FAO ***Dr. Amina Souag, Dr. Hannan Azhar***

Canterbury Christ Church University

School of Engineering, Technology and Design

B.Sc. (Hons.) Computer Science

2023-2024

Individual Project (IP40)

Title: *Utilizing Transformers for American Sign Language Fingerspelling Recognition*

Author: *Jamie Pinnington*

Supervisors: *Dr. Amina Souag, Dr. Hannan Azhar*

Email: *JP878@canterbury.ac.uk*

This report is submitted in partial fulfilment of the requirement for the B.Sc. (Hons.) *Computer Science* at Canterbury Christ Church University

Signed *Jamie Pinnington*

Date of Submission: *13th May 2024*

1. Acknowledgements

We'd like to thank the Kaggle community for their invaluable insight and contributions for data augmentations.

Contents

List of Figures

List of Tables

2. Introduction:

THIS Individual Project documents the conceptualization and development of a machine learning model for translating American Sign Language (ASL) fingerspelling videos into text. The foremost aim of the project, part A, is to create a model that can interpret ASL fingerspelling video and predict the corresponding text to a realistic degree. Given the success of part A, a smaller secondary objective part B is to develop a user-friendly application that allows the user to input ASL fingerspelling videos and receive the corresponding text output. The project outcomes are two-fold, a machine learning model and a user-friendly proof-of-concept that serves academia.

The field of AI has seen rapid development in image and video recognition, however, the most prominently researched topics related to fingerspelling is automatic-speech-recognition (ASR). Interpreting sign language, gestures, and fingerspelling are even smaller niches, and development in these areas is lower. Leading figures such as Bowen Shi (Shi et al., 2021) provided the first attempts to recognize fingerspelling in ”wild” environments.

Research for ASL fingerspelling recognition is lacklustre because of its niche topic, and small amount of overall ASL users worldwide, perhaps as high as 2 million (Mitchell et al., 2006; Ethnologue, 2023).

Furthermore, the complexity of ASL fingerspelling recognition which has a high degree of variability in hand shape, movement, and speed of signing, and much more characteristics makes recognition a challenging task. This requires a large dataset and a robust model to solve this task, and over the years created datasets are small, lack diversity, and are not developed realistically to a real-world environments.

The purpose of this project is to make use of emerging technologies and datasets in order to produce a model that can further the academic field, and improve accessibility for Deaf and Hard of Hearing individuals. The idea is that for a lot of users fingerspelling is faster than typing on a keyboard or smartphone, and so could provide a means of alternative smoother communication between Deaf and Hard of Hearing individuals, and technology. The model produced in this project will be a proof-of-concept, and will not be a final product, but will serve as a stepping stone for future research and development in the field.

The Cross-Industry Standard Process for Data Mining (Hotz, 2018), is a widely used methodology for data mining and artificial intelligence projects. It involves 6 steps that in practice are iterative, and can be revisited at any time as needed. The steps are Business Understanding, Data Understanding, Data Preparation, Modelling, Evaluation, and Deployment.

Over the course of this document, we will document a literature review to discover our roots, following with the CRISP-DM methodology outlining our strategic and technical decisions. In here, expect to find a detailed explanation of the project’s development, the model’s architecture, and the application’s design. We will also discuss the project’s management, risks involves, milestones achieved, and state the achieved outcome of the project.

3. Chapter 1: Background and Related Work

The literature review for this project is maintained as a separate piece of work, and so may have repetition to other chapters of this project.

3.1. Introduction

3.1.1. Background

Sign language is the primary form of communication for the deaf and hard of hearing community. It allows communication when the spoken language is not possible, and or when the speaker or receiver is deaf or hard of hearing. Depending on the situation, and like any language, it requires both parties to be fluent in the language to communicate effectively. However, this is not always the case. American Sign Language(ASL) is a complete, complex language that employs signs made with the hands and other movements, including facial expressions and postures of the body, and is used natively in the United States of America and globally by many individuals.

Whilst no attempt has officially been made to survey the language, and most current estimates are based off of historical surveys that prove to be inaccurate Mitchell et al. (2006). It is estimated that there are over 1 million signers Ethnologue (2023), but others estimates are as high as 2 million Mitchell et al. (2006). ASL communicates through a variety of means including gestures, non-manual markers and lexical signs. The most understood are lexical vocabulary, each corresponding to a word or morpheme. Gestures and non-manual markers such as facial expression can complement and convey more interactive or meaningful lexical signs. Additional constructs include usage of space, role shifting and classifiers.

3.1.2. Purpose

The primary aim of this project is to advance the field of automatic ASL fingerspelling recognition by evaluating the performance of various machine learning models under different conditions.

3.1.3. Scope

If a model has been used to interpret ASL fingerspelling then we will attempt to understand why it's being used.

Assessing the effectiveness of different models in recognizing ASL fingerspelling through a series of controlled experiments that simulate various real-world conditions (e.g., differing light conditions, backgrounds, and hand positions).

Identifying the technical constraints associated with each model, including computational requirements and scalability, to understand their feasibility for widespread implementation.

3.1.4. Research Questions

- RQ-1: Comparative Analysis of Machine Learning Models: What are the strengths and weaknesses of different machine learning (ML) models, such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Transformer models, in the context of ASL fingerspelling recognition?

- RQ-2: Performance Evaluation: How do various machine learning models perform in terms of accuracy, processing speed, and reliability for ASL fingerspelling recognition under different conditions (e.g., varying lighting, hand positions, backgrounds)?
- RQ-3: Dataset and Model Suitability: How does the choice of dataset, including its size, diversity, and quality, influence the effectiveness of different machine learning models in recognizing ASL fingerspelling?
- RQ-4: Real-World Applications: Considering practical applications like kiosk systems, which machine learning models offer the best balance between technical performance and user experience for ASL fingerspelling recognition?
- RQ-5: Technical Challenges: What technical challenges are commonly faced across different machine learning models in ASL fingerspelling recognition, and how adaptable are these models to address such challenges?
- RQ-6: Impact of Environment Variables: To what extent do environmental variables (like hand orientation, motion speed, and background noise) affect the performance of different machine learning models in ASL fingerspelling recognition?
- RQ-7: State of the Art and future directions: What are the most recent and influential works in the field of ASL fingerspelling recognition, and what are the emerging trends and future directions?

3.2. Methodology

3.2.1. Literature Identification

This literature search was completed using the databases IEEE Xplore, Google Scholar, ACM Digital Library, and ScienceDirect. Search terms such as ”ASL fingerspelling recognition”, ”Deep learning for ASL recognition”, ”ASL recognition with CNN”, ”Accuracy of ASL recognition models”, ”Latest trends in ASL recognition”, and ”ASL recognition in real-time” were used alone and in conjunction with boolean operators ”AND”, ”OR” to refine the search results. The search was limited to papers published in the last 5 years, and only peer-reviewed journal articles, conference papers, and high quality theses were considered. The search was also limited to papers written in English. The search was conducted in between November and December 2023, and the results were filtered to include only papers that were published between 2018 and 2023. In addition to this search which was completed in order to find relevant literature for the model architecture comparison, the references of the papers that were selected were used to find additional relevant literature, that dates further back in order to substantiate our historical context and technical background

3.2.2. Literature Evaluation

Of the literature that fit out search criteria, the most relevant papers were selected based on the following criteria: the relevance to the research questions, papers that specifically address ASL fingerspelling, ML models in sign language interpretation, papers that used widely recognized datasets relevant to ASL recognition, editorials, opinion pieces, and non-peer reviewed articles were excluded. Papers that were preferred had a clear methodology, defined objectives and analysis of data. Papers with high citation counts were also given preference. Papers that were selected were read in full, and the results were summarized in a table, which is included in the results section.

3.3. Historical Context (RQ-7)

- Provide a brief overview of the evolution of ASL fingerspelling recognition.
- Highlight key milestones and breakthroughs in the field's history.
- Connect historical developments to current trends and future directions.

Early approaches to sign language recognition, according to Saeed et al. (2022) used robotic like data/power gloves which were wired, with sensors to capture hand movements and gestures. They aimed to record the finger position and flexion in order to classify shapes. These approaches were limited by the need for specialized hardware and the inability to capture facial expressions and other non-manual markers. Rule-based classifiers did the legwork by detecting specific input pattern of sensors to an output by programmatic rules. This approach was not practical or user-friendly.

The move to ML was occuring in parallel to hardware based approaches, as vision-based approaches were developed to overcome the limitations of the hardware based approaches and was instrumental in the development of sign language recognition. This approach used computer vision techniques to detect and track the hand and fingers. They were able to capture more information than the hardware based approaches, but were limited by the need for a controlled environment and the inability to capture facial expressions and other non-manual markers, just as the hardware based approaches were. At this stage, there were no large datasets developed, and the datasets that were available were not standardized, and were not publicly available. The vocabulary was relatively small von Agris et al. (2008), which meant that the recognition was limited to a small number of signs. The recognition was also limited to a single signer, and was not robust to variations in lighting, hand orientation, and background noise.

A great deal of focus was on feature extraction and classification, various algorithms were being pursued to extract hand features like posture. Hidden Markov Models (HMMs) were used to solve this temporal sequential task, where each sign or gesture is defined by the transition from one state to another. HMMs use the transition state to understand meaning. This approach was limited by the need for a large amount of data to train the model von Agris et al. (2008).

Another leap with vision was the usage of support vector machines (SVMs) to together with HMMs to enhance classification. SVMs were more effective at classifying spatial features such as hand shape and geolocation of digits, and videos that have depth, where gestures or shapes could look similar in 2D or 3D Vogler and Metaxas (1999).

Moving around the late 00's, a significant feat was neural networks such as used in Munib et al. (2007), which used a 3-layer network with backpropagation and Hough transform. Although 92.3% accuracy was achieved, this was still comparable to models using SVMs and HMMs, as well as the hardware based approaches before. The dataset was still limiting, with" 300 samples of hand sign images; 15 images for each sign." Munib et al. (2007).

