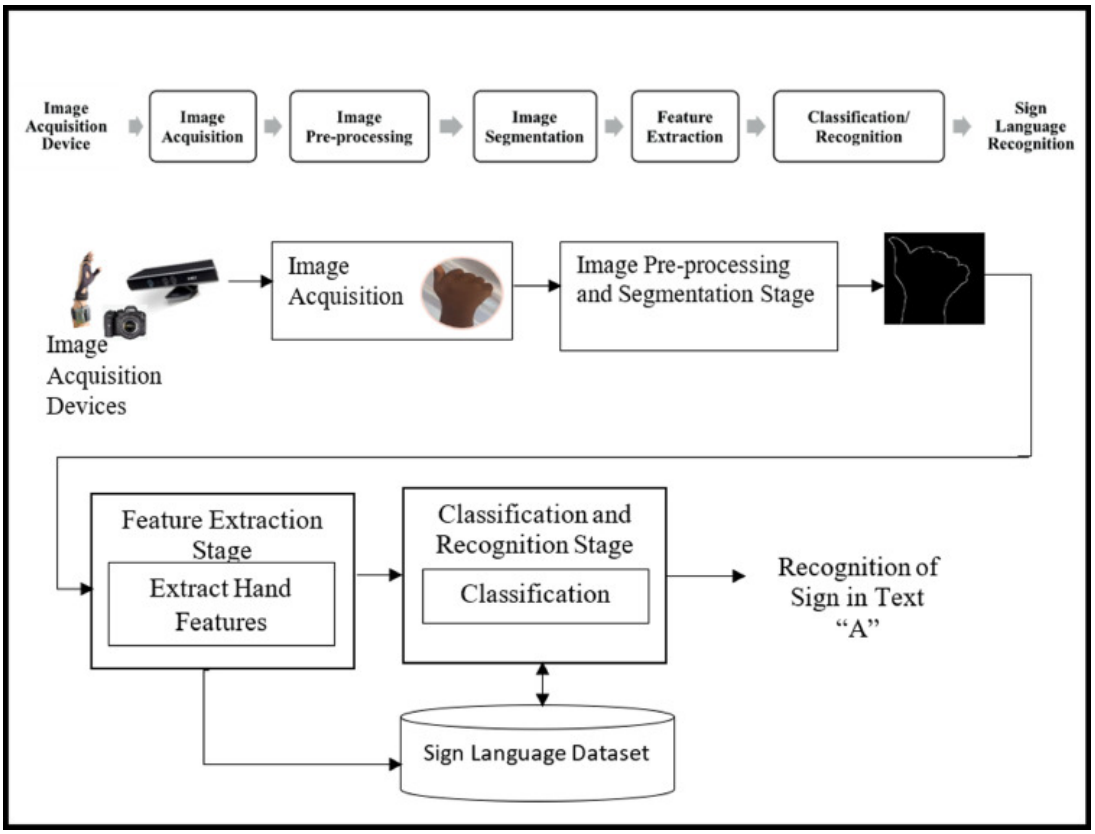

Figure 2.1: Example pipeline of a sign language recognition model

The project requires the following to be implemented in order to have a working program:

- **Gesture Detection and Tracking**: This is what will determine the shape of the training data and its nature. It consists of identifying and tracking the different body parts present in the given frame, especially hands.

- **Gesture Recognition**: There also needs to be a framework which can accurately predict what gesture has been displayed, from a set of different gestures.

- **Real-time Detection**: The program should allow the gestures to be detected in real-time, giving immediate results.

An appropriate dataset is used to perform any form of training or comparison on. Specifically, we target an isolated BSL dataset. However, the nature of sign language in general means that we can expand on different variations of sign language in order to infer how well the model will perform.

2.2 Datasets

BSLDict

BSLDict is a large dataset of isolated BSL gestures [5]. The data has been obtained from BSL sign aggregation platform *signbsl.com*, with a total of 14k video clips for a vocabulary of 9k labels.

The data provides three main issues:

- **Lack of examples per label:** A lot of the labels have only one or two videos. This is a problem as there is not enough data for testing or to use deep learning for efficient training.

- **Different signs per label:** Many labels contain ambiguous sign gestures. Some of the labels have different ways to represent them and some overlap with other labels as well. This leads to poor accuracy when trying to train on them.

- **Ambiguous number of hands:** Many labels contain sign gestures where it is unknown how many hands are used. Some signs have the hand resting within the frame, which can cause issues for models which depend heavily on the position of both hands.

The issue of lacking examples was one reason why DTW was used as an algorithm for this project. As deep learning models require a sufficient amount of training data to be sufficiently accurate, we use a model based on DTW instead as it can retain accuracy with less data.

Figure 2.2: Examples from the BSLDict dataset

LSA64

LSA64 is a sign database for isolated Argentinian Sign Language (LSA) gestures [3]. It was created to produce a dictionary for LSA and to be used for training automatic sign recognition models. The dataset contains 3200 videos, with 10 signers executing 5 repetitions of 64 different types of signs.

This dataset was used mostly for initial testing purposes given the issues that were encountered with BSLDict. It provides both sufficient data and consistent signs per label.

Signers wore coloured gloves to aid hand recognition and segmentation, as well as wearing dark clothing. This does not affect this project due to the use of Mediapipe to recognise signs.

Figure 2.3: Snapshot from the LSA64 dataset

2.3 Model Selection

2.3.1 Neural Networks

Studies have been conducted on many different neural network architectures, focused on mainly on image classification. The best performing networks include 3D Convolutional Neural Networks (3D CNNs), such as I3D, and Recurrent Neural Networks (RNNs), such as Long Short-Term Memory (LSTM) networks. I3D has shown remarkable performance in sign language recognition tasks due to its ability to capture spatiotemporal information from videos. It has been used in various sign language recognition projects, such as recognizing American Sign Language (ASL) gestures, notably on the MS-ASL dataset [2]. RNNs, particularly LSTMs, are also commonly used for sign language recognition due to their ability to handle sequential data. These networks have been used in projects aimed at recognizing both isolated gestures [6] and continuous sign language sentences [7].

2.3.2 Dynamic Time Warping

Dynamic Time Warping (DTW) is a well-known algorithm for comparing sequences that may have different lengths and may be slightly misaligned. It was first introduced by Sakoe and Chiba in 1978 [8] and has since been widely used in many areas such as speech recognition, pattern recognition, bioinformatics, and finance. The idea to use this algorithm for sign language recogntion was inspired from content in the paper by Huu et al. [9].

The basic idea of DTW is to find the optimal alignment between two sequences by warping one of the sequences in the time dimension. The algorithm constructs a distance matrix between the two sequences, where each element represents the distance between a point in the first sequence and a point in the second sequence. The optimal path through this matrix is then found using dynamic programming, which minimizes the total distance between the two sequences.

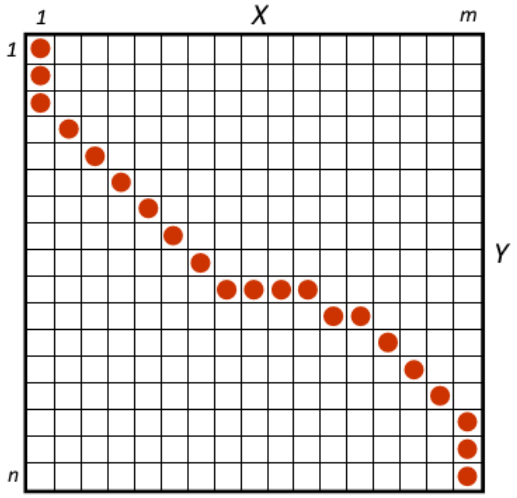

Figure 2.4: Matrix with warp path

Let $X = x_1, x_2, ..., x_m$ and $Y = y_1, y_2, ..., y_n$ be two sequences of length m and n, respectively. The algorithm aims to find the optimal path $\pi = (i_1, j_1), (i_2, j_2), ..., (i_k, j_k)$ that warps one of

the sequences (usually the shorter one) to align with the other sequence. The optimal path π is the one that minimizes the total distance between the two sequences, which is defined as:

$$DTW(X,Y) = \min_{\pi} \sqrt{\sum_{(i,j)\in\pi} (x_i - y_j)^2}$$

where (i, j) represents a point in the distance matrix.

To find the optimal path π, the DTW algorithm constructs a distance matrix D of size $m \times n$, where each element $D_{i,j}$ represents the distance between x_i and y_j. The matrix is initialized with large values such that $D_{i,j} = \infty$ for all i and j. The first element is set to $D_{1,1} = (x_1 - y_1)^2$. Then, for each element $D_{i,j}$, the algorithm finds the minimum distance among the three neighboring elements:

$$D_{i,j} = (x_i - y_j)^2 + \min(D_{i-1,j}, D_{i,j-1}, D_{i-1,j-1})$$

The path through the distance matrix that corresponds to the optimal path π is then found by backtracking from the bottom-right corner of the matrix to the top-left corner, following the minimum distance at each step. The resulting path is the optimal path π, which can be used to align the two sequences. The time and space complexity of the algorithm are in the quadratic order. In the distance matrix, each cell is filled in exactly once, with each being filled in constant time. That leads to a time and space of complexity of $O(m \times n)$.

Algorithm 1 is a simple implementation of the DTW algorithm in pseudocode.

Algorithm 1: Simple DTW algorithm

Input : X: array $[x_1, x_2, \ldots, x_m]$, Y: array $[y_1, y_2, \ldots, y_n]$
Output: Distance: float

```
dp = matrix of size m + 1 × n + 1
for i ← 1 to m do
    for j ← 1 to n do
        dp[i, j] = inf
    end for
end for
dp[0, 0] = 0
for i ← 1 to m do
    for j ← 1 to n do
        cost = abs(X[i − 1] − Y[J])
        dp[i, j] = cost + min(dp[i − 1, j], dp[i, j − 1], dp[i − 1, j − 1])
    end for
end for
return dp[m, n]
```

DTW has been shown to be an effective algorithm for comparing sequences in many applications, but it has some limitations. One of the main limitations is its computational complexity, which is quadratic in the length of the sequences. This can be a problem for very long sequences or for applications that require real-time processing. In recent years, there have been many extensions and variations of DTW that aim to overcome some of these limitations [10].

2.3.3 Feature Extraction

Feature extraction is a crucial step in sign language recognition systems as it enables the conversion of raw video data into meaningful and informative representations that can be used for classification purposes. In this section, we discuss the feature extraction process used in our sign language recognition project, which involves the extraction of hand and body pose landmarks and the calculation of hand angles and relative distance features.

The project employed Mediapipe Holistic, an open-source library developed by Google, to extract landmarks from the hand and body poses in the sign language videos. This library provides a robust and accurate solution for the detection of body and hand keypoints, including wrist, elbow, shoulder, and finger joints. Specifically, we used the pre-trained Holistic model, which uses a deep neural network to detect the keypoints and estimate their positions in real-time. More details about Mediapipe are described in both Section 2.4.3 and Section 3.3.

From the extracted landmarks, angles between each of the 21 hand landmarks were calculated for each hand. This resulted in 441 angles, which captured the intricate hand movements and positions that are characteristic of sign language gestures. These angles were calculated using the law of cosines and were normalized to ensure consistency across different signers and videos.

In addition to hand angles, the relative distance between the wrist and shoulder landmarks was also calculated for each hand. This feature provides information about the arm extension, which is a critical aspect of sign language gesture recognition. The distance feature was calculated as the Euclidean distance between the wrist and shoulder landmarks and was also normalized to ensure consistency.

These methods were used instead of relying on the raw landmark data as distance and positioning greatly affect the accuracy of the results. Angles and relative distances are less sensitive to these factors, thus allowing the model to be accurate even if users are positioned differently or are further away.

2.3.4 Alternative Data-Capturing Devices

Aside from an image feed, there are alternative ways to capture data about hand and body positioning. One sign language recognition paper [11] used the Leap Motion controller [1], which reports data in a similar format to Mediapipe. There are also similar studies which have been done using other devices, such as the Microsoft Kinect [12], and the Cyberglove [13]. The technology can provide more reliable data than images, due to using sensors rather than machine-learning to estimate positions. However, it is sufficient to demonstrate that the model works on a specific format of data, rather than be limited by the device capturing the data. Further work can be undertaken to obtain more accurate results using 3D sensors instead. Moreover, the cost of purchasing such devices is unsuitable for the purpose of a Part II project.

[1] http://www.leapmotion.com

2.4 Tools and Libraries

Python

Python is a popular high-level programming language that is widely used in scientific computing, data analysis, and machine learning applications. Python was chosen as the main language for this project for the following reasons:

Python has a large and active community of developers, which has contributed to a vast ecosystem of libraries and frameworks that can be used for machine learning and computer vision applications. Some of the popular libraries include NumPy, Pandas, Matplotlib, OpenCV, and Scikit-learn. These libraries provide a range of functionalities, such as data manipulation, visualization, and model development, which can significantly simplify and speed up the development of the project.

Moreover, Python is known for its simplicity and ease of use, making it an ideal language for rapid prototyping and experimentation. The syntax of Python is straightforward and easy to learn, making it an excellent choice for beginners who are new to programming. Additionally, Python's rich standard library provides a wide range of built-in functions and modules that can be used to perform various tasks, such as file I/O, networking, and regular expressions.

2.4.1 Visual Studio Code

Visual Studio Code (VSCode) is a popular and versatile code editor that is widely used for software development, including for Python-based projects. I chose this editor for the following reasons:

VSCode has excellent support for Python programming. It comes with a wide range of built-in features and extensions that facilitate Python development, including code highlighting, linting, debugging, and auto-completion. Additionally, VSCode has a rich and active extension marketplace, where users can access and install a wide range of third-party extensions that enhance its capabilities for Python development.

VSCode provides an intuitive and user-friendly interface that makes it easy to navigate and manage the project code. The built-in file explorer allows for quick navigation through project files and directories, while the integrated terminal provides a convenient way to execute commands and scripts directly within the editor, without needing an external terminal.

Another advantage of VSCode is its support for Git, a version control system that is commonly used in software development projects. With VSCode, we can easily integrate Git into our project workflow, allowing us to manage and track changes to the codebase, collaborate with other developers, and maintain project versions, all done in a very user-friendly environment.

Note that Github Copilot, an AI-powered auto-completion tool, has been used partly in this project while writing the code. This mostly involves autocompleting similar code and finishing standard code fragments required for running the program.

2.4.2 OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning library that provides a wide range of functions for image and video processing, object detection, feature extraction, and more. OpenCV was chosen for the following reasons:

OpenCV supports various programming languages, including Python, C++, and Java, making it a versatile library that can be used in different environments and applications. The project is implemented in Python, and OpenCV provides an intuitive and easy-to-use Python interface that simplifies the development process and speeds up the implementation of the sign language recognition model.

It also has built-in support for hardware acceleration, which can be useful given that the project requires real-time performance as a requirement for the success criteria.

Finally, it is one of the most popular libraries for this purpose, being used by a lot of projects in the field of computer vision. It is well documented and has a lot of support available on its website.

2.4.3 Mediapipe-Holistic

Mediapipe Holistic is an open-source library developed by Google that provides a suite of real-time, multi-person pose estimation and tracking solutions. It uses machine learning models to detect and track multiple key points on the human body, including facial landmarks, hand landmarks, and body pose, which can be used for a wide range of applications, including sign language recognition. In this section, we discuss why Mediapipe Holistic is suitable for this project.

Mediapipe Holistic provides a simple and easy-to-use API that abstracts away the complexity of training and deploying machine learning models for pose estimation and tracking. It comes with pre-trained models that can accurately detect and track facial landmarks, hand landmarks, and body pose, which significantly reduces the development time and cost of building a detection model from scratch.

The model is also designed for real-time performance, which is crucial for this project. It provides efficient algorithms that can process video streams in real-time and accurately track multiple body parts simultaneously, which is essential for detecting and recognizing sign language gestures in real-world scenarios.

Additionally, Mediapipe Holistic is an open-source library, which means that it is free to use, distribute, and modify. This open-source nature provides flexibility and enables us to customize and extend the library to suit the specific requirements of the recognition project.

2.4.4 dtw-python

The `dtw-python` library is a Python implementation of the DTW algorithm, which provides a set of functions for computing the DTW distance between two time series data sequences. The library provides an efficient implementation of the DTW algorithm that can handle sequences of different lengths and allows for the customization of the distance function used to compute the DTW distance.

2.4.5 FastDTW

FastDTW is a variant of the DTW algorithm that trades accuracy for faster computation times [14]. It works by reducing the number of elements that need to be computed in the distance matrix, resulting in a smaller memory footprint and faster computation times.

The FastDTW algorithm consists of three main steps: a coarsening step, a projection step, and a refinement step.

The `fastdtw` library in Python provides an implementation of the FastDTW algorithm, as well as other variants of the DTW algorithm. It is useful for this project given that there will be real-time detection involved, so a faster algorithm would speed up the predictions.

2.4.6 Git and Github

Git was used for version control as this is the system that I am most comfortable with using and familiar with. The repository is hosted on Github and changes were pushed regularly. A copy of this dissertation is also available on Github.

Chapter 3

Implementation

In this chapter, the overall implementation of the project will be described, mentioning the challenges faced along the way. The implementation pipeline consists of the following key processes:

- **Recording signs:** This includes the code which uses OpenCV to capture frames. They also integrate most of the functionality for determining when to process a given sequence of frames, when to reset the recording, and displaying visual and textual output to the user.
- **Pre-processing datasets:** These scripts help process the datasets into a suitable format for the program to read them. LSA64 has one script which does the entire processing step, whereas BSLDict has an additional script to filter labels that have less than a desired amount of data.
- **Extracting features from the datasets:** This is the code that is used to obtain the relevant features that will be used to train our model. Mediapipe holistic is used to first extract landmarks from video frames, then to convert these to angles and relative distances. This is discussed further in the relevant following section.
- **Comparing signs:** This part of the pipeline focuses on the computation using the DTW algorithm. It includes the code which returns the distances between any two compared signs and also the functionality to determine if any given sign is output with a certain confidence.

There are many scripts which when combined together, achieve those processes. The project makes use of objects to make the code reusable and to easily be able to expand it.

The implementation of this project is based on an existing implementation[1]. Methods to integrate pose recognition and to process the datasets have been added to make the model more accurate, and alternative similarity algorithms were also used.

Licensing

Below is a table showing what licenses that software and libraries involved in this project use:

[1] https://github.com/gabguerin/Sign-Language-Recognition–MediaPipe-DTW

Software	License
Python 3.10.5	PSFL
NumPy	PSFL
pandas	PSFL
Mediapipe	Apache 2.0
dtw-python	GNUv2+
fastdtw	MIT
gabguerin's implementation	MIT

3.1 Repository Overview

This section provides an overview of the source code and how the scripts are organised in the repository. The project has been organised such that it is easy to add new datasets and to train on them, provided a new script to process it is written.

- `datasets` ..... Folder to store datasets
- `models`
  - `hand_model.py` ..... § 3.4.2 **71 lines**
  - `pose_model.py` ..... § 3.4.2 **48 lines**
  - `sign_model.py` ..... § 3.4.2 **76 lines**
- `utils`
  - `bsldict_data_processing.py` ..... § 3.2 **37 lines**
  - `bsldict_preprocessing.py` ..... § 3.2 **38 lines**
  - `compute_dtw.py` ..... § 3.5.1 **56 lines**
  - `compute_fastdtw.py` ..... § 3.5.2 **45 lines**
  - `constants.py` ..... § 3.10.1 **13 lines**
  - `feature_extraction.py` ..... § 3.4 **173 lines**
  - `lsa64_data_processing.py` ..... § 3.2 **42 lines**
  - `mediapipe_utils.py` ..... § 3.3 **67 lines**
  - `performance.py` ..... § 4.5 **37 lines**
  - `sign_eval.py` ..... § 3.8 **116 lines**
  - `sign_recorder.py` ..... § 3.7 **120 lines**
  - `webcam_manager.py` ..... § 3.9 **71 lines**
- `eval.py` ..... Script to run test data evaluation **64 lines**
- `main.py` ..... Script to run real-time detection **46 lines**

3.2 Data Pre-processing

The model needs the data to be in a required format in order to process it. As such, we need to pre-process existing datasets in order to make them suitable.

We order the videos by having a directory for each label, containing all videos which fall under that corresponding label. This is illustrated as follows:

```
train_videos
├── Accept
│   ├── Accept-0.mp4
│   └── Accept-1.mp4
└── Appear
    ├── Appear-0.mp4
    └── Appear-1.mp4
```

The scripts `bsldict_data processing.py` and `lsa64_data_processing.py` process their respective datasets to obtain it in such a format.
`bsldict_preprocessing.py` is a script used if we wish to filter labels which have more than a minimum number of videos. For example, it can be used to filter the dataset by getting all labels which have at least 10 videos.

3.3 Mediapipe Extraction

Using our datasets, we now need to extract information from them. This is done by processing each frame in a given video individually.

3.3.1 Mediapipe Holistic

Mediapipe Holistic is the model that outputs the needed data to start the feature extraction process.

Mediapipe Holistic utilizes a deep neural network model that is trained on a large dataset of labeled images and videos. The model is designed to recognize various human poses, hand gestures, and facial landmarks, and it achieves this by analyzing the pixels of the input image or video stream [15]. Mediapipe Holistic provides three tracking pipelines, namely for the face, hands, and pose. The latter two are of relevance to this project.

3.3.2 Mediapipe Pipeline

One problem with using multiple separate models is that each is optimised for their specific purposes [15]. For example, the pose estimation model requires a 256×256 input video frame. However, cropping the hand and face from this frame to feed their respective models results in an input which has a too low resolution to obtain accurate landmarks.

To solve this, Mediapipe Holistic uses a multistage pipeline. First, the human pose is estimated using the pose pipeline. Using the inferred pose landmarks, it then derives the regions of interest (ROI) for the face and hands, employing a recrop model along the way to improve it. The ROIs are then passed into the respective models to obtain the remaining landmarks.