Perhaps the greatest leap was the rise of deep learning, as neural networks got deeper in terms of model layering. Image and video processing research as a whole was in full swing, convolutional neural networks (CNNs), recurrent neural networks (RNNS), and Long short-term memory (LSTMS), were a few which had significant impact. CNNs were adept at automatically extracting and learning

Yann Lecun is credited with setting the precedence of CNNs in 1998, with the LeNet-5 architecture Lecun et al. (1998). This was a significant leap in the field of computer vision, and was the first time that a model was able to learn features automatically, which is why due to greatly increased GPU processing power, CNNs came back into the fold at the 2012 ImageNet challenge Krizhevsky et al. (2012). CNNs are ideal and particularly adept at processing data that is grid-like, as in images that have dimensions. A series of layers are used to extract and identify features by breaking down the image into smaller parts, understanding that, and over and over, and combining them to understand the whole image. Functions and pooling layers are used to optimize the output for classification.

RNNs are a type of neural network that are adept at processing sequential data, such as text, audio, and video. They are able to remember previous inputs and use that information recurrently, because the network has a directed cycle network, meaning information can persist inside the network Sherstinsky (2020). This is particularly useful for ASL recognition, as the signs are sequential, and the order of the signs is important.

LSTMs are a type of RNN that are able to remember information for long periods of time, and are able to overcome the vanishing gradient problem that is common in RNNs Sherstinsky (2020).

While these models and approaches are valid for the challenge of American Sign Language, commonly fingerspelling isn't factored in, and these models struggle to perform well on fingerspelling. This may be because fingerspelling is a sequential task where the order of the letters is important, and that the subtle differences between letters are difficult to distinguish. More so, in practise the signed spellings change rapidly and do not necessary finish a movement, the hand is continuously transitioning sequentially from one letter to the next to form a word. This is a challenge for models that are not able to capture the temporal nature of the task.

Currently, the experimental methods are Connectionist Temporal Classification (CTC) Graves et al. (2006); Shi et al. (2018a), Attention Bahdanau et al. (2016), Transformers Vaswani et al. (2023), and using large language models (LLMs) to improve accuracy among others.

Table 1: Summary and Analysis of ASL Fingerspelling Recognition Models (2018-2023)

Reference	Model Used	Framework	Dataset	Key Findings	Performance Metrics	Challenges Addressed
S Kumar et al. (2018)	RNN, LSTM, Attention, Encoder/Decoder	[Not Specified]	NCSLGR Corpus	Recognition and translation of ASL glosses	GRR: 86%, GER: 23%	Real-time recognition and translation
Weerasooriya and Ambegoda (2022)	RF, KNN, LR	[Not specified]	FASSL custom dataset	Developed a classifier for static signs using a small dataset	Accuracy: 87.9% (correct estimates)	Pose classification with limited data
Cihan Camgoz et al. (2020)	Transformers with CTC loss	PyTorch	PHOENIX14T	State-of-the-art results in recognition and translation	WER, BLEU-4 scores	Translation from sign language videos to spoken language sentences
Abiyev et al. (2020)	CNN, SSD, FCN	[Not specified]	Kaggle ASL Fingerspelling	High accuracy, vision-based translation	Accuracy: 92.21%	Real-time translation, robustness in ASL recognition
Bantupalli and Xie (2018)	CNN, LSTM, RNN	OpenCV	Self-created Dataset	Effective recognition with custom CNN model	Accuracy: 98.11%	Robust recognition in controlled environments
Kabade et al. (2023)	ResNet, Bi-LSTM, CTC, Attention	[Not specified]	ChicagoFSWild	Recognition using optical flow and attention, preprocessing for occlusions	Letter accuracy: 57%	Recognition in 'wild' conditions, occlusions
Shi et al. (2018a)	CNN, LSTM, CTC	Faster R-CNN	Custom YouTube Dataset	Improved accuracy with hand detection	Test Acc: 41.9% with CTC	Recognition in the wild, varying conditions
Shi et al. (2019b)	CNN, RNN, CTC, Attention	TensorFlow	ChicagoFSWild, ChicagoFSWild+	Enhanced recognition in uncontrolled environments	Word Error Rate: 27.2	Recognition in diverse and challenging real-world scenarios

Reference	Model Used	Framework	Dataset	Key Findings	Performance Metrics	Challenges Addressed
Shi et al. (2021)	2D/3D-CNN, Bi-LSTM	OpenPose	ChicagoFSWild, ChicagoFSWild+	Superior detection in uncontrolled environments	AP@IoU: 0.495, MSA: 0.386	Handling fine-grained handshapes and signer's pose
Nguyen and Do (2019)	1) LBP, HOG descriptors, multi-class SVM, 2) End-to-end CNN 3) CNN weights as feature extractor for Linear-kernel SVM	[Not specified]	Massey Dataset	Three diverse methods for fingerspelling recognition	Recognition rate: 97.49%, 98.23%, 98.30%	Adaptability in feature extraction and classification approaches
Chong and Lee (2018)	SVM and DNN	TensorFlow, Scikit-learn	Self-created Dataset	Comparison of SVM and DNN for ASL recognition; effective use of LOO approach for bias avoidance	Recognition rate: 72.79%, 88.79%	Multi-class classification with 36 classes (26 letters and 10 digits)
Bantupalli and Xie (2018)	CNN (Inception) for spatial features, LSTM for temporal features	TensorFlow, Keras	American Sign Language Dataset	Efficient extraction of temporal and spatial features; use of Inception and LSTM models	Accuracy up to 93% (Softmax Layer), 58% (Pool Layer)	Managing longer sequences with LSTM; preventing overfitting with dropout
Shi et al. (2022)	FSS-Net (End-to-End Model for Fingerspelling Detection and Text Matching)	[Not Specified]	ChicagoFSWild, ChicagoFSWild+	Introduced explicit temporal localization for fingerspelling search and retrieval. Demonstrated effective fingerspelling detection in varying conditions.	mAP: 0.684 (YouTube), 0.584 (DeafVIDEO), 0.629 (Misc)	Fingerspelling detection in diverse visual conditions; handling open vocabulary and arbitrary-length queries; confusion between similar handshapes; detection failures.

Reference	Model Used	Framework	Dataset	Key Findings	Performance Metrics	Challenges Addressed
Gajurel et al. (2021)	Fine-Grained Visual Attention with Transformer Model (CTC, CNN, LSTM)	[Not specified]	ChicagoFSWild	Significantly improved state-of-the-art performance in fingerspelling recognition using Transformer-based contextual attention mechanism	Letter Accuracy: 46.96 % (dev), 48.36% (test)	Addressed challenges in capturing fine-grained details in unsegmented continuous video data. Focused on improving generalization and regularization of the model.

3.4. Methods, Tools, and Techniques (RQ-1, RQ-2, RQ-3)

3.4.1. Image-based Methods

The system performs static feature extraction on each individual image. Information from the image such as position of hand, texture, shape, colour, and other features are extracted and used to classify the image. The system doesn't identify or understand the temporal nature of the task, and is limited to the information that can be extracted from the image. ML models such as CNNs and SVMs are used to classify the extracted features to specific letters. This approach wouldn't be suitable for real-time recognition.

3.4.2. Video-based Methods

This time, the system performs sequential feature extraction, by extracting temporal features the models can understand the transitions between letters. This is critical for differentiating letters that might be similar when static, or depending on the orientation. For instance, the letter ”h” and ”u” have the same hand shape, but differ in meaning depending on orientation. Models typically include RNNs and LSTMs which are particularly useful at retaining information over time, which are important for a real-time sequential task, that must understand the order of the letters.

3.4.3. Framework-based Methods and Tools

The MediaPipe framework, introduced by Lugaresi et al. (2019), is an open-source collection of libraries and tools designed to facilitate the development and deployment of artificial intelligence (AI) and machine learning (ML) applications. Its implementation is particularly beneficial for creating ML pipelines, offering a suite of pre-trained models that excel in tasks such as gesture recognition and hand segmentation. As a relatively new tool, MediaPipe is gaining traction among developers who require robust solutions for specific layers of the ML pipeline without the need to develop models from scratch.

OpenCV, as detailed by Culjak et al., is a comprehensive open-source library of optimized algorithms that provides extensive support for image and video processing tasks. Widely adopted in the field of computer vision, OpenCV is commonly utilized for detecting signs, numerals, alphabets, and more, often serving as a cornerstone in the preprocessing stages of machine learning models Srinivasan et al. (2023). Its versatility and performance make it a popular choice for researchers and practitioners working on sign language recognition and related areas.

TensorFlow Abadi et al. (2016) and PyTorch Paszke et al. (2019) are both development systems, each individual ecosystems in their own right where developers can build and train models at scale. They are both open-source, and are both widely used in the field of machine learning. TensorFlow is developed by Google, and PyTorch is developed by Meta. TensorFlow is more mature, and has a larger community, and is more widely used in production. PyTorch is more flexible, and is more popular for research and experimentation. Both frameworks are used in the development of ASL recognition models.

3.5. Comparative Analysis of Machine Learning Models (RQ-1)

S Kumar et al. (2018) and several others utilize RNN, LSTM, and Attention Mechanisms, highlighting the importance of sequential data processing in sign language. CNN combined with LSTM, as in Shi et al. (2018a), indicates the effectiveness of capturing both spatial and temporal features. The use of Transformers, such as in Cihan Camgoz et al. (2020), showcases advanced capabilities in translation tasks.

The study by Weerasooriya and Ambegoda (2022) using RF, KNN, and LR represents traditional machine learning approaches, effective for smaller datasets. In contrast, Chong's comparison Chong and Lee (2018) between SVM and DNN illustrates the evolving landscape from classical to modern neural network-based approaches. Performance Metrics and Dataset Dependency:

High accuracy in controlled environments, like Bantupalli's 98.11% accuracy Bantupalli and Xie (2018), contrasts with moderate success in 'wild' conditions, such as Kabade's 57% letter accuracy Kabade et al. (2023). The choice of datasets, ranging from custom ones to larger, more diverse datasets like PHOENIX14T or ChicagoFSWild, influences the model selection and performance, as seen across multiple studies.