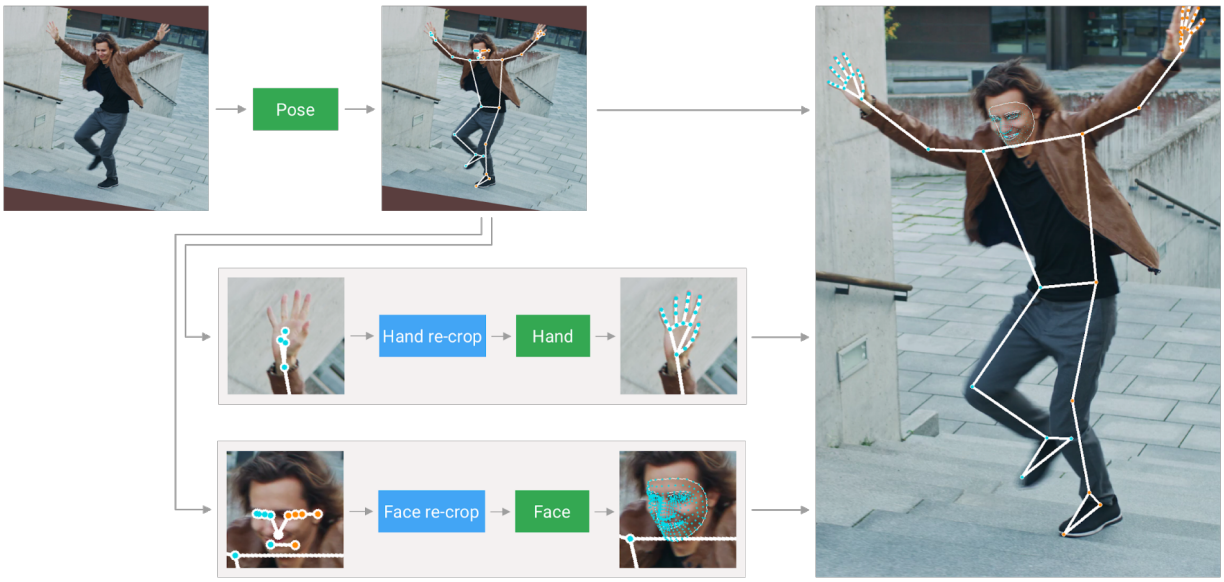

Figure 3.1: Pipeline Overview (Image from mediapipe.dev)

3.3.3 Hands Detection Pipeline

Detecting hands with articulated fingers is a complex task, due to factors such as varying hand spans relative to image frames. Unlike faces, hands lack high-contrast areas, increasing the complexity of detecting them. As such, usage of other features such as arms and the torso can help localise hand positions more accurately.

Mediapipe employs palm detection, as boundary-estimation is relatively easier compared to hands with articulated fingers. A Single-Shot Detector model called BlazePalm is used to detect initial hand locations. An encoder-decoder feature extractor is used for bigger scene context awareness, an approach similar to RetinaNet. Further explanation about these methods are described below.

Single-Shot Detection

Single Shot MultiBox Detector (SSD) is a state-of-the-art object detection algorithm that was proposed by Liu et al. in 2016 [16]. SSD is a single-shot detector, which means that it can detect objects in a single forward pass through the network. This makes SSD much faster than two-stage detectors like R-CNN and Fast R-CNN, which require two forward passes through the network.

SSD works by predicting a set of bounding boxes and confidence scores for each object in an image. The bounding boxes are predicted using a set of anchor boxes, which are predefined boxes that are placed at different locations and scales in the image. The confidence scores are predicted for each anchor box, indicating how likely it is that the anchor box contains an object.

After the bounding boxes and confidence scores have been predicted, they are used to generate a set of final detections. This is done by first applying non-maximum suppression (NMS) to the bounding boxes. NMS is a technique that is used to remove duplicate detections.

In NMS, the bounding boxes are sorted by their confidence scores, and then the boxes that have a high overlap with other boxes are removed.

After NMS has been applied, the remaining boxes are then classified using a softmax classifier. The softmax classifier outputs a probability distribution over the set of object classes. The class with the highest probability is then assigned to the box.

SSD has been shown to be very effective for object detection. It has achieved state-of-the-art results on a number of benchmark datasets, including the PASCAL VOC dataset [17] and the COCO dataset [18]. SSD is also very fast, making it suitable for real-time object detection applications.

Non-Maximum Suppresion

Non-maximum suppression (NMS) is a popular technique used in computer vision applications to eliminate duplicate or redundant detections of the same object in an image. It is commonly used in object detection tasks such as pedestrian detection, face detection, and vehicle detection. The algorithm works by selecting the detection with the highest confidence score and suppressing any overlapping detections that have a lower confidence score.

The NMS algorithm can be described mathematically as follows:

Given a set of bounding boxes, $B = b_1, b_2, ..., b_n$, and their corresponding confidence scores, $S = s_1, s_2, ..., s_n$, where n is the number of bounding boxes, the goal is to select a subset of bounding boxes, $B^* \subseteq B$, that have a high confidence score and do not overlap significantly with each other.

First, we sort the bounding boxes in descending order of their confidence scores. Let b_i be the bounding box with the highest confidence score, i.e., $s_i = \max_{j=1}^{n} s_j$. We add b_i to the subset B^* and remove it from the set B.

Next, we calculate the Intersection over Union (IoU) between b_i and each of the remaining bounding boxes $b_j \in B$, where $j \neq i$. The IoU is defined as:

$$\text{IoU}(b_i, b_j) = \frac{\text{area}(b_i \cap b_j)}{\text{area}(b_i \cup b_j)}$$

If $\text{IoU}(b_i, b_j) >$ threshold, where threshold is a predetermined threshold value (e.g., 0.5), we remove b_j from the set B. We repeat this process until all the bounding boxes in B have been processed.

The final output is the set B^* of selected bounding boxes.

The NMS algorithm helps to reduce the number of false positives and improves the accuracy of object detection models by removing redundant detections.

3.3.4 Pose Detection Pipeline

The process for pose detection, BlazePose, is similar to that used for hands [19]. The BlazePose detection pipeline consists of a lightweight pose detector followed by a pose estimation component.

Pose Detection

The first stage of the BlazePose pipeline is body part detection. Unlike the hands detection pipeline, BlazePose does not use the NMS algorithm as it breaks down in cases where poses are

highly articulated. This is because multiple bounding boxes satisfy the IoU threshold. Instead, a face detector is used as a proxy for the person detector. This is augmented with additional person alignment parameters such as the middle point between the person's hip. The CNN used in the face detector is based on the BlazeFace architecture [20], which uses a modified MobileNetV1 model with depth-wise separable convolutions.

Pose Estimation

The pose estimation component predicts the location of the 33 landmarks, using the bounding boxes that were obtained from the first stage.

To achieve accurate pose estimation, a combined heatmap, offset, and regression approach is adopted. During training, the heatmap and offset loss are used to supervise the model, but the corresponding output layers are removed before running inference. This approach effectively uses the heatmap to supervise a lightweight embedding, which is then used by the regression encoder network. This approach is inspired by the Stacked Hourglass approach of Newell et al. [21], but in this case, a tiny encoder-decoder heatmap-based network is stacked with a subsequent regression encoder network.

To achieve a balance between high- and low-level features, skip-connections are actively utilized between all the stages of the network. However, the gradients from the regression encoder are not propagated back to the heatmap-trained features. This approach not only improves the heatmap predictions but also substantially increases the coordinate regression accuracy.

3.3.5 Mediapipe Implementation

The goal here is to obtain the position of the different connections between joints.
`mediapipe_utils` provides methods which help in locating the landmarks given a frame. The detection is relatively straightfoward to implement, using the provided Holistic model that is imported as part of the library. We can just use the model's `process` method. We also have to do color conversion because frames from OpenCV use BGR color rather than RGB. To add visual representation to the frames, the method `draw_landmarks` can be used to add drawing specifications to display the landmarks and the connections between them.

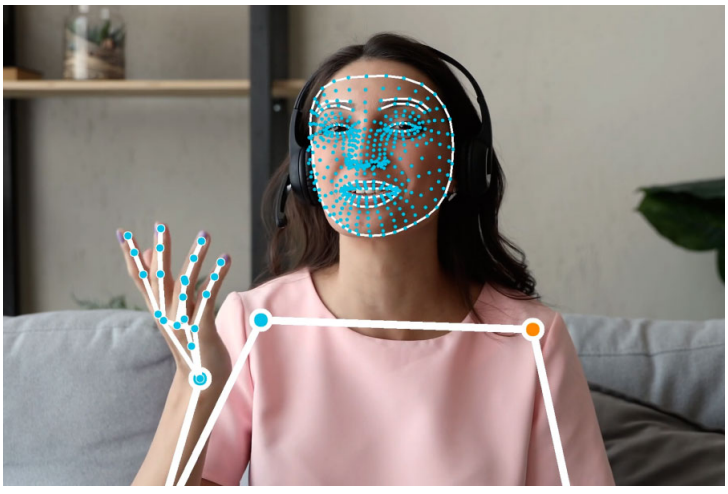

Figure 3.2: Visual display of landmarks of Mediapipe Holistic (Image from mediapipe.dev)

3.4 Features and Reference Signs Representation

Once a list of connections from Mediapipe has been obtained, we want to store this data in memory in a suitable format and be able to reload them. This will serve as the raw training data. The `feature_extraction` script provides a range of methods to do so. The initial training stage involves obtaining a collection of reference signs. These are what the recorded signs will be compared to to evaluate similarity.

3.4.1 Extract to Pickle Files

Initially, the `extract_features` method is used to obtain the landmarks from the data in our training folder. Every video is scanned and the relevant data extracted by Mediapipe is stored in pickle files.

We use this opportunity here to also extract the data of the laterally-inverse of the frames. The reason for this is to obtain the features of the sign performed in a left-handed manner. In BSL, the leading hand used to perform the signing determines the handedness of the sign. Thus, flipping the image is sufficient to obtain the relevant data.

Each training video has two entries corresponding to it: its own extracted data and its flipped data. Each entry contains a pickle file for each hand and the pose, which is essentially an array of landmarks.

3.4.2 Models

The project uses three models to represent features extracted from the vectors provided by Mediapipe Holistic.

Hand Model

The `hand_model` class creates objects that store the feature vector representing all the angles between the connections provided by Mediapipe on an arbitrary hand. Mediapipe provides 21 connections per hand, resulting in a feature vector of $21 * 21 = 441$ angles. We use the dot product formula to obtain the angle given two vectors (See Figure 3.2):

$$\theta = cos^{-1}\Big(\frac{a \cdot b}{|a||b|}\Big), \text{ where } a \text{ and } b \text{ represent connections}$$

The model only represents one particular frame, given a sequence of them. The final sign gesture is thus represented by a sequence of hand models, one for each hand.

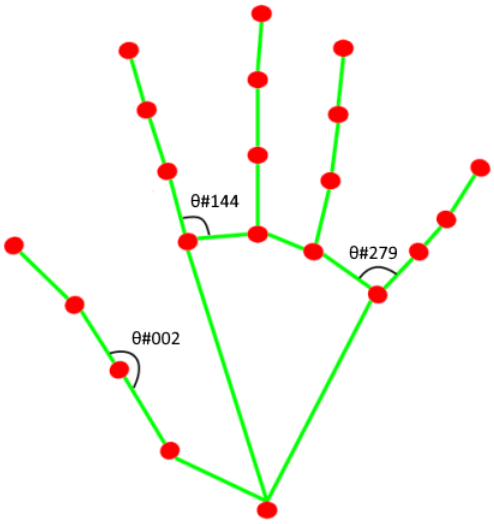

Figure 3.3: The angles on the Mediapipe connection drawing

Pose Model

The `pose_model` takes a different approach. Instead of using angles, we use the distance between the shoulder and elbow as a normalisation distance, normalising coordinates provided by Mediapipe. This is because the pose landmarks also include those of the lower body, including the legs. Hence, we only use the landmarks of the shoulder, elbows and wrist, as they are more likely to be present in the frames. Taking a and b as the position vectors representing the shoulder and elbow respectively, we obtain the new coordinates as follows:

$$v_i = \frac{u_i}{|b - a|}, \text{ for every coordinate } u \text{ in the pose landmarks}$$

Sign Model

The `sign_model` object stores the hand and pose models for each frame in the sequence representing the sign, essentially being the representation of the sign as its features. This is what is compared when computing the similarity between a reference sign and a recorded sign.

Each object contains the following attributes:

- `has_left_hand` and `has_right_hand`, which are booleans which are true if such hand is present in the video.
- `lh_embedding` and `rh_embedding`, which are the lists of feature vectors for each frame for each hand.
- `left_arm_embedding` and `right_arm_embedding`, which are the lists of feature vectors for each frame for each arm.

3.4.3 Loading Reference Signs

To perform predictions, the reference data stored (Section 3.4) needs to be loaded for the session. The `load_reference_signs` method is used to iterate through the entries containing

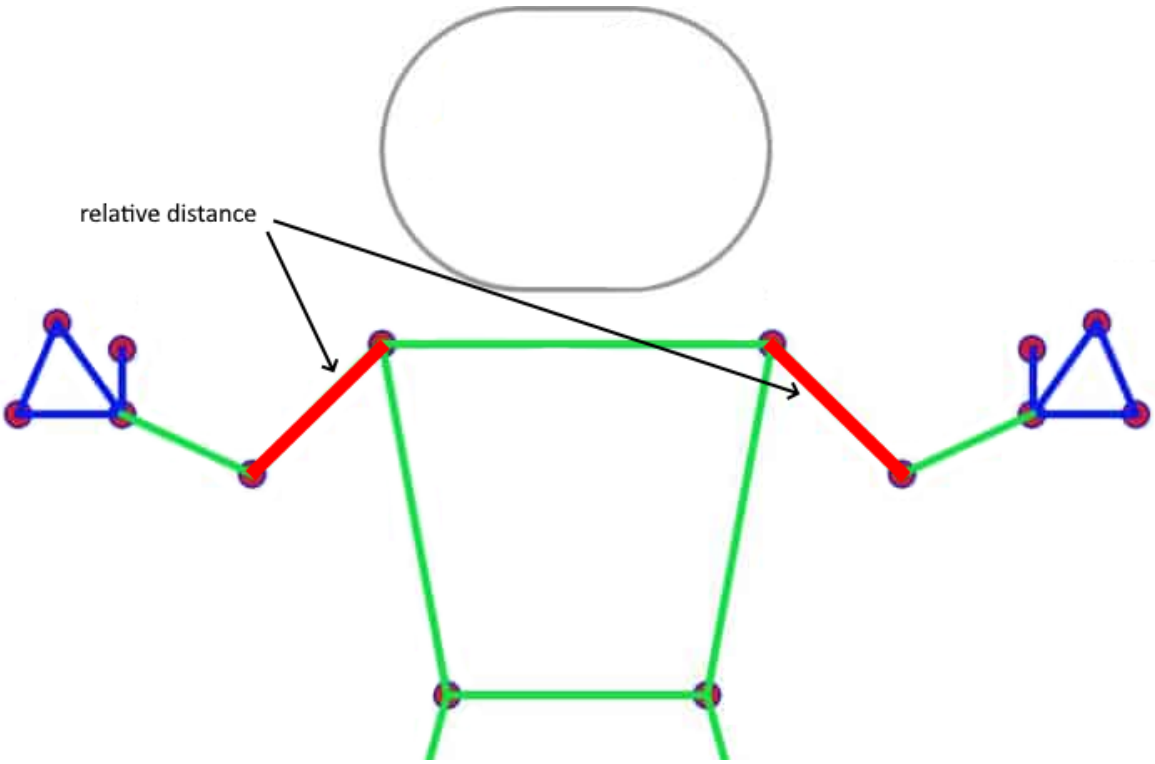

Figure 3.4: The relative distance used to normalise the landmarks

the pickle files. For each entry, we initialise a new `sign_model` object, which is itself initialised with the respective `hand_model` and `pose_model` objects constructed from the data stored in the pickle files.

For efficiency, we use a `pandas.DataFrame` to store the name of the sign, its respective `sign_model`, and a distance initialised to 0. This will be overwritten when computing the distances.

3.5 Computing Similarity

The project provides two scripts for computing similarity: `compute_dtw` and `compute_fastdtw`, which use the `dtw-python`[22] and `fastdtw` [14] libraries respectively. The project provides an easy way to toggle between the two algorithms.

3.5.1 dtw-python

The outline of the DTW algorithm is mentioned in Section 2.3.2. The implementation of the algorithm uses the `dtw` method from the `dtw-python` library, without any additional arguments. The package allows windowing to be used, which can speed up the algorithm, but an alternative algorithm, FastDTW, is proposed later instead.

The `dtw_distances` method is what fills the `pandas.DataFrame` with distances. It takes the DataFrame as an argument along with the sign model representing the recorded sign. The embeddings of the sign model are extracted to variables as they are the sequences to be compared.

The model starts iterating through every sign present in the DataFrame. It discards those

which do not have the same hands present in the recorded sign and sets their distances to infinity in the distance column of the DataFrame. The `dtw` method from `dtw-python` [22] is then called with the respective embeddings of the reference and recorded sign as arguments. The distance obtained is then set in the corresponding entry of the DataFrame.

Challenges Faced

The `dtw` method handles the case of having a sequence of length 1 differently. In such a case, it transposes the array. This happened in a few instances on training due to some signs registering a particular hand in only one frame. This is dependent on the confidence tracking of Mediapipe. As such, this specific case is handled by duplicating the singular entry present in the sequence to obtain an array of length 2.

3.5.2 fastdtw

A high-level overview of the FastDTW algorithm is given in Section 2.4.5. This section will provide a detailed overview of the algorithm, as outlined in the paper proposing it [14] and also cover the implementation, which was done using the `fastdtw` library.

Speeding up DTW

FastDTW uses ideas from two approaches to speed up the DTW algorithm.

Constraints are used to limit the number of cells that are present in the distance matrix. For example, the Sakoe-Chiba band [8] limits the cells to a band shape and the Itakura parallelogram [23] limits them to a parallelogram with corners at the start and end of the distance matrix.

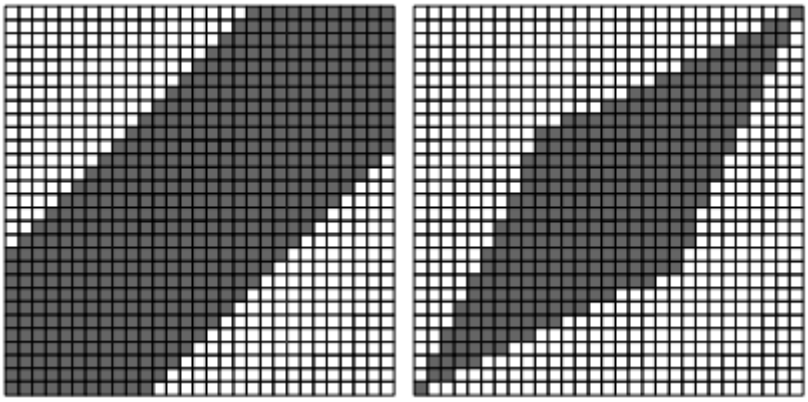

Figure 3.5: Sakoe-Chiba Band (left) and Itakura Parallelogram (right)

Only the shaded cells are filled in by the DTW algorithm. We determine the window, or width, of the shaded area using a parameter to the algorithm.

This method does not guarantee that the optimal global alignment will be found, as it only limits the valid warping paths to those within the constraints of the window.

Using constraints improves the run-time of the algorithm by a constant factor, but it still runs in $O(m \times n)$ if the size of the window is a function of the length of the input time series.

They work best where the optimal global alignment is expected to be close to a linear warp and passes through the bounded area of the distance matrix.

Data abstraction [24] is another way to speed up the algorithm. The aim is to reduce the representation of the data, a smaller matrix in this case, to run the algorithm on.

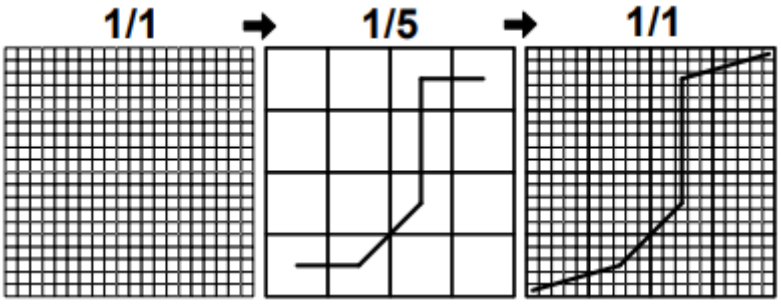

Figure 3.6: Finding warp path on lower-resolution distance matrix

Instead of running the algorithm on a full-resolution matrix, the number of points in the time series are reduced, leading to a lower resolution matrix. The algorithm is then used on this matrix to obtain a warp path, which is then mapped back onto the original full-resolution matrix.

The resulting warp path is usually not optimal, and the level of inaccuracy increases the more the matrix is simplified. This can be significant in applications where local variations have a great effect on the warp path.

This method speeds up the algorithm by a constant factor as well, but the overall time and space complexity remain unchanged.

Algorithm Description

The algorithm is based on graph bisection [25], which involves splitting a graph into smaller, roughly equivalent parts. The goal is to minimise the sum of the edges that would be broken. A multi-level approach aims at correcting large graphs with the solutions from smaller ones, in a dynamic-programming fashion. FastDTW uses this to find an accurate warp path.

The algorithm involves the following operations:

1. *Coarsening*, which involves representing the time series in a reduced form, using as few points as possible.

2. *Projection*, which is to find the warp path on a lower-resolution matrix. This will be used as an initial baseline for further refinement after projecting it on the full-resolution matrix.

3. *Refinement*, which is to refine the projected warp path through local adjustments of the warp path.

Coarsening is used to generate many different resolutions of the matrix, each of them gradually decreasing by a factor of 2 starting from the original time series.

Projection is then used to find the warp path on the lowest resolution matrix. This is then projected on the next lowest-resolution matrix. This is represented as the dark-shaded cells in Figure 3.7.

The dark-shaded cells are used as a window for the next run of DTW, being used as a constraint in the *Refinement* step. This does not guarantee that the optimal global path will be present within that constraint, but the accuracy can be increased by increasing the *radius* of the projected warp path. In Figure 3.7, lightly-shaded cells represent those included in the window, with the *radius* parameter set to 1.

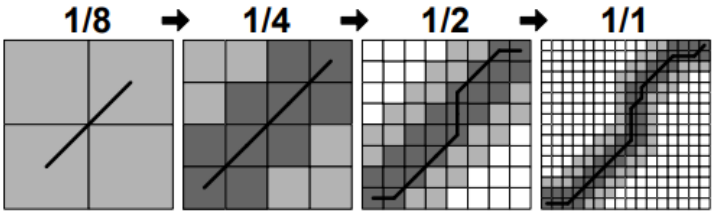

Figure 3.7: Gradual refinement of warp path as resolution increases

FastDTW evaluates only the cells in the neighbourhood of the projected warping path. The number of cells that the algorithm computes scales linearly with the length of the time series. This leads to a complexity of $O(n)$, if we consider n to be a representation of both time series being compared.