Real-time recognition and translation needs, addressed in studies like Abiyev et al. (2020), demand fast and efficient models. Recognition in uncontrolled environments, as explored by Shi et al. (2019b), requires robust models capable of handling diverse and challenging scenarios.

Unique approaches like Shi's FSS-Net (Shi et al., 2022) emphasize innovation in addressing specific challenges like temporal localization and open vocabulary in fingerspelling detection. Gajurel's study (Gajurel et al., 2021) using a Transformer model with fine-grained visual attention highlights efforts in improving model generalization and handling unsegmented continuous video data.

3.6. Challenges in ASL Fingerspelling Recognition (RQ-5, RQ-6)

There are currently a numerous amount of challenges facing all types of ML models in ASL fingerspelling recognition. These challenges can be categorized into three main categories: technical, data, and real-world challenges. The variability in hand shape and motion neccitate deeper models in order to learn the complex patterns of large amounts of data Gajurel et al. (2021). Fluent signers can spell quickly and smoothly than novice signers, which can be difficult for models to capture the fluidity of the motion Gajurel et al. (2021). Depending on the angle or position of the hand, occlusion from other fingers or the body can occur, which can make it difficult for models to distinguish between letters for signers like "A" and "S" Shi et al. (2018a). Classification is harder because the model cannot see the whole hand, and the model must learn to recognize the letter from a partial view of the hand Shi et al. (2018a). Overlapping is another challenge, where the movement from one letter to another too quickly, may overlap as there is no distinct pause, or boundary between the spelt letters. Diverse background and lighting conditions. One of the first steps in any sign language task, is to detect and segment the hand from the background, which can be difficult in the wild due to the varying backgrounds, skin colours, and lighting conditions Shi et al. (2019b). This is a challenge for models that are not robust to these conditions, and can lead to poor performance.

3.7. State of the Art and Real-World Applications (RQ-4, RQ-7)

Automatic Speech Recognition (ASR) is a separate domain from Automatic Sign Language Recognition (ASLR). ASR is the task of converting speech to text and is a much more mature field. While one is an audio task, and the other a visual one, they both involve converting a sequence of symbols to text. As there is more research throughput into ASR, it's very common to see overlapping models and models adapted from ASR to ASLR. An example of this is the Conformer-CTC model Gulati et al. (2020) and the more recent advancement Squeezeformer Kim et al. (2022). Although academia is advancing the field of ASLR formally, there are many active practitioners experimenting on websites through Kaggle competitions **?**. Whilst this is not a formal academic setting, it is a good indicator of the state of the art, and the models that are being used in the real world. Furthermore, it mustn't be understated that these models and results aren't peer-reviewed, and are not necessarily easily reproducible, but a lot are and so are their methods we can be adapted. However, these developments are on the internet immediately, and aren't being kept back as research papers that could take months and years to publish. They could be used to inform the development of our own model, given the time constraints of this project.

3.8. Conclusion

The literature review extensively covers the development and challenges of ASL fingerspelling recognition, focusing on various machine learning models and their performance under diverse conditions. Research indicates that early solutions relied heavily on hardware-based systems, which were restrictive due to their need for specialized equipment and inability to capture nuanced elements like facial expressions. Transitioning into machine learning, the field saw an evolution from rule-based classifiers and early vision-based systems to more sophisticated models like Hidden Markov Models (HMMs) and support vector machines (SVMs).

Recent advancements have shifted towards deep learning, with Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Transformers showing promising results in handling the complexities of ASL fingerspelling. These models excel in capturing both spatial and temporal features necessary for effective recognition. Studies highlight the importance of dataset quality, diversity, and size, which significantly impact the performance of these models. Moreover, real-world application testing underscores the potential of these technologies to be integrated into practical solutions, such as interactive kiosks for service delivery.

The literature reviewed underscores a significant progression from manual, hardware-dependent methods to sophisticated machine learning techniques that offer more flexibility and accuracy in ASL fingerspelling recognition. While the advancements in deep learning models, particularly CNNs and RNNs, mark substantial improvements in recognition accuracy and processing speed, challenges remain. These include the need for extensive datasets, the variability in environmental conditions, and the real-time processing demands of practical applications.

The review also highlights the critical impact of choosing the right model and dataset for specific applications, suggesting a tailored approach based on the specific requirements of the use case, such as the need for real-time performance in kiosk systems. Future research should continue to explore the integration of more adaptive and robust models that can handle the variability in real-world scenarios more effectively. Additionally, fostering a closer collaboration between technological advancements and the nuanced needs of the ASL community will be essential to ensure that these solutions enhance real-life communication for deaf and hard of hearing individuals.

At the state-of-art we can conclude the Transformers with respect to other variants such as upcoming Squeeze formers are the most effective models for this task, and that for inference CTC decoding or Beam Search is actively being implemented.

4. Chapter 3: Methodology

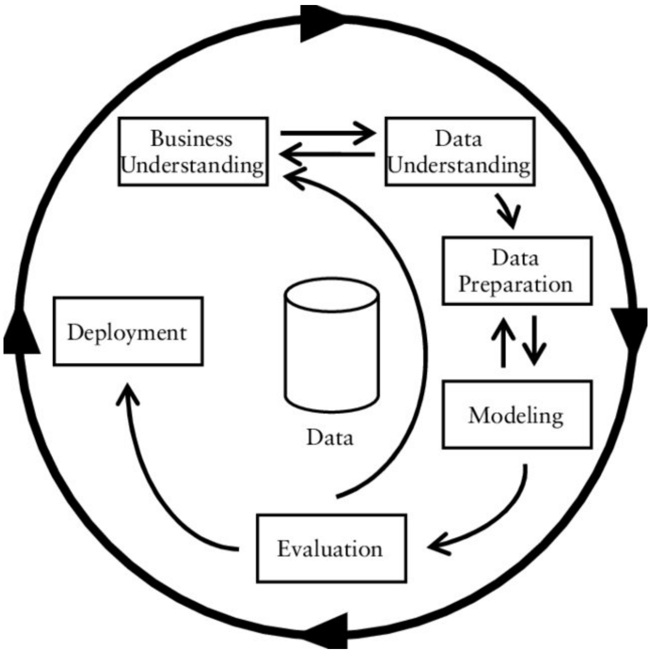

Figure 1: CRISP-DM Methdology (Löfström, 2009)

The CRoss Industry Standard Process for Data Mining (CRISP-DM) is a widely used methodology that provides a structured approach to undertaking data science projects from a process perspective. In this task, we emphasize the usage of CRISP-DM methodology as a preferable process over a standalone software development methodology, this is because the project requires a higher level of management than a naive implementation that agile gives. Agile-like processes can be integrated into CRISP-DM with usages of rapid prototyping and usual backlogging, testing, etc, but found the CRISP-DM ”pipeline” is adaptable to the real world goal of developing data science projects.

CRISP-DM is inherently iterative, it understands that whilst developing in say the evaluation phase, unexpected consequences allow us to move back to the data preparing or modelling phase to adjust. Furthermore, the feedback loops observed in Figure 1 explicitly accommodates returning to previous steps to refine insights.

The 6 phases include business understanding, data understanding, data preparation, modelling, evaluation, and deployment.

4.1. Business Understanding

The objective of the project is to increase accessibility for the Deaf and Hard of Hearing community, and by developing artificial intelligent models, this would be achieved. In order to tackle this, we must understand if we have the ability to; developing AI requires computing resources (hardware platforms), as well as software, furthermore we must understand if the datasets available exist in order to complete the objectives, and if there are any data security concerns and or legal issues.

The computer resource available to us are NVIDIA RTX 3090 GPU with 24 GB of GDDR6X memory, an AMD Ryzen 9 7950X 16-Core Processor and 64 GB of DDR5 memory. This is critical because one, seq2seq tasks can require a lot of memory and processing power from graphics cards / accelerators due to the amount of parameters and input sizes, and an NVIDIA monopoly due to the CUDA framework (CUD).

At this early stage, we must decide upon the major tools. Artificial intelligence splits into two major deep learning frameworks, and are TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019). PyTorch has a tremendous amount of modularity, and out of the box support for major models, comparing memory consumption and speed is harder than implementations of models are different, but is it generally agreed PyTorch is faster and has lower memory consumption.

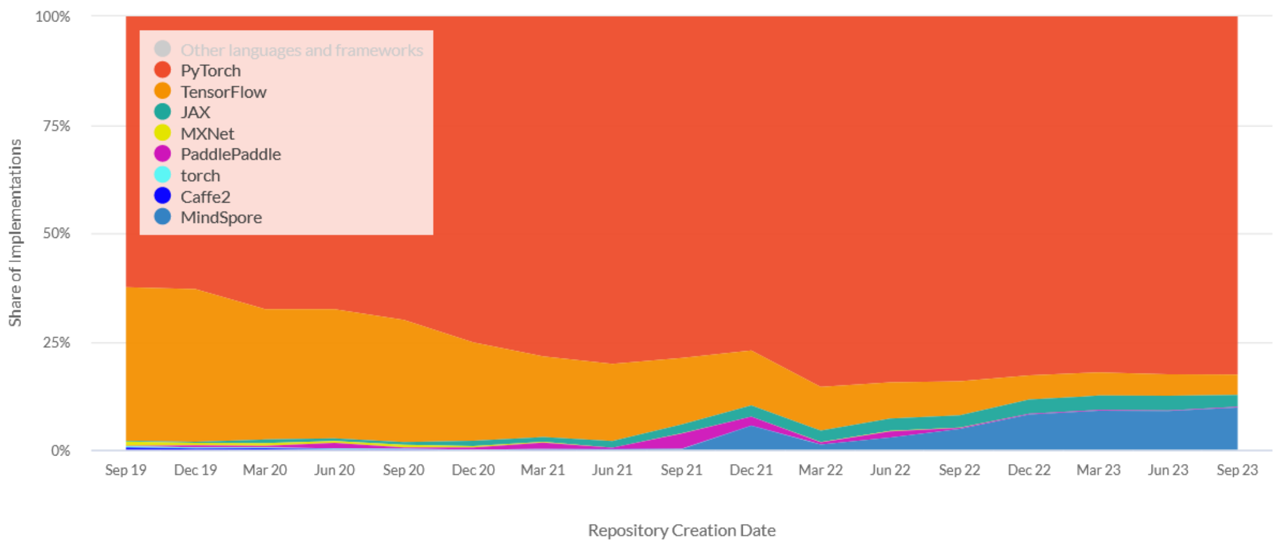

Figure 2: TensorFlow vs PyTorch (Boesch, 2023)

See Figure 2, that shows the number of created repositories, i.e. new projects with certain frameworks. For a comparison on these frameworks from Gardenz Boesch (Boesch, 2023) and anecdotal evidence from increasing usage from popular data science website Kaggle (Anthony Goldbloom and Ben Hamner), we conclude on PyTorch due to its ease of use modular system, out-of-box features, and increasing popularity shift compared to TensorFlow.