The following pseudocode, as outlined in the relevant paper [14], describes the algorithm:

Algorithm 2: FastDTW algorithm

Input : X: array $[x_1, x_2, \ldots, x_m]$, Y: array $[y_1, y_2, \ldots, y_n]$
, radius: int
Output: Distance: float
// The minimum size of the coarsest resolution, the base case
$minSize = radius + 2$
if $|X| \leq minSize$ ***or*** $|Y| \leq minSize$ **then**
    // For small time-series, run DTW
    return *DTW(X, Y)*
else
    $reducedX = X.reduceByHalf$
    $reducedY = Y.reduceByHalf$
    $lowResPath = FastDTW(reducedX, reducedY, radius)$
    $window = HighResWindow(lowResPath, X, Y, radius)$
    return *DTW(X, Y, window)*
end if

Lines 8-9 are executed multiple times until the base case is reached. Line 10 is a recursive call which updates the window one very call, which is then passed to the DTW algorithm to do the adjustments with the new constraints.

Implementation

The `fastdtw_distances` method works similarly to `dtw_distances`. It uses the `fastdtw` method from the `fastdtw` library instead to calculate the distances. This implementation does

not suffer from the same issue of handling sequences of length 1. This implementation runs in linear time, rather than in $O(m \times n)$.

3.6 Sign Prediction

Given the `pandas.DataFrame` with all distances computed, a prediction can now be made. The DataFrame is first sorted in ascending order of distances, with the top row being the sign with the lowest distance. Lower distances indicate higher similarity.

name	sign_model	distance
10 Accept	<models.sign_model.SignModel object at 0x00000...	18927.110506
12 Accept	<models.sign_model.SignModel object at 0x00000...	23533.023133
14 Accept	<models.sign_model.SignModel object at 0x00000...	27783.11955
191 Breakfast	<models.sign_model.SignModel object at 0x00000...	32484.470454
16 Accept	<models.sign_model.SignModel object at 0x00000...	32487.176239
1 Accept	<models.sign_model.SignModel object at 0x00000...	32923.369393
190 Breakfast	<models.sign_model.SignModel object at 0x00000...	33194.436166
13 Accept	<models.sign_model.SignModel object at 0x00000...	35414.323482
18 Accept	<models.sign_model.SignModel object at 0x00000...	35556.541815
93 Barbecue	<models.sign_model.SignModel object at 0x00000...	35573.322004

Figure 3.8: Dataframe with more likely labels, sorted by distance

3.6.1 Batch Size and Threshold

The `get_sign_predicted` method takes an argument for `batch_size`. This indicates how many signs we need to consider in order to make a reliable prediction. For example, we can take a batch size of 5, which means we take the 5 signs with the lowest distances and output the most common one. This increases the confidence in the prediction made. We choose a label l as follows:

$$l = \max\left(\frac{l_i}{N}\right), \text{ for } 0 \leq i < N$$

N corresponds to the chosen `batch_size`. A reasonable upper bound for N can be obtained by considering the label in the training data with the fewest amount of examples. This ensures that every label has the possibility of achieving a confidence of 100%.

The threshold argument is used to determine if a sign is common enough to be output with a certain minimum confidence. It is defined as the minimum ratio of appearances to the batch size that a sign needs to achieve in order to be output. The model outputs "Unknown Sign" if the threshold is not met.

3.7 Sign Recorder

The `SignRecorder` class contains all the methods which bind the program for real-time detection together. It establishes the required pipeline and uses the methods and classes from other scripts to yield a prediction from a recorded sign. An overview of the pipeline is given below:

1. Initialise `SignRecorder` object with an empty list, a DataFrame passed by reference, and a desired sequence length to determine how many frames to record.
2. Start recording using the `record` method. A boolean `is_recording` is used to determine when the object is recording. The distances in the DataFrame are set to 0.
3. While recording, a frame from the input is processed and the results are appended to to the list. This is done while the number of frames processed is less than the sequence length. If this exceeds the sequence length, then recording is stopped and the array is converted to a `sign_model` object.
4. The `sign_model` object is compared with each `sign_model` object in the DataFrame and the resulting distance is stored in the corresponding row.
5. Once all `sign_model` object in the DataFrame have been compared, the DataFrame is sorted in ascending order, starting with the lowest distances at the top.
6. The top `batch_size` rows are chosen and the output label is determined as described in Section 3.6.1.

3.8 Sign Evaluator

The `SignEval` class contains all the methods which enables the model to be tested on testing data. It uses a similar pipeline to that used in the `sign_recorder` class. One difference is that there is no sequence length passed as parameter, as the length of the video sequences are used instead. There is also no variables and methods associated with recording required. The pipeline can be outlined as follows:

1. Initialise `SignEval` object with an empty list, and a DataFrame passed by reference.
2. While the video has frames, the list is filled with the results of each processed frame.
3. When the video has no more frames left, the list is used to initialise a `sign_model` object.
4. The `sign_model` object is compared with each `sign_model` object in the DataFrame and the resulting distance is stored in the corresponding row.
5. Once all `sign_model` object in the DataFrame have been compared, the DataFrame is sorted in ascending order, starting with the lowest distances at the top.
6. The top `batch_size` rows are chosen and the output label is determined as described in Section 3.6.1.

The `sign_eval` object can be used to generate a list of predicted labels, which can be compared with the list of actual labels to obtain a measure of accuracy.

3.9 Webcam Manager

The `WebcamManager` class contains the methods to control the visual interface of the OpenCV feed. It provides information such as what is the output label and when the model is recording. This class is only relevant for real-time detection.

3.10 Usage

The implementation provides a record button to start a capture of a specific number of frames. Once pressed, the feed will capture that amount of frames and process them. To indicate that the feed is recording, a circle is used as an indicator. It turns red while the feed is recording.

A hand and pose model will be generated for each frame and they are all stored in a sign model. Once compared, the program will output the most probable label as a prediction, which will be displayed on the bottom of the feed.

3.10.1 Constants

`constants.py` is a script containing parameters and constants. `FASTDTW` is a boolean which toggles between using the basic DTW algorithm and the FastDTW implementation. `BATCH_SIZE` is an integer representing the batch size mentionned in Section 3.6.1. This can be tweaked according to the data being trained on. `THRESHOLD` is an integer representing the threshold mentionned in Section 3.6.1. Changing this value will affect the confidence of the model. `DATA_PATH`, `TEST_PATH`, and `FEATURE_PATH` are constants which dictate where training videos, test videos, and keypoint data are stored. `mappings` is a dictionary which is used to process the LSA64 dataset, associating its corresponding labels by Id.

Chapter 4

Evaluation

This chapter describes the results obtained, as well as the evaluation methods and metrics used. Possible optimisations and relevant factors affecting accuracy are also discussed. The testing for all datasets involves few labels, due to the hardware limitations (not enough memory to load all reference signs at the same time). The key results are that the model yields high accuracy on both datasets, similar to other models that have been evaluated on the same datasets. It is important to note that the model does not perform well when presented with signs that it has not been trained on, and when the training data has an ambiguous number of hands present in them.

4.1 LSA64 Results

4.1.1 5-label Training

The model was tested on the LSA64 dataset, using both the basic `dtw-python` implementation and the `fastdtw` one. The model was trained on a sample of five different labels for sign gestures: `Accept`, `Appear`, `Argentina`, `Away`, and `Barbecue`. `Accept`, `Appear`, and `Barbecue` require both hands whereas the `Argentina` and `Away` can be performed with a single hand. A set of 50 videos, 10 for each label was used for testing. The accuracy for different amount of training data are as follows:

Num. of videos	Accuracy (dtw-python)	Accuracy (fastdtw)
1	0.78	0.78
5	0.80	0.72
10	0.94	0.74

Table 4.1: Results using dtw-python

As Table 4.1 shows, the accuracy of the model increases as the number of training videos per label increases. `dtw-python` obtains better results than `fastdtw` consistently, with only the case of using a single video per label yielding a similar accuracy. This suggests that `fastdtw` is as good as `dtw-python` for cases where there is a low amount of training data. Further evidence is needed to make a conclusion. This will be examined in further sections.

The effect of using a different amount of training data is discussed further, including effects due to the number of hands, and how each implementation responds to different labels.

Training on 1 video per label

The overall accuracy was 78% for both `dtw-python` and `fastdtw`. The distribution of correctly predicted labels is shown below:

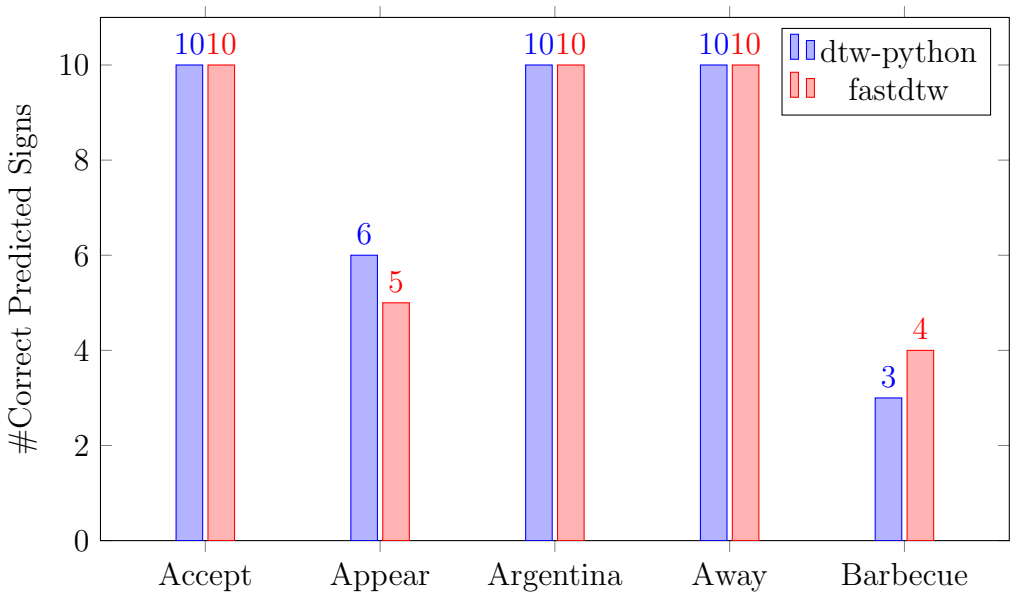

Figure 4.1: # Correctly predicted signs per label, using one video per label for training

`Appear` and `Barbecue` seem to perform poorly on both implementations. They both involve two hands, but Accept does as well and performs very well nevertheless. Other factors that might affect the accuracy could include:

- `Appear` is executed relatively fast, and part of the hand is hidden during the execution of the sign. This could be affecting the mediapipe detection.
- `Barbecue` uses both hands in a horizontal position. This seems to make it hard for mediapipe to locate the landmarks for the hands.
- The model might be biased in favour of `Accept` due to an inherent feature of the sign gesture. This would make it seem more accurate, even if the model was predicting it for other signs.

Training on 5 videos per label

The `dtw-python` implementation had an accuracy of 80%, whereas the `fastdtw` implementation had an accuracy of 72%. The correct label distribution is shown below:

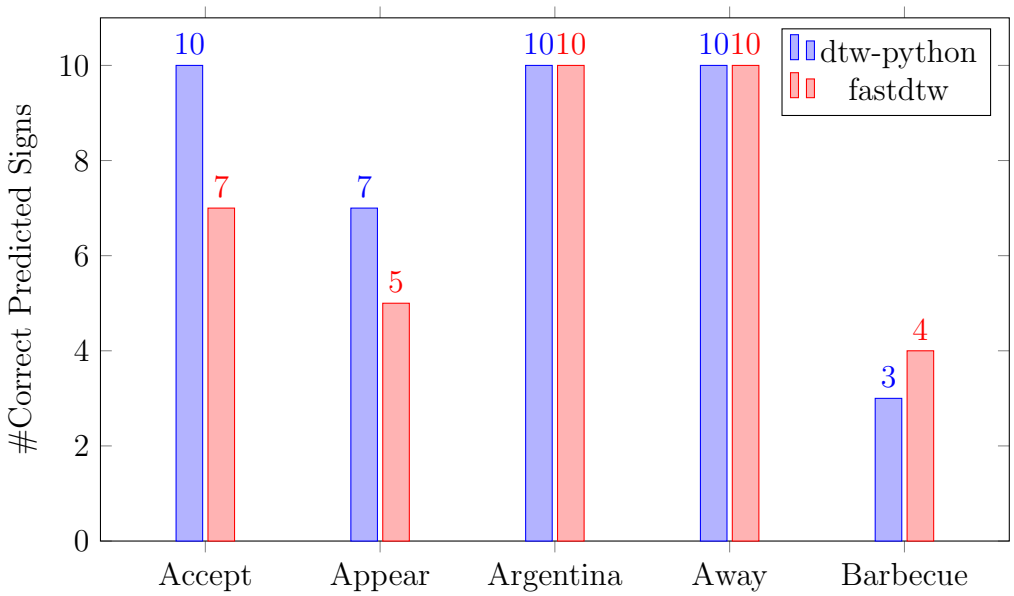

Figure 4.2: # Correctly predicted signs per label, using 5 videos per label for training

We note the same inconsistencies that are present in Figure 4.1. `fastdtw` also performs slightly less efficiently overall compared to `dtw-python`. It is important to note however that `fastdtw` gets all the signs predicted correctly for signs involving a single hand (`Argentina` and `Away`). This indicates possibility for optimisation in further implementations.

Training on 10 videos per label

The `dtw-python` implementation had an accuracy of 94%, whereas the `fastdtw` implementation had an accuracy of 74%. The correct label distribution is shown below:

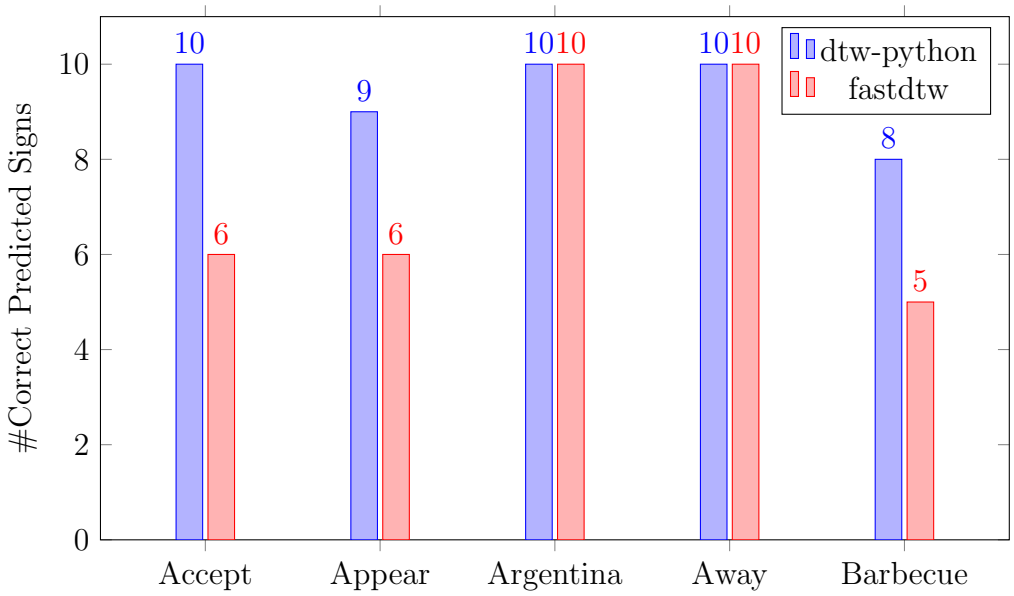

Figure 4.3: # Correctly predicted signs per label, using 10 videos per label for training

Accuracy is higher overall for this training set. We note the same lower accuracy for `fastdtw` when it comes to two-handed signs and the same bias against `Appear` and `Barbecue`.

Overall

The results are similar for all conditions, with `dtw-python` being affected more by the change in training data as opposed to `fastdtw`, which stayed pretty consistent. The fact that certain signs performed consistently well and some did not get recognised as much indicates that features inherent to the signs are more likely to be the problem rather than the implementation. It is also possible that the pipeline is less suitable for such signs.

4.1.2 Left-handed Videos Testing

The model supports left-handed detection as well, even when trained on right-handed videos. Testing was conducted in a similar fashion on the same testing data used for right-handed testing, but with all of the videos flipped instead. The training data consisted of the same videos used in Section 4.1.1. Both the `dtw-python` and `fastdtw` implementations had an overall accuracy of 78%. The distribution for correctly predicted labels is shown below:

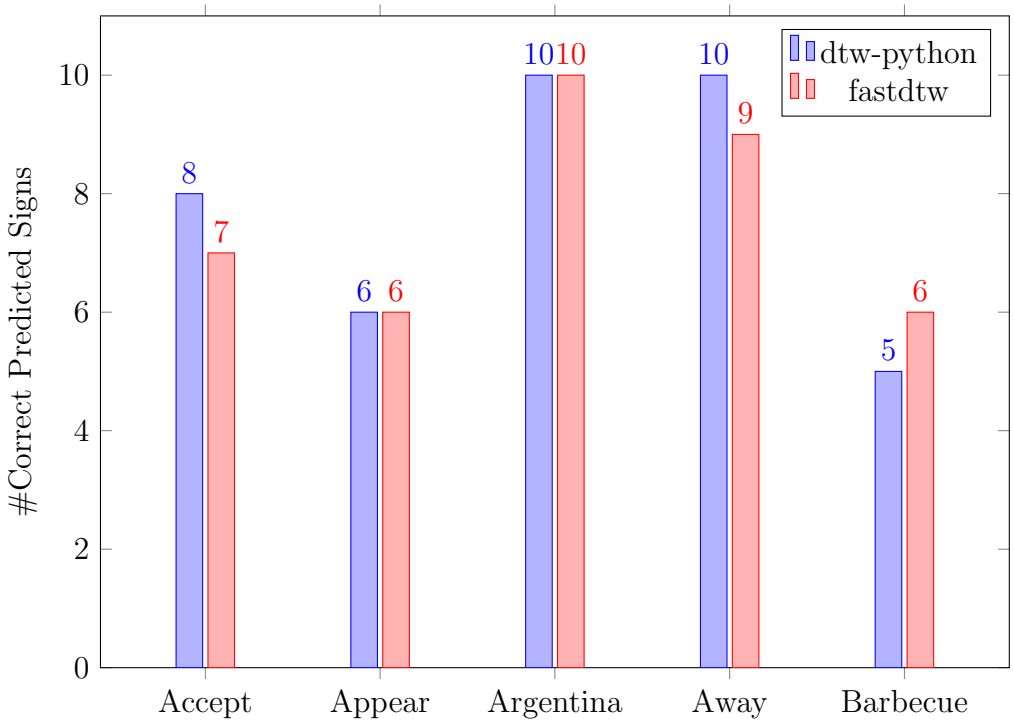

Figure 4.4: # Correctly predicted left-handed signs per label

We note similar results to that obtained by the model as seen in the previous sections. The overall accuracy seems to be slightly lower, despite using flipped frames for training. The differences in the results can be explained by the fact that the layout of the coordinates extracted from mediapipe for a flipped video is not identical to that of coordinates extracted from a non-flipped video.

4.2 BSLDict Results

The BSLDict dataset proved to be harder to evaluate due to its lack of suitable data. The dataset was filtered to extract labels which have a sufficient amount of data, which was $n = 12$ in

that case. The dataset was evaluated using both the `dtw-python` and `fastdtw` implementations. The labels used were `Conduct`, `One`, `Only`, `Pick`, and `Stop`. `One` requires a single hand to perform and the others require both hands. As many of the labels contained ambiguous data (`One` contained one video for the sign representing `Fourteen`, which can be inferred from the mouthing cues and comparing with existing signs), the labels were filtered again to leave the most common representations.

The testing was conducted on real-time data for this dataset. This is due to the lack of suitable data, given the size of the filtered dataset. Instead, I replicated each demonstrated sign as shown in the training videos in front of the webcam and noted the results. 20 repetitions were executed per sign as this was a reasonable amount to conduct in real-time while providing an accurate estimate of the efficiency of the model. 10 of the signs were performed right-handed and the other 10 were done left-handed. Additionally, 20 additional signs from the previous LSA64 dataset were performed in front of the model trained on the BSLDict signs to evaluate the confusion matrix. Given that the model is not optimised to detect negatives, it is expected that the model yields poor results when it comes to detecting them. However, it is of interest to know about its performance so that this aspect can be improved in future work. The model then yielded the following results on the resulting data:

Label	Num. of training videos	Accuracy (dtw-python)	Accuracy (fastdtw)
`Conduct`	4	0.80	0.70
`One`	4	1.00	0.95
`Only`	4	0.75	0.75
`Pick`	4	0.10	0.05
`Stop`	4	1.00	0.95

Table 4.2: Results for each label

The model achieved an overall accuracy of 73% using `dtw-python` and 68% using `fastdtw`. The individual results are illustrated in Table 4.2. Additionally, the confusion matrix illustrates the number of positives and negatives for both implementations.

$$\texttt{dtw-python:} \begin{bmatrix} 96 & 0 \\ 22 & 2 \end{bmatrix} \quad \texttt{fastdtw:} \begin{bmatrix} 98 & 0 \\ 21 & 1 \end{bmatrix}$$

The results demonstrate high accuracy overall, but we also note that the sign `Pick` performs poorly overall. Analysis of the training data used for it and the results leads to the following deductions:

- The training data for `Pick` implies that the sign for it uses only one hand, but the data itself has both hands visible, one of them resting in a neutral position. This demonstrates the need for further data pre-processing, such as cropping the frames.
- Many of the predictions for `Pick` yielded `One` instead. Both signs are relatively similar, especially compared to the other signs in the training data. This indicates that the model developed a bias for some labels, which is likely given the low amount of training data.

We also note that `fastdtw` is slightly less accurate than `dtw-python`, but still obtaining comparable results.

The confusion matrices shows that the model struggles to recognise negatives, mostly classifying them as signs in the training data. The lack of training data prevents the `THRESHOLD` argument to be changed much in order to get more confident results.

4.3 Other Studies

The results can be compared to similar studies that have been done on the same datasets. The best performing models and their accuracies are illustrated in Table 4.3 shown below.

Model	Dataset	Accuracy
`dtw-python`	LSA64	0.94
`fastdtw`	LSA64	0.78
SPOTER [4]	LSA64	1.00
Inception V3 [26]	LSA64	0.96

Table 4.3: Results for each label

We can see that the `dtw-python` implementation performs nearly as well as SPOTER and Inception V3. `fastdtw` is seen to yield less accurate results. One advantage of the model proposed is that it can yield accurate results using a fraction of what the other models are trained on.

4.4 User Study

As this project involves real-time detection, a user study was conducted to measure the usability of the model in real-time and to also evaluate its accuracy on a wider range of users in real-life circumstances. The aim of the user study was to establish if the model was suitable to be used for real-time detection and if the `fastdtw` implementation was faster than the `dtw-python` implementation.