The ”ASL Fingerspelling Recognition Corpus (version 1.0) is a collection of hand and facial landmarks generated by MediaPipe version 0.9.0.1 on videos of phrases, addresses, phone numbers, and urls fingerspelling by over 100 Deaf signers” (Manfred Georg et al., 2023) released by Google with a licence to adapt and use freely, even commercially (CCD). This dataset presents an opportunity, released shortly before this projects conception in early 2023. It is a large dataset in size (180 GB) consisting of diverse conditions, number of signers, representation of signing fluency and cultures, essentially the complexity of the dataset allows a larger and more complicated model as the amount of features that can be extracted is high. This dataset can change the playing field, as previous research for ASL fingerspelling came from the ChicagoWild (Shi et al., 2018b) and ChicagoWild+ (Shi et al., 2019a) datasets, which were the largest ASL fingerspelling collections ”in the wild” conditions, essentially videos available online, predominately used by Bowen Shi (Shi et al., 2018a).

Academic research, as well as applied research has been moving away from these datasets due to legal issues regarding the source of the datasets, videos from the public realm such as YouTube were being downloaded and used without permission, this presented legal issues, and thus commerciality of models developed with it impossible would be of heightened concern.

4.2. Data Understanding

Given our chosen dataset, we must explore the data we are working with. This includes the complexity of the data, in terms of size, shape, format, distribution, and quality. We need to model the data, so we can understand the relationships between fields. The data was created using MediaPipe (Lugaresi et al., 2019), an open-source framework for creating AI pipelines, it contains various pre-trained models that can label images and videos, for groups of the body such as hands, face, and pose.

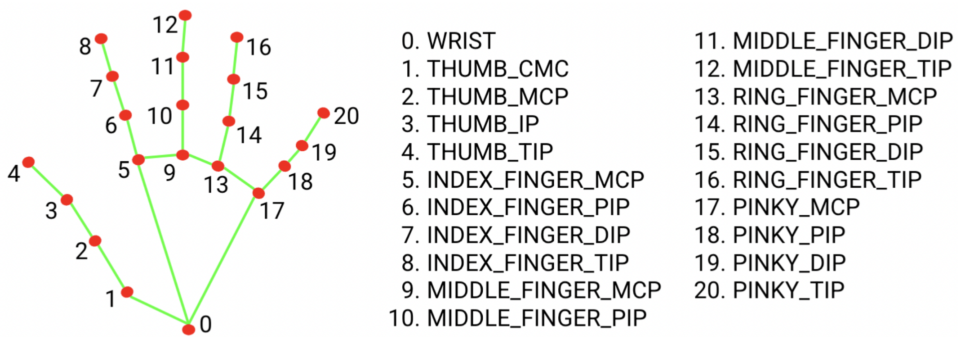

Figure 3: MediaPipe Framework Hand Landmarks (Lugaresi et al., 2019)

Figure 3 shows the mapping of landmarks for hands, which are points used to track the movement of the joints and important features, all groups have this structure. The dataset consists of 121 parquet files, columnar-storage structure for large datasets that work in a distributed environment such as Hadoop

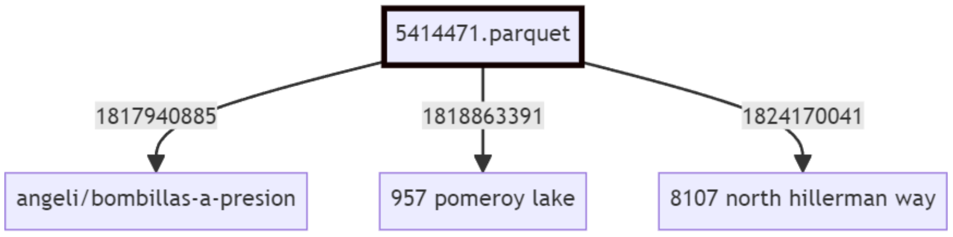

Figure 4: Dataset Structure: train.csv

The first figure, referenced as Figure 4, illustrates the train.csv file where each sequence_id corresponds to a signed phrase, linking to a separate document. Each sequence_id organizes multiple sequences or rows of frames, with the subsequent columns detailing landmarks divided into their x, y, and z coordinates.

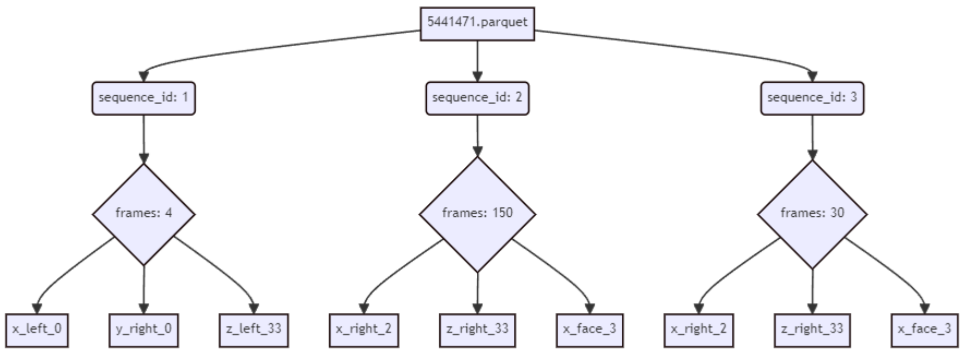

Figure 5: Parquet File Structure

The second figure, referred to as Figure 5, depicts the structure of a Parquet file, highlighting that the dataset is sequential. Analysis of the dataset reveals variability in the length of phrases and the number of frames per sequence. The dataset comprises 1,629 spatial coordinates that capture the x, y, and z positions for each of the 543 landmarks identified across categories like 'face', 'left_hand', 'pose', and 'right_hand'.

Exploring the dataset, we create a Python script, and use pip packages such as pandas, pyarrow, and numpy to load and manipulate the data. Pandas and pyarrow allows us to load the parquet into a data frame to get some characteristics of one file.

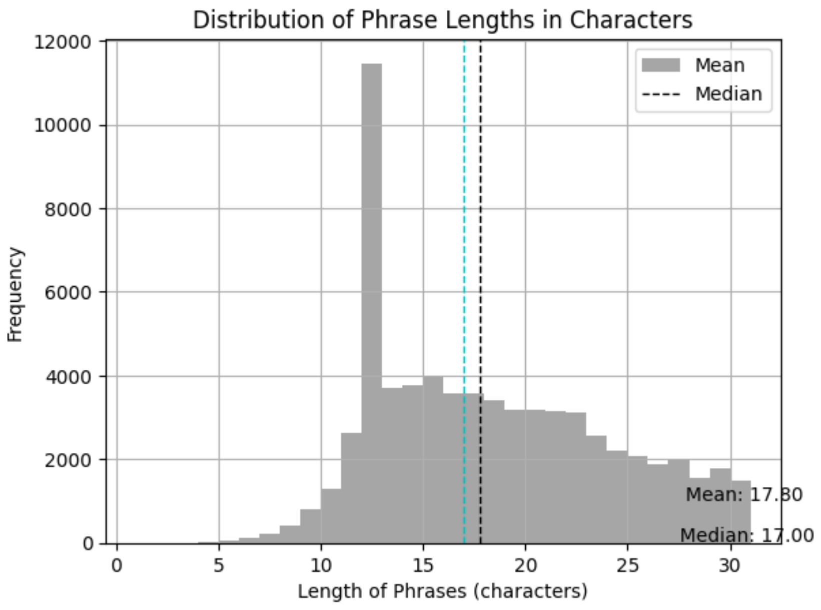

Figure 6: Distribution of Phrase Lengths (no. of characters)

The length of the phrases, by the number of characters is at most 32, there are a number of short phrases which would need to be investigated further less than 10.