4.4.1 Methodology

A group of 5 participants took turns performing sign language gestures that the model is trained on. They are provided with a few examples from the testing data to familiarise themselves with the gestures. The user study is carried out separately on both the LSA64 dataset and the BSLDict dataset.

4.4.2 LSA64 Testing

The model was trained on the same videos as the ones used for the *10 videos per label* section in Subsection 4.1.1. Each user performed 10 repetitions of each sign, 5 for each implementation. This leads to a total of 50 repetitions. The results obtained are illustrated below:

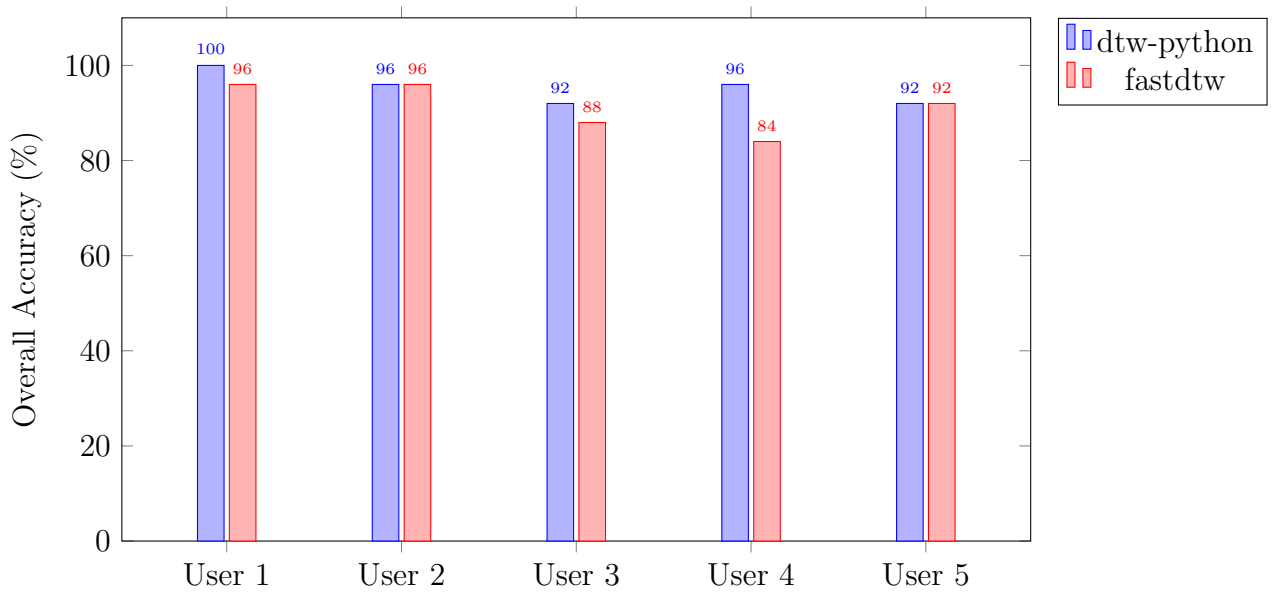

Figure 4.5: # Correctly predicted signs per label

All users obtained very high accuracy overall for both implementations. The `dtw-python` implementation had an overall accuracy of 95.2% and the `fastdtw` implementation had an overall accuracy of 91.2%, for all users. The distribution of correctly predicted labels was uniform, with no sign performing noticeably better except `Accept`, which was correctly predicted for all users.

4.4.3 BSLDict Testing

The model was trained on the same videos as in Section 4.2. The users performed 10 repetitions for each sign, 5 for each implementation. The results obtained for each user are illustrated below:

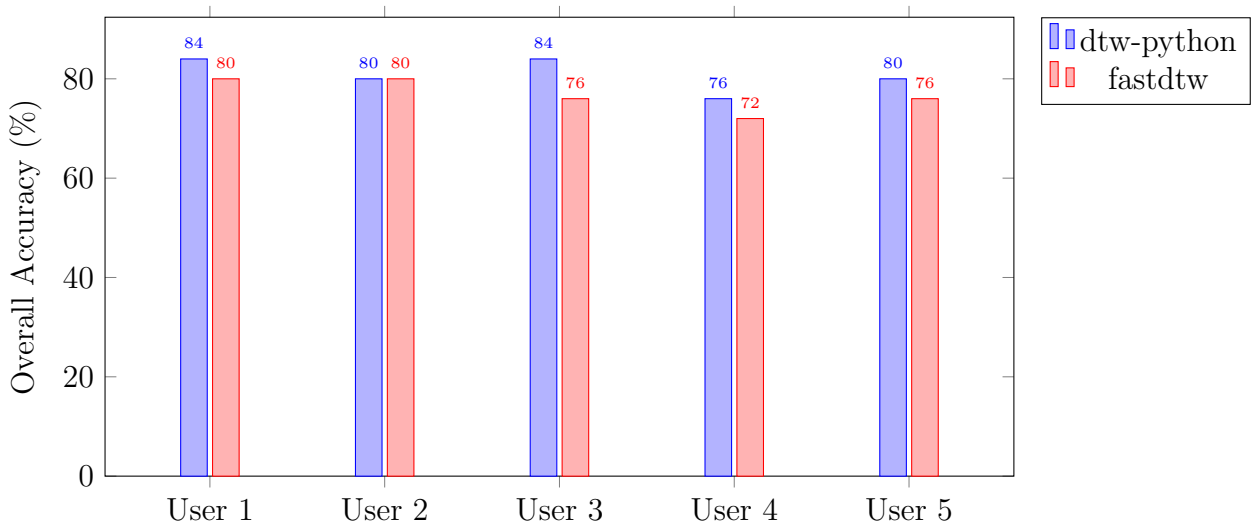

Figure 4.6: # Correctly predicted signs per label

Users obtained less accurate results, with the `dtw-python` implementation having an overall

accuracy of 80.8%, and 80.0% for the `fastdtw`. The principal reason for the low accuracy was the `Pick` sign, which had an overall accuracy of 16%, correlating with results obtained in Section 4.2. The reasons for this are likely the same, and we can also exclude the possibility of the bias to be due to a specific user as well.

4.4.4 User Feedback

Users gave qualitative feedback on the evaluation process used for the user study. The overall opinion leaned towards needing to hold the signs for some period on time for signs in the LSA64 dataset, and the inaccuracy of `Pick` for the BSLDict dataset. Users also did not find the need to hide the non-signing hand for single-handed signs intuitive, and would rather have both hands present on the webcam feed. A breakdown of ratings given by the users is illustrated below:

Model	Usability (/10)	Satisfaction based on Accuracy(/10)
User 1	8	9
User 2	7	8
User 3	9	8
User 4	6	6
User 5	7	8

Table 4.4: Results for each label

4.5 Performance

It is desired to test if the `fastdtw` implementation provides faster computation, given that it yields less accurate predictions as seen in the previous sections. The average time used to process the extracted results from videos is used as a comparison metric. The model used the time from the evaluation of the LSA64 dataset, under the same conditions as in Section 4.1.1. The results are illustrated below:

Implementation	Processing Time (s)
`dtw-python`	0.985 ± 0.363
`fastdtw`	1.694 ± 0.913

Table 4.5: Average processing time for each implementation

The results demonstrate that `fastdtw` actually performs slower than the base `dtw-python` implementation. Given that this contradicts the previous hypothesis that `fastdtw` provided an optimisation of the algorithm, further testing is done in an isolated environment. The code used to perform such testing is shown in the appendix.

Both implementations were run on randomly-generated lists with varying sizes, with the time series being compared having the same length. All computations were run a 100 times to obtain an accurate mean and standard deviation. The results are illustrated below:

Implementation	Array Size	Processing Time (s)
`dtw-python`	100	0.00084 ± 0.00035
`dtw-python`	1000	0.03945 ± 0.00439
`dtw-python`	10000	3.33221 ± 0.06959
`fastdtw`	100	0.00415 ± 0.00037
`fastdtw`	1000	0.05859 ± 0.00843
`fastdtw`	10000	0.58846 ± 0.01833

Table 4.6: Average processing time for each implementation

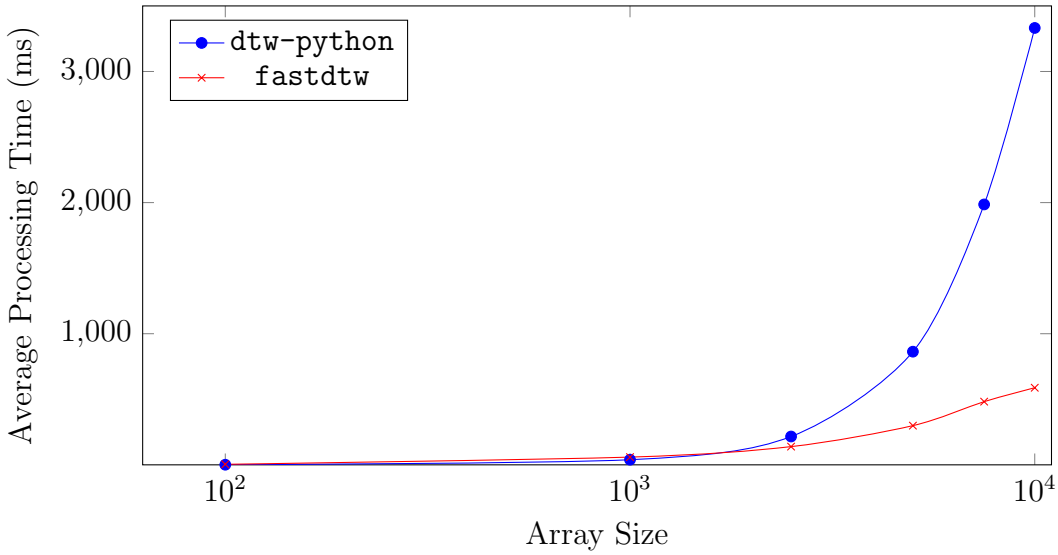

Figure 4.7: Variation of processing time with array size

The results from the isolated test shows that the processing time is dependent on the size of the array. `dtw-python` grows quadratically, compared to `fastdtw` which grows linearly. The difference seems to start showing when the array size is at least 2000. As such, this suggests that there is not enough data being compared to leverage the benefits of `fastdtw`. A few potential ways to make full use of its performance would include:

- Capturing more frames during the determined recording length. This would lead to longer lists being compared.
- Adapting the comparison to longer sequences, such as sign gestures that take longer to execute.

This is important to consider if further work is done to research the benefits of this method on continuous sign language recognition.

Chapter 5

Conclusion

5.1 Accomplishments

The original aim of this project was to provide an image learning model that successfully recognises a subset of British Sign Language with reliable accuracy and efficiency. Success of the project was also determined to be based on the implementation of the training pipeline and the possibility of using other external devices other than webcams.

Over the course of the project, the success criterion of implementing an image learning model was augmented by instead training on videos. This corresponds to one of the extensions covered in the project proposal, relating to the evaluation of signs with more movement rather than static ones. The model also does per-frame processing on the training data, which fulfills this success criterion. The resulting model achieved similar accuracy compared to other sign language recognition models.

The model was also trained on Argentinian Sign Language. This demonstrates the ability to easily incorporate new training data to the model to augment its vocabulary. As such, this not only meets the relevant success criteria listed in the proposal, but also cover the extension about incorporating other variations of sign languages apart from British Sign Language into the model pipeline.

Research about other external devices has been done. The separation between the data capture and the training process means that the model can be successfully tuned to a different device to capture data, rather than coordinates from Mediapipe with video obtained from a webcam. Thus, this success criterion has also been met. Actual testing using such devices was not conducted due to their unavailability.

Finally, the project incorporated a faster algorithm for processing, namely `fastdtw`. Despite being tested and shown to achieve slower processing times compared to the basic implementation provided by `dtw-python`, the algorithm scales much more efficiently as the sequences grow in size, indicating possibility for further extensions.

5.2 Further Work

While the model met all the required success criteria, the work done during the project timeline revealed that there is a lot of room for improvement. One point that has been brought up was the model's inability to recognise negatives reliably, which are gestures that it has not been trained on. Being able to do so would enable even more possible further work, such as continuous

sign language recognition.

Continuous sign language recognition involves much more precise processing, with heavy emphasis on data segmentation to extract features. It would be interesting to see how well the DTW algorithm would fare on such data and if it would still be a viable model to be used for such purposes.

Bibliography

[1] S. Albanie, G. Varol, L. Momeni, H. Bull, T. Afouras, H. Chowdhury, N. Fox, B. Woll, R. Cooper, A. McParland, *et al.*, "Bbc-oxford british sign language dataset," *arXiv preprint arXiv:2111.03635*, 2021.

[2] H. R. V. Joze and O. Koller, "Ms-asl: A large-scale data set and benchmark for understanding american sign language," *arXiv preprint arXiv:1812.01053*, 2018.

[3] F. Ronchetti, F. Quiroga, C. A. Estrebou, L. C. Lanzarini, and A. Rosete, "Lsa64: an argentinian sign language dataset," in *XXII Congreso Argentino de Ciencias de la Computación (CACIC 2016).*, 2016.

[4] M. Boháček and M. Hrúz, "Sign pose-based transformer for word-level sign language recognition," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 182–191, 2022.

[5] L. Momeni, G. Varol, S. Albanie, T. Afouras, and A. Zisserman, "Watch, read and lookup: learning to spot signs from multiple supervisors," in *Proceedings of the Asian Conference on Computer Vision*, 2020.

[6] T. Liu, W. Zhou, and H. Li, "Sign language recognition with long short-term memory," in *2016 IEEE international conference on image processing (ICIP)*, pp. 2871–2875, IEEE, 2016.

[7] D. Guo, W. Zhou, H. Li, and M. Wang, "Hierarchical lstm for sign language translation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[8] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE transactions on acoustics, speech, and signal processing*, vol. 26, no. 1, pp. 43–49, 1978.

[9] P. C. Huu, Q. Le, and T. Le, "Human action recognition using dynamic time warping and voting algorithm," *VNU J Sci Comp Sci Com Eng*, vol. 30, no. 3, pp. 22–30, 2014.

[10] C. A. Ratanamahatana and E. Keogh, "Everything you know about dynamic time warping is wrong," in *Third workshop on mining temporal and sequential data*, vol. 32, Citeseer, 2004.

[11] C.-H. Chuan, E. Regina, and C. Guardino, "American sign language recognition using leap motion sensor," in *2014 13th International Conference on Machine Learning and Applications*, pp. 541–544, IEEE, 2014.

[12] S. Lang, M. Block, and R. Rojas, "Sign language recognition using kinect," in *Artificial Intelligence and Soft Computing: 11th International Conference, ICAISC 2012, Zakopane, Poland, April 29-May 3, 2012, Proceedings, Part I 11*, pp. 394–402, Springer, 2012.

[13] H. Wang, M. C. Leu, and C. Oz, "American sign language recognition using multi-dimensional hidden markov models.," *Journal of Information Science and Engineering*, vol. 22, no. 5, pp. 1109–1123, 2006.

[14] S. Salvador and P. Chan, "Toward accurate dynamic time warping in linear time and space," *Intelligent Data Analysis*, vol. 11, no. 5, pp. 561–580, 2007.

[15] I. Grishchenko and V. Bazarevsky, "Mediapipe holistic — simultaneous face, hand and pose prediction, on device," Dec 2020.

[16] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, pp. 21–37, Springer, 2016.

[17] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, pp. 303–338, 2010.

[18] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pp. 740–755, Springer, 2014.

[19] V. Bazarevsky, I. Grishchenko, K. Raveendran, T. Zhu, F. Zhang, and M. Grundmann, "Blazepose: On-device real-time body pose tracking," *arXiv preprint arXiv:2006.10204*, 2020.

[20] V. Bazarevsky, Y. Kartynnik, A. Vakunov, K. Raveendran, and M. Grundmann, "Blazeface: Sub-millisecond neural face detection on mobile gpus," *arXiv preprint arXiv:1907.05047*, 2019.

[21] A. Newell, K. Yang, and J. Deng, "Stacked hourglass networks for human pose estimation," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VIII 14*, pp. 483–499, Springer, 2016.

[22] T. Giorgino, "Computing and visualizing dynamic time warping alignments in r: the dtw package," *Journal of statistical Software*, vol. 31, pp. 1–24, 2009.

[23] F. Itakura, "Minimum prediction residual principle applied to speech recognition," *IEEE Transactions on acoustics, speech, and signal processing*, vol. 23, no. 1, pp. 67–72, 1975.

[24] S. Chu, E. Keogh, D. Hart, and M. Pazzani, "Iterative deepening dynamic time warping for time series," in *Proceedings of the 2002 SIAM International Conference on Data Mining*, pp. 195–212, SIAM, 2002.

[25] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Application in vlsi domain," in *Proceedings of the 34th annual Design Automation Conference*, pp. 526–529, 1997.

[26] R. S. Rajput, A. Purushothaman, R. Jeena, C. M. Manu, *et al.*, "Sign language gesture recognition from spatio-temporal features," 2022.

Appendix A

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Sign Language Recognition

Dissertation Author
Originator: The Author
14 October 2022

Project Supervisor: Mr. Filip Svoboda
Director of Studies: [Redacted for Blind Grading]
Project Overseers: Dr. Ferenc Huszar & Dr. Andreas Vlachos
University Teaching Officer: Professor Nicholas Lane

Introduction

Sign Language has been adopted by many people who are unable to speak, or hear. While technology has been used to help this affected minority group to a certain degree, there are still a lot of facilities that can be improved to help them communicate. For example, even Windows does not support a form of sign language recognition. This is despite the advances that we have concerning digital assistants, which could help even further. Furthermore, there are different variations of sign languages, and most of the research focus has always exclusively been done on American Sign Language (ASL), leading to less inclusion. Thus, it is of interest to develop reliable software which could help in including variations of sign language as inputs to modern day devices.

Work to be done

The goal of this project is to implement a reliable detection model that can recognise a certain subset of sign language expressions displayed in front of a camera. The particular variation of sign expressions that will be focused on is British Sign Language. Given that there is a

possibility of a lack of reliable raining data, there should also be a way of adding new labelled signs to the training data.

I will first start by performing image labelling on data gathered. Collection will be done either on an existing dataset if a suitable one is available, or manually created if none are. This will be followed by training the model on the collected dataset, and then perform testing. The model will be evaluated with respect to other similar implementations.

Python seems like a suitable language for this project due to its popularity and variety of machine learning libraries available for it. It is a high-level language which has simple and readable code, making the implementation part of the project less tricky. OpenCV is a popular computer vision library that could be used for the project and which is popular for these applications. The method of recognising hand gestures is not confirmed and will be decided after the research and preparation phase. A good starting point for that would be contour detection.

Success Criteria

The project will be considered a success if I have done all of the following:

- Implemented an image learning model that successfully recognises a subset of British sign language with reliable accuracy and efficiency.
- Implemented a reliable way of capturing training data for extending the vocabulary of the model.
- Evaluated the efficiency and accuracy of the model, comparing it to the results of similar sign language recognition models.
- Explore possibility of using external devices other than webcams, such as devices equipped with depth-sensors (the Kinect[1]for example).

Extensions

- Expand the training data to cover other variations of sign languages.
- Optimise the algorithms to achieve faster processing.
- Evaluate the model with languages that involve more movement, rather than more static expressions.

Starting Point

I have some experience using python at an intermediate level.

I have never used OpenCV or any other machine learning libraries. Most of my experience with machine learning comes from coursework

[1]See https://en.wikipedia.org/wiki/Kinect

Resources required

A list of required resources:

- My own machines:
  - Laptop (Intel i7-10750H, 16GB RAM, Windows 11)
  - Backup Laptop (Intel i5-7200U, 8GB RAM, Ubuntu 21.04)
- OpenCV and other machine learning libraries, which will be settled on upon further research and noted in the reports.
- Github will be used to store regularly backups of the code along with a copy of the dissertation and proposal. Additionally, they will be kept on OneDrive and Google Drive.

Timetable and milestones

Work Package 1 : 13 Oct - 27 Oct
Task: Research about machine learning models and of similar implementations.
Milestone: Have run basic detection scripts on machine which can do basic contour detection or hand tracking if possible.

Work Package 2 : 27 Oct - 10 Nov
Task: Perform data collection and start implementing the sign language expression data capture component.
Milestone: Successfully obtain a reasonable amount of data for training and testing, along with a component that can successfully store training data from displayed sign expressions.

Work Package 3 : 10 Nov - 24 Nov
Task: Start implementing the base model, along with conducting some more research if needed to fill in the gaps.
Milestone: Model successfully able to differentiate hand and background, along with skeleton code for recognising signs set up.

Work Package 4 : 24 Nov - 08 Dec
Task: Finish implementation to recognise different sign expressions when displayed.
Milestone: Successfully able to differentiate different sign expressions shown on camera.

Work Package 5 : 08 Dec - 22 Dec
Task: Slack time / Start drafting implementation chapter and catch up on late deliverables. Start to get structure of dissertation written down.
Milestone: Finalised implementation of project.

Work Package 6 : 22 Dec - 05 Jan
Task: No Work on Project
Milestone: Finalised implementation of project and outline of dissertation setup.

Work Package 8 : 05 Jan - 19 Jan

Task: Write implementation of draft dissertation
Milestone: Draft Implementation chapter prepared and sent to supervisor for feedback.

Work Package 9 : 19 Jan - 02 Feb
Task: Write Progress Report and prepare report presentation
Milestone: Submit Progress Report (by 12:00 03 February)
Milestone: Presentation Ready

Work Package 10 : 02 Feb - 16 Feb
Task: Slack Time / Work on late deliverables
Milestone: Project has successfully been implemented and implementation feedback has been incorporated into draft dissertation.

Work Package 11 : 16 Feb - 02 Mar
Task: Conduct evaluation of implementation. Setup possible optimisations if needed.
Milestone: Demonstrate a convincing argument of the efficiency and possible optimisations of the implementation to supervisor.

Work Package 12 : 02 Mar - 16 Mar
Task: Write Evaluation chapter for draft dissertation
Milestone: Draft Evaluation chapter prepared and sent to supervisor for feedback.

Work Package 13 : 16 Mar - 30 Mar
Task: Finish draft dissertation and add missing chapters.
Milestone: Draft dissertation completed and sent to supervisor

Work Package 14 : 30 Mar - 13 Apr
Task: Incorporate draft dissertation feedback into final dissertation and resent to supervisor for a second round of feedback.
Milestone: Dissertation completed with feedback incorporated.

Work Package 15 : 13 Apr - 27 Apr
Task: Slack Time / Finalise Project. Catch up on any late deliverables.
Milestone: Final Project and Dissertation Ready for submission

Work Package 16 : 27 Apr - 11 May
Task: Slack Time
Milestone: Dissertation submitted (by 12:00 12 May)