The length of sequences is highly variable whereas the phrases is not so much. The length of sequences seems to reflect with the length of phrases but is weighted towards shorter sequences, as the number of frames

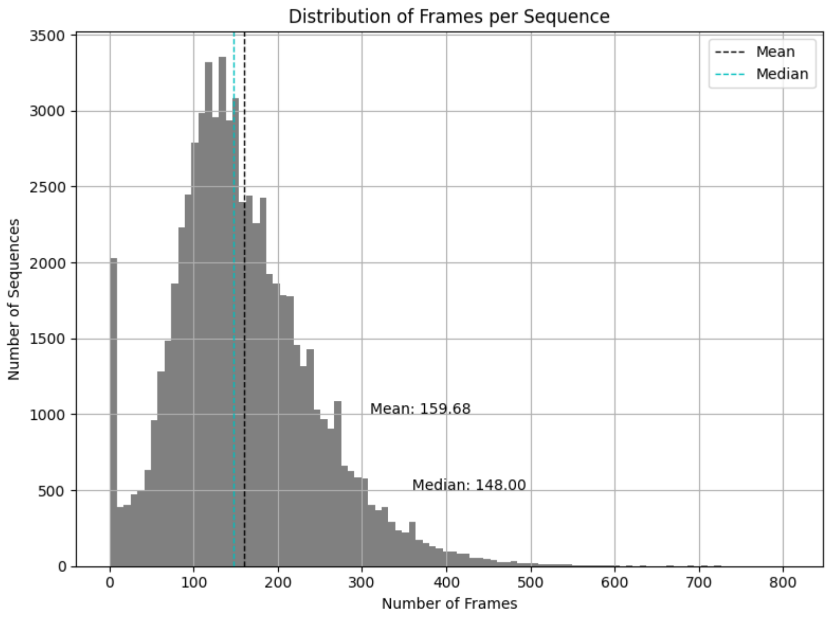

Figure 7: Distribution of Sequence Lengths (no. of frames)

increases, the number of characters in the phrase increases in order to complete the phrase. Given potential clipping errors, there seems to be outliers from 400+ frames, which might signify a failure by signer to stop the recording, or a failure in the recording process. This makes plausible sense as there are no phrases in that ball pack area. There's an understanding that phrase lengths are not tightly clipped, and unnecessary frames exist at the end, which should correlate to actual fingerspelling.

There are a number of NaN values, for where landmarks have no value, this could be due to the holistic model unable to detect, or the landmark simply wasn't in frame at that time.

4.3. Data Preparation

For the code for the data preparation and preprocessing, please refer to Appendix G. Preparing the dataset is a critical step, and can be as important as the model itself. The 121 parquet files contains the 68 parquet files totalling 95 GiB, each roughly 1.4 GiB, from the training set, and 53 parquet files from a supplemental set. The model we decide to use, will have to loop over the data given in iterations called epochs, each epoch a run through of the data, and the model will learn from the data, and adjust its weights and biases to minimize the loss function. The two parts of the dataset are nominal throughout, so we could only use the training set which reduces the overall size of the dataset to 95 GiB and inputs steps into the model per epoch.

Depending on the model chosen, certain preprocessing can happen at run-time or before the model is trained. The dataset must be prepared, so that it is in a format that most machine learning models can understand. This also includes performing augmentations, which are transformations on the data that should enhance the quality of the data, but do not necessarily change the underlying meaning of it. This is done to increase the robustness of the dataset, so that complex models can extract features without over or under fitting, and by doing so the model can train for longer. At some point data augmentation becomes trial and error.

The parquet files of the dataset are too large to be kept in memory, and the size of the data should be as minimum as possible without losing value. For the preprocessing stage we use TensorFlow's TFRecord file format, which allows us to store large amounts of data in binary format, which reduces the overall disk size used to 19.2 GiB, with the average TFRecord while being reduced over 85% to 160 MiB. This is advantageous to do because as we require constant tinkering due to the CRISP-DM methodology, we require data that can be preprocessed quickly instead of waiting for this step before training.

Furthermore, this saves our computing resources that could be crucial in allowing us the largest possible model.

There are a number of common data augmentations that are used in image and video processing, such as rotation, scaling, translation, flipping, and cropping, removing, random replacement of tokens, adding artificial noise, and more.

The file_id of each parquet file is processed to create a TFRecord file, it is read and the columns of the sequence_id as well as a selection of columns is extracted. The selection columns are the landmarks for the hands, face, and pose, that we have grouped, and specified the x, y, and z coordinates. We take a large range of the face to include lips and eyes, all the left and right hand, and 23 of the pose, which excludes the forearms.

Standard scaling for the FACE, LHAND, RHAND, POSE is performed with respect with mean and standard deviation to normalize the data so that essentially all landmarks of that group are equally treated so that larger valued features aren't prioritized. This can increase model performance of models with gradient descent optimization and neural networks with non-linear activation functions.

Referring back to Figure 7 we recall different chunks at frames 256 and 384, we will use these as the maximum frame lengths. The sequences that are above this maximum length we don't want to truncate as the rest of the frames could be important, and we'd lose valuable data compared to its target. Interpolation is used to create a sequence of frames that extrapolates a linear trend from the existing frames beyond the maximum length. This is an attempt to maintain the integrity of the data, but to shorten it.

Next, the unhandled NaN values are filled with 0s as the model will simply not be able to process NaNs. The NaNs, now 0s, are counted for the left and right hand, which are used to create a process whereby we remove corrupted data that is short, and has no hands. Mikel Bober-Irizar found that by using the levenstein distance and greedy decoding, found the average string which is used to fill in (Mikel Bober-Irizar).

Sequences where twice the length of the target phrase is less than the maximum amount of NaNs from the left or right hand are serialized and moved into a schema where a single sequence contains all its landmarks and corresponding frames equating to the target phrase it represents. It is then written to file. The features are stored as a FloatList, and the phrase is stored as a ByteList with utf-8 encoding.

This whole process is used with Python's multiprocessing library, which allows us to run the process a number of items at once, keeping in sync. This speeds up processing by a factor of the number of processes allocated. Preprocessing that takes 10 minutes with 8 cores takes roughly 1.25 minutes.

5. Chapter 4: System Design and Architecture

5.1. Modelling

At this point, we have a sufficient pipeline of preprocessing. Given our literature review, there are a multitude of machine learning and deep learning models. Transformers are still state-of-the-art in seq2seq tasks, and our choice of PyTorch as a framework we must perform our preprocessing steps that will occur at runtime.

A Transformer in PyTorch has an expectation that the shape of the data at the input is [batch size, sequence length, features]. The batch size is the number of sequences that are inputted at once, the sequence length is the number of frames in the sequence, and the features are the number of landmarks. PyTorch has a system whereby data pipes are used to modularly create a stream of data, this can be used to create a data loader that can be used to feed the model, this can be done in parallel with other processes to increase input.

Referring to Appendix F, specifically F.8 for a large overview of the pre-processing that is completed to serve the Transformer as well as Appendix G for the implementation.

The usage of data pipes allows us to iteratively stack pipes, and perform transformations on the data, the newly created TFRecords are split 80:20 for the train and validation set. The config of the pipe is used to slice TFRecords depending on where we want the index to start and end, take a training set which would be 0-96 for the full set. A shuffler is used to list and buffer TFRecords into memory, the iterable pipe then opens the file in binary mode, opens and decompressed the file, loading an Example.

Previously, during the creation of the TFRecords, we encapsulated the features and phrases of the dataset into a structure known as an Example. Each Example is a serialized representation of a single data instance, and in our setup, it was defined to include a sequence of landmarks, frames, and a target phrase. Once deserialized, these records are processed through tokenization. Tokenization can vary, from ordinal to one-hot encoding, but it essentially involves converting the characters of the phrases into numerical formats. For use with a Transformer model, this conversion generates a list of tensors. Additionally, we prepend a start token at the beginning of each sequence and append an end token at the end to facilitate processing. The tokens are important for the model to understand the beginning and end of a sequence, and to differentiate between the two. A model should learn when to start and stop predicting additional tokens.

Transformers are designed to handle variable-length sequences by padding them to a uniform length within each batch. This is why the maximum sequence length was previously set to either 256 or 384. With a batch size of 128, both the sequence frames and the tokenized phrases are padded with specific tokens to reach this maximum length. For the sequence frames, a padding token of 0 is used, and for the tokenized phrases, a special class token indexed as 61 is used, ensuring all sequences within a batch are of the same length.

At the same time, we calculate and pass on a list of the original lengths of the sequences frames and target phrase before padding.

For an in-depth explanation of Transformers, refer to Vaswani et al. (2023). Other than changing the parameters to fit the dimensionality and task of our dataset, the original Transformer uses positional encoding to give the model information about the order and position of tokens in a sequence. We employ learned positional embeddings instead, which are trained alongside the model. This allows the model to learn a more optimal, specific positional encoding for this task.

Our model has an encoder and a decoder, it has 8 encoder layers, 2 decoder layers, and 4 heads each. The input is passed through an 3 1d convolutional layers to extract features, and concatenated with an embedding layer, and passed through the encoder. The target tokens are passed through an embedding layer to generate token embeddings, and are concatenated with another embedding layer. This gives the tokens positional information which should provide more nuanced relationships.

The lengths of the source and phrases that we retained earlier, are used to mask the inputs to the model. Transformer will attend to all tokens, but we want the model to not attend to padding tokens, as well as being able to cheat and lookahead. We provide padding masks that apply 0s to the padding tokens, and the Transformer doesn't attend to these tokens are the score is minimal. We also provide a look-ahead mask for the target, which is a triangular matrix that masks the future tokens. These are critical for the model to learn that it is decoding, and not classifying, as the model should only be predicted the next token given the previous.

There were a number of additional steps that were used to modify the model, such as label smoothing, which is a regularization technique that prevents the model from becoming too confident in its predictions. We also experimented with increased dropout, and weight decay of the optimizer. We used a fused AdamW optimizer, an optimizer is controller that updates the weights and biases of the parameters in the model per step or batch given the returned loss function. AdamW was used with a stacked learning rate scheduler, that gave a warm-up period of 10 epochs followed by 190 epochs with CosineDecay scheduler, a scheduler is a function that changes the learning rate of the optimizer slowly over time. This allows the gradients and parameters to "warm up" and then slowly converge to a minimum, this is important as the model can get stuck in local minima, and not converge to a global minimum.

After hand rolling everything we moved to a library called PyTorch Lightning. After hundreds and hundreds of lines of PyTorch code, Lightning (William Falcon) greatly reduces boilerplate, has a lot of optimization and allows you to focus on what's important. We were able to change the precision of our model from 32-bit to bf16-mixed, which is a mixed precision training that uses 16-bit floating point numbers for the model weights and 32-bit for the gradients. This reduces the memory usage and speeds up training, by almost 2x.

A lot of manual work is moved into simple Lightning modules which are wrappers around your model architecture, it also has a data module to encompass the data pipeline, and a trainer to control the training, validation and prediction loops. This allows us to greatly enhance the ease of use of PyTorch natively.

In this stage, we were constantly iterating to find something that was good enough to move in that direction with.

6. Chapter 6: Testing and Validation

6.1. Evaluation

During the early stages of development, the model was evaluated using Ray Tune (Ray). Initially, due to simpler model architectures and lower memory requirements, it was feasible to concurrently run multiple instances (typically 2 or 4) to test various hyperparameters. These parameters included batch size, gradient accumulation, learning rates, feature dimensionality, the number of heads and layers, sequence length, type of optimizer, fusion techniques, and schedulers. This comprehensive testing was crucial to understand how different configurations influenced the model's performance on the dataset.

As the training progresses, the model utilizes the CrossEntropy loss function to gauge the discrepancy between the predicted probabilities and the actual target labels across 62 classes. Each training step involves generating a probability for the next token, with the loss being calculated as the negative logarithm of the probability assigned to the correct token, summed across all predictions. Following the loss calculation, the model undergoes back propagation—adjusting its weights and biases to minimize the loss. This optimization is repeated iteratively with each batch. Ideally, this should result in a reduction of training loss over time. Simultaneously, it is expected that the validation loss should decrease at a comparable or better rate, contingent on the level of regularization applied.

Referring to Figure H.10 substantial regularization, such as L1/L2 and dropout, intentionally impairs the training performance by temporarily disabling numerous neurons within the Transformer's feed-forward network. This approach, while potentially increasing training loss, enhances the model's ability to generalize to unseen data. This explains why the training loss may occasionally exceed the validation loss. This allows the model training to remain on the rails for longer.

Table 2: Edit Distance Comparison

Predicted	Target	Edit Distance
+ wust tannot figure this out	i just cannot figure this out	3.0
+t r ahoulders anees and toes	head shoulders knees and toes	6.0
+he ftn of the parts	the sum of the parts	4.0
+o nou gike to go camping	do you like to go camping	3.0
+aare ts a high priorita	space is a high priority	5.0
+o n good deed to some	do a good deed to someone	5.0
+ good stimulus deserves a bood response	a good stimulus deserves a good response	2.0
+ou aall aose your aoice	you will lose your voice	5.0
+ware did you get tuch a silly idea	where did you get such a silly idea	4.0

After inference of 5137 targets, we logged the predicted string, target string, and the edit distance. The model quite quickly understood what the end token was, and so we eventually assigned most if not all padding tokens as end tokens. For inference, we sliced the string from the first end token, to the target without the start token and end token. This was used to calculate the mean levenstein distance, of which was 4.7, and with a mean length of 28.7 for the target phrase. This gives us an error rate of 16.37%.

We used Lightning to perform inference using the predict step, which was used on a holdout set, which was a set of data that the model had never seen before. The Levenshtein or edit distance, is the minimum number of single-character edits required to change one word into another. This was used to calculate the distance between the predicted phrase and the target phrase. The edit distance was calculated for each phrase in the holdout set, and the average was taken. The lower the edit distance, the better the model is at predicting the target phrase. 2 shows a selection of the predicted phrases, the target phrases, and the edit distance between them. It seems we have a bug in our code, the predicted phrase starts on the first character, but it doesn't seem to make a prediction despite having an input yet. However, it is still quite robust at long and short phrases, and is definitely readable.

7. Chapter 7: Management of Project

In regard to the project management, CRISP-DM invaluable in setting out how to begin and in what steps, which was preferred over a software development methodology alone. The project was managed using a Gantt chart J.14, which was used to plan and track the project's progress. It worked very well up until March, the project was on track, and our risk register J.15 included a high potential risk of overflowing project time due to project complexity. Project halted for a whole month as the model was not performing as expected, and so the project was behind schedule. The project was able to recover, this was possible down to streamlining the development process by bringing in Ray Tuning and Lightning. This allowed the model to be training multiple hyperparameters at all times.

The usage of work break down documents such as J.12 were more prominent early on, they allowed us to set expectations of the work that was required and to move in a structured way. As we moved deeper into the model development in regard to augmentations, these became less useful as the work was more iterative.

8. Ethical Considerations Chapter

The dataset used could be used identify or track individuals, although the consent was given to use their data, we mustn't attempt to derive any identification.

9. Legal Considerations Chapter

Not applicable.

10. Conclusion Chapter

This Individual Project documents the conceptualization, development, and evaluation of a machine learning model designed to translate American Sign Language (ASL) fingerspelling videos into text. As the final chapter of this journey, it encapsulates the full scope of the research, development efforts, and key findings, while critically assessing the achievements and limitations encountered along the way.

The primary aim of the project, Part A was to develop a machine learning model capable of interpreting ASL fingerspelling and predicting the corresponding text with a realistic degree of accuracy. This goal was ambitiously set against a backdrop of limited research and development in the niche area of ASL fingerspelling recognition. Despite these challenges, the project has successfully demonstrated that it is indeed feasible to use advanced AI techniques to interpret and translate ASL fingerspelling into text using the new ASL Fingerspelling dataset. The model achieved an error rate of 16.37% in predicting target phrases, a significant accomplishment for a novice given the complexity of the task and the variability inherent in human sign language. This requires further research to confirm these results.

However, the secondary objective—Part B, which aimed to develop a user-friendly application that allows for the input of ASL fingerspelling videos and outputs corresponding text was not achieved. This shortfall was primarily due to the model's performance not meeting the expected benchmarks necessary for a seamless user application in time. This outcome highlights a critical learning point: the necessity of setting realistic expectations and preparing for iterative developments when tackling such complex AI-driven projects.

Throughout the project, extensive research and a rigorous literature review guided the strategic and technical decisions. The CRoss Industry Standard Process for Data Mining (CRISP-DM) methodology provided a structured approach, emphasizing an iterative process that was crucial in navigating the unexpected challenges and refining the project's direction. The selection of PyTorch as the development framework due to its modularity and the employment of the Transformer model architecture were decisions that underpinned the technical success of the model.

This project has contributed to the academic field and offers a proof that advances the technology available for improving communication accessibility for the Deaf and Hard of Hearing communities. It underscores the potential of machine learning in bridging communication gaps through the translation of ASL fingerspelling into text, setting a precedent for future research in this area.

Several recommendations can be made based on the lessons learned for future projects in this domain. These include:

Iterative Testing and Development: Continuous testing phases are crucial, especially in projects involving complex AI models. These phases should be designed to not only assess the accuracy but also the applicability of the model in real-world scenarios.

Robust Dataset Utilization: The importance of a diverse and extensive dataset cannot be overstated. Future projects should aim to expand the dataset used, possibly incorporating more dynamic and varied environmental conditions to train more robust models.

User-Centric Design: Part B of the project's aim highlights the importance of aligning technical achievements with user needs. Future projects should integrate user experience design earlier in the process to ensure that the end product is both functional and user-friendly.

In summary, this project provides a strong foundation for novices in AI for future work and outlines that the most state-of-art models such as Squeezeformer are not always necessarily feasible or worthwhile.

References

, . CC BY 4.0 Deed | Attribution 4.0 International | Creative Commons. URL: https://creativecommons.org/licenses/by/4.0/.

, . CUDA Toolkit - Free Tools and Training. URL: https://developer.nvidia.com/cuda-toolkit.

, . Ray Train: Scalable Model Training — Ray 2.21.0. URL: https://docs.ray.io/en/latest/train/train.html.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X., 2016. TensorFlow: A system for large-scale machine learning. doi:10.48550/arXiv.1605.08695, arXiv:1605.08695.

Abiyev, R., Idoko, J.B., Arslan, M., 2020. Reconstruction of Convolutional Neural Network for Sign Language Recognition, in: 2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE), IEEE, Istanbul, Turkey. pp. 1–5. doi:10.1109/ICECCE49384.2020.9179356.

Anthony Goldbloom, Ben Hamner, . Kaggle: Your Home for Data Science. URL: https://www.kaggle.com/.

Bahdanau, D., Cho, K., Bengio, Y., 2016. Neural Machine Translation by Jointly Learning to Align and Translate. doi:10.48550/arXiv.1409.0473, arXiv:1409.0473.

Bantupalli, K., Xie, Y., 2018. American Sign Language Recognition using Deep Learning and Computer Vision, in: 2018 IEEE International Conference on Big Data (Big Data), pp. 4896–4899. doi:10.1109/BigData.2018.8622141.

Boesch, G., 2023. Pytorch vs Tensorflow: A Head-to-Head Comparison. URL: https://viso.ai/deep-learning/pytorch-vs-tensorflow/.

Chong, T.W., Lee, B.G., 2018. American Sign Language Recognition Using Leap Motion Controller with Machine Learning Approach. Sensors 18, 3554. doi:10.3390/s18103554.

Cihan Camgoz, N., Koller, O., Hadfield, S., Bowden, R., 2020. Sign Language Transformers: Joint End-to-End Sign Language Recognition and Translation, in: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, Seattle, WA, USA. pp. 10020–10030. doi:10.1109/CVPR42600.2020.01004.

Culjak, I., Abram, D., Pribanic, T., Dzapo, H., Cifrek, M., . A brief introduction to OpenCV .

Ethnologue, 2023. American Sign Language | Ethnologue Free. URL: https://www.ethnologue.com/language/ase/.

Gajurel, K., Zhong, C., Wang, G., 2021. A Fine-Grained Visual Attention Approach for Fingerspelling Recognition in the Wild, in: 2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC), IEEE, Melbourne, Australia. pp. 3266–3271. doi:10.1109/SMC52423.2021.9658982.

Graves, A., Fernández, S., Gomez, F., Schmidhuber, J., 2006. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks, in: Proceedings of the 23rd International Conference on Machine Learning, Association for Computing Machinery, New York, NY, USA. pp. 369–376. doi:10.1145/1143844.1143891.

Gulati, A., Qin, J., Chiu, C.C., Parmar, N., Zhang, Y., Yu, J., Han, W., Wang, S., Zhang, Z., Wu, Y., Pang, R., 2020. Conformer: Convolution-augmented Transformer for Speech Recognition. doi:10.48550/arXiv.2005.08100, arXiv:2005.08100.

Hotz, N., 2018. What is CRISP DM? URL: https://www.datascience-pm.com/crisp-dm-2/.

Kabade, A.E., Desai, P., C, S., G, S., 2023. American Sign Language Fingerspelling Recognition using Attention Model, in: 2023 IEEE 8th International Conference for Convergence in Technology (I2CT), pp. 1–6. doi:10.1109/I2CT57861.2023.10126277.

Kim, S., Gholami, A., Shaw, A., Lee, N., Mangalam, K., Malik, J., Mahoney, M.W., Keutzer, K., 2022. Squeezeformer: An Efficient Transformer for Automatic Speech Recognition. doi:10.48550/arXiv.2206.00888, arXiv:2206.00888.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. ImageNet Classification with Deep Convolutional Neural Networks, in: Advances in Neural Information Processing Systems, Curran Associates, Inc.. pp. 1097–1105. URL: https://proceedings.neurips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

Lecun, Y., Bottou, L., Bengio, Y., Haffner, P., 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE 86, 2278–2324. doi:10.1109/5.726791.

Löfström, T., 2009. Utilizing Diversity and Performance Measures for Ensemble Creation. Ph.D. thesis.

Lugaresi, C., Tang, J., Nash, H., McClanahan, C., Uboweja, E., Hays, M., Zhang, F., Chang, C.L., Yong, M.G., Lee, J., Chang, W.T., Hua, W., Georg, M., Grundmann, M., 2019. MediaPipe: A Framework for Building Perception Pipelines. doi:10.48550/arXiv.1906.08172, arXiv:1906.08172.

Manfred Georg, Phil Culliton, Sam Sepah, Sohier Dane, Thad Starner, Ashley Chow, Glenn Cameron, 2023. Google - american sign language fingerspelling recognition. URL: https://kaggle.com/competitions/asl-fingerspelling.

Mikel Bober-Irizar, . Static Greedy Baseline [0.157 LB]. URL: https://kaggle.com/code/anokas/static-greedy-baseline-0-157-lb.

Mitchell, R.E., Young, T.A., Bachelda, B., Karchmer, M.A., 2006. How Many People Use ASL in the United States?: Why Estimates Need Updating. Sign Language Studies 6, 306–335. URL: `https://www.jstor.org/stable/26190621`, `arXiv:26190621`.

Munib, Q., Habeeb, M., Takruri, B., Al-Malik, H.A., 2007. American sign language (ASL) recognition based on Hough transform and neural networks. Expert Systems with Applications 32, 24–37. doi:`10.1016/j.eswa.2005.11.018`.

Nguyen, H.B., Do, H.N., 2019. Deep Learning for American Sign Language Fingerspelling Recognition System, in: 2019 26th International Conference on Telecommunications (ICT), IEEE, Hanoi, Vietnam. pp. 314–318. doi:`10.1109/ICT.2019.8798856`.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S., 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. doi:`10.48550/arXiv.1912.01703`, `arXiv:1912.01703`.

S Kumar, S., Wangyal, T., Saboo, V., Srinath, R., 2018. Time Series Neural Networks for Real Time Sign Language Translation, in: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), IEEE, Orlando, FL. pp. 243–248. doi:`10.1109/ICMLA.2018.00043`.

Saeed, Z.R., Zainol, Z.B., Zaidan, B.B., Alamoodi, A.H., 2022. A Systematic Review on Systems-Based Sensory Gloves for Sign Language Pattern Recognition: An Update From 2017 to 2022. IEEE Access 10, 123358–123377. doi:`10.1109/ACCESS.2022.3219430`.

Sherstinsky, A., 2020. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network. Physica D: Nonlinear Phenomena 404, 132306. doi:`10.1016/j.physd.2019.132306`, `arXiv:1808.03314`.

Shi, B., Brentari, D., Shakhnarovich, G., Livescu, K., 2021. Fingerspelling Detection in American Sign Language, in: 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, Nashville, TN, USA. pp. 4164–4173. doi:`10.1109/CVPR46437.2021.00415`.

Shi, B., Brentari, D., Shakhnarovich, G., Livescu, K., 2022. Searching for fingerspelled content in American Sign Language. URL: `http://arxiv.org/abs/2203.13291`, `arXiv:2203.13291`.

Shi, B., Del Rio, A.M., Keane, J., Michaux, J., Brentari, D., Shakhnarovich, G., Livescu, K., 2018a. American Sign Language Fingerspelling Recognition in the Wild, in: 2018 IEEE Spoken Language Technology Workshop (SLT), pp. 145–152. doi:`10.1109/SLT.2018.8639639`.

Shi, B., K. Livescu, D. Brentari, G. Shakhnarovich A., Martinez Del Rio,, J. Keane, 2018b. American Sign Language fingerspelling recognition in the wild. SLT .

Shi, B., Livescu, K., D. Brentari, G. Shakhnarovich, A. Martinez Del Rio, J. Keane, 2019a. Fingerspelling recognition in the wild with iterative visual attention. ICCV .

Shi, B., Rio, A.M.D., Keane, J., Brentari, D., Shakhnarovich, G., Livescu, K., 2019b. Fingerspelling Recognition in the Wild With Iterative Visual Attention, in: 2019 IEEE/CVF International Conference on Computer Vision (ICCV), IEEE, Seoul, Korea (South). pp. 5399–5408. doi:10.1109/ICCV.2019.00550.

Srinivasan, R., Kavita, R., Kavitha, M., Mallikarjuna, B., Bhatia, S., Agarwal, B., Ahlawat, V., Goel, A., 2023. Python And Opencv For Sign Language Recognition, in: 2023 International Conference on Device Intelligence, Computing and Communication Technologies, (DICCT), pp. 1–5. doi:10.1109/DICCT56244.2023.10110225.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2023. Attention Is All You Need. doi:10.48550/arXiv.1706.03762, arXiv:1706.03762.

Vogler, C., Metaxas, D., 1999. Parallel hidden Markov models for American sign language recognition, in: Proceedings of the Seventh IEEE International Conference on Computer Vision, pp. 116–122 vol.1. doi:10.1109/ICCV.1999.791206.

von Agris, U., Zieren, J., Canzler, U., Bauer, B., Kraiss, K.F., 2008. Recent developments in visual sign language recognition. Universal Access in the Information Society 6, 323–362. doi:10.1007/s10209-007-0104-x.

Weerasooriya, A.A., Ambegoda, T.D., 2022. Sinhala Fingerspelling Sign Language Recognition with Computer Vision, in: 2022 Moratuwa Engineering Research Conference (MERCon), IEEE, Moratuwa, Sri Lanka. pp. 1–6. doi:10.1109/MERCon55799.2022.9906281.

William Falcon, . Welcome to PyTorch Lightning — PyTorch Lightning 2.2.3 documentation. URL: https://lightning.ai/docs/pytorch/stable/.

Appendices

Appendix A. Glossary

ASL (American Sign Language) - A complete, natural language that serves as the predominant sign language of Deaf communities in the United States and most of Anglophone Canada.

Fingerspelling - The use of hand movements to spell out words and phrases using a manual alphabet in sign language.

CRISP-DM (Cross Industry Standard Process for Data Mining) - A widely used data mining process model that describes common approaches used by data mining experts and lists typical phases of a project.

TensorFlow and PyTorch - Open-source machine learning libraries for research and production, providing a range of tools, libraries, and community resources that let researchers create complex machine learning algorithms and developers easily build and deploy ML powered applications.

Transformer Model - An architecture that handles sequential data without the use of recurrence. It uses self-attention to weigh the significance of different words in a sequence.

MediaPipe - A framework for building multimodal (video, audio, any time-series data) applied machine learning pipelines, developed by Google.

Parquet File - A columnar storage file format available to any project in the Hadoop ecosystem, designed to provide efficient data compression and encoding schemes.

TFRecord - A simple record-oriented binary format that works well with TensorFlow. It allows for data to be serialized and read with a structure that TensorFlow is optimized to handle.

Data Augmentation - Techniques used to increase the amount of data by adding slightly modified copies of already existing data or newly created synthetic data from existing data.

Seq2Seq (Sequence-to-Sequence) - A category of machine learning models that translates a sequence of items (such as words in a sentence) from one domain into another.

Landmarks - Specific points on an object that are used to track the object's properties, such as position or orientation. Often used in gesture recognition to track the positions of hands or face.

Hyperparameters - Parameters whose values are used to control the learning process and must be set before training (as opposed to the values that are learned during training).

Edit Distance (Levenshtein Distance) - A measure of the similarity between two strings, which is the minimum number of operations required to transform one string into the other.

Epochs - Complete passes over the entire dataset during training, used to help the model learn from the data iteratively.

Batch Size - The number of training examples utilized in one iteration.

Gradient Descent Optimization - A method to minimize an objective function by moving in the direction of steepest descent as defined by the negative of the gradient.

Lookahead Masking and Padding Masking - Techniques used in Transformer models to prevent the model from cheating by using future tokens or irrelevant padding in the sequence.

PyTorch Lightning - A lightweight PyTorch wrapper for high-performance AI research. Simplifies the process of running new experiments and reduces boilerplate code.

CUDA Framework - A parallel computing platform and API model created by NVIDIA allowing for dramatic increases in computing performance by harnessing the power of the GPU.

Sequential Data - Data that is ordered sequentially, often containing dependencies between successive elements, common in time-series data or language processing.

Tokenizer - A tool used in data preprocessing, particularly in natural language processing, that converts text into a format that can be fed into a model, usually by splitting text into words or symbols.

Padding - The process of standardizing the lengths of sequences by adding a specific value, usually zero, to sequences shorter than a specified length during data preprocessing.

Cross-Entropy Loss - A loss function commonly used in classification tasks, which measures the performance of a classification model whose output is a probability value between 0 and 1.

Back Propagation - A method used in artificial neural networks to improve the model by iteratively adjusting the weights of neurons in order to minimize the difference between the actual output and the desired output.

Regularization (L1/L2, Dropout) - Techniques used to reduce overfitting in machine learning models by penalizing model complexity or randomly dropping units during training.

Feed-Forward Network - A type of neural network where connections between the nodes do not form a cycle. This is the simplest type of artificial neural network.

Positional Encoding - A technique used in models that handle sequential data (like Transformers) to indicate the relative position of data points in a sequence.

Learned Positional Embeddings - Unlike fixed positional encodings, these are learned during training and can adapt to the specificities of the data.

Label Smoothing - A regularization technique often used in classification problems to make the model less confident about its predictions by softening the target labels.

AdamW Optimizer - A variant of the Adam optimizer that decouples the weight decay from the optimization steps, typically leading to better training stability.

Learning Rate Scheduler - Component in training optimization that adjusts the learning rate over time, typically reducing the learning rate according to a predefined schedule to help the model converge to a better minimum.

Cosine Decay Scheduler - A strategy for reducing the learning rate following a cosine curve, which starts high and gradually lowers, simulating a restart at the end of each cycle.

Warm-Up Period - A phase at the beginning of training where the learning rate gradually increases from a lower value to its initial set value, which can help in stabilizing the learning process.

Triangular Matrix - In the context of Transformers, used in the look-ahead mask to ensure that the predictions for a particular position can only depend on known outputs at positions before it.

Mixed Precision Training - Technique that uses both 16-bit and 32-bit floating-point types during model training for improving computational efficiency and memory usage without significant loss in model performance.

Ray Tune - A Python library for experiment execution and hyperparameter tuning at scale, which helps in optimizing model performance.

Appendix B. Marking Scheme

Given this was closer to a research project, please refer to the Default marking schema.

Appendix C. Changes to the Project Initiation Document

Not applicable.

Appendix D. Current Environment Investigation Report

Not applicable.

Appendix E. Requirements Specification

Not applicable.

Appendix F. Design Report

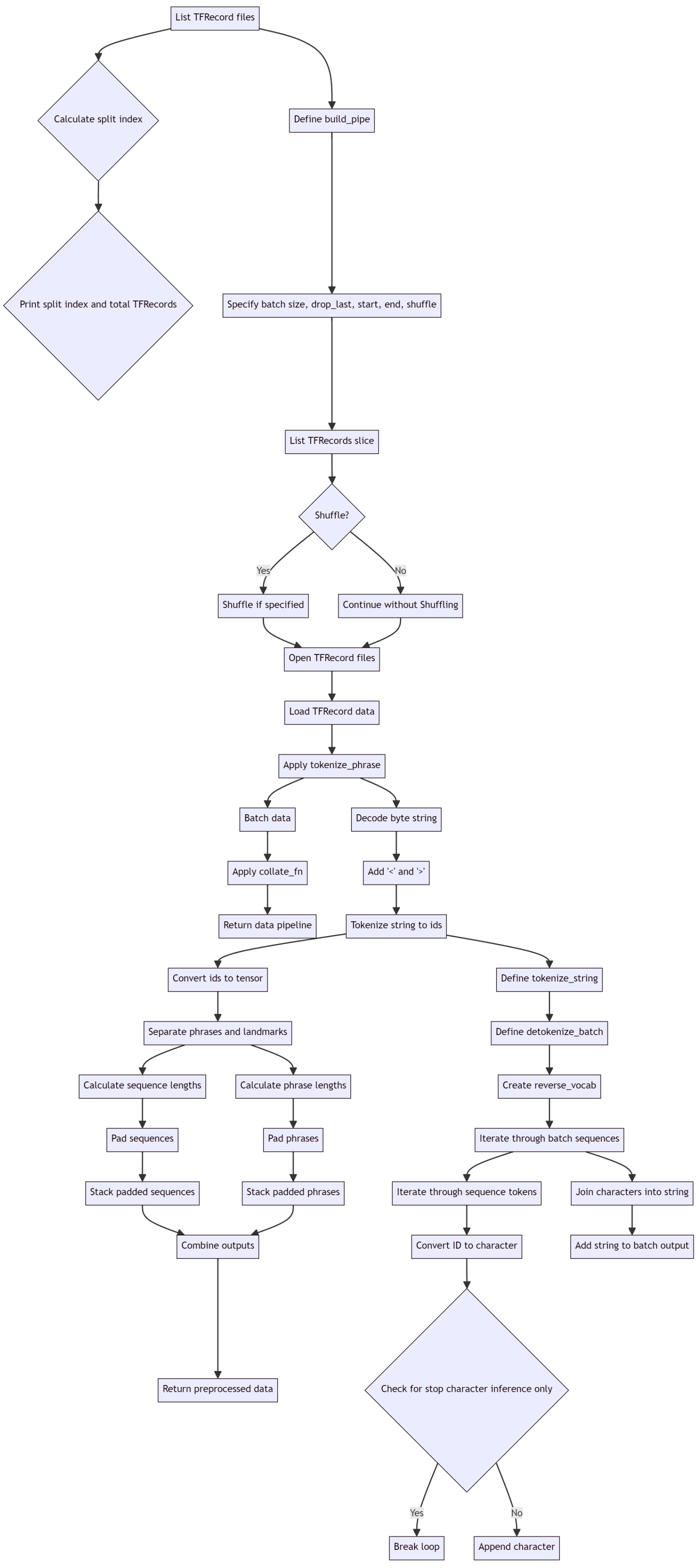

Figure F.8: Pre-processing at Runtime Model

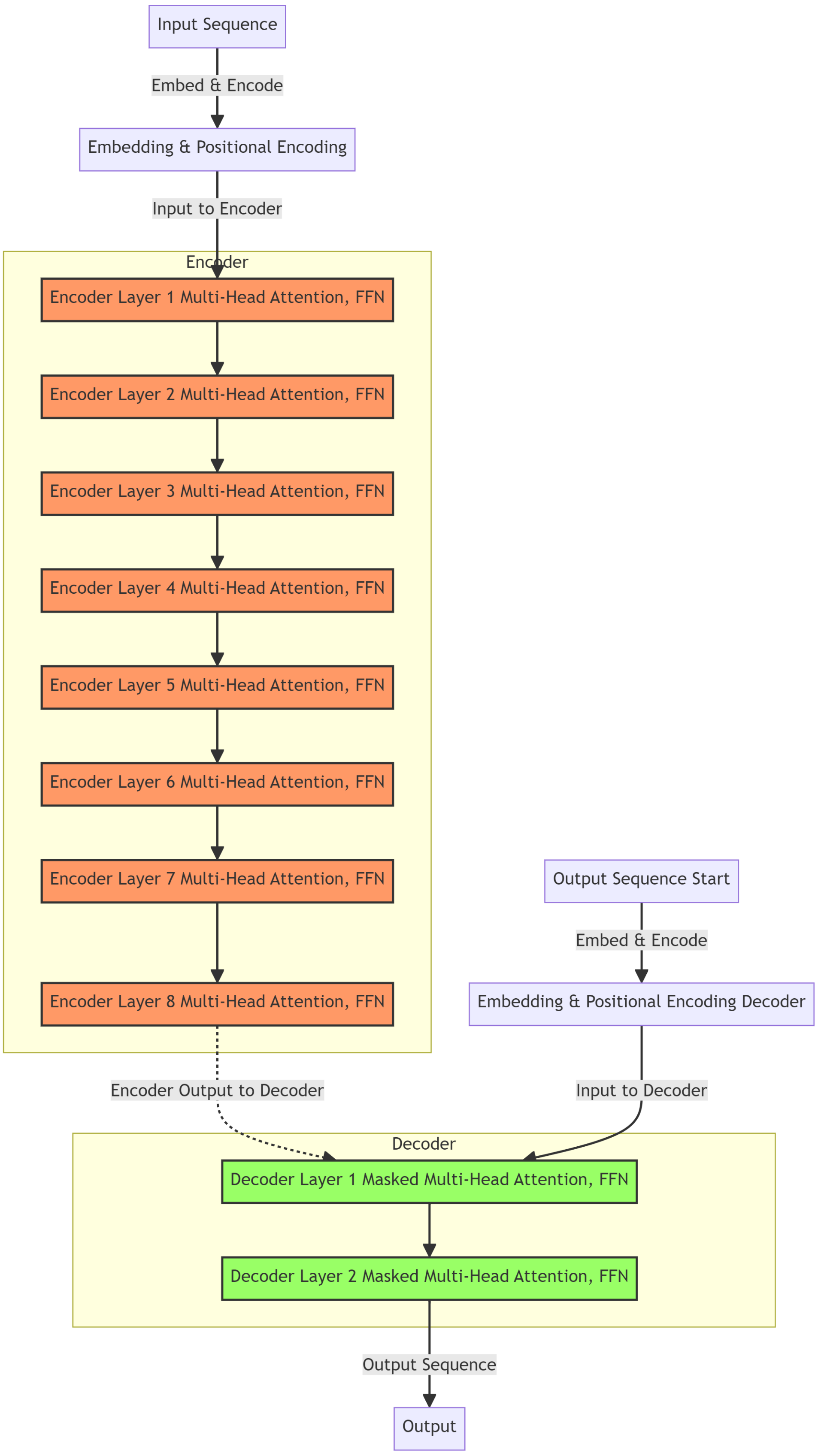
Input Sequence
Embed & Encode
Embedding & Positional Encoding
Input to Encoder
Encoder
Encoder Layer 1 Multi-Head Attention, FFN
Encoder Layer 2 Multi-Head Attention, FFN
Encoder Layer 3 Multi-Head Attention, FFN
Encoder Layer 4 Multi-Head Attention, FFN
Encoder Layer 5 Multi-Head Attention, FFN
Encoder Layer 6 Multi-Head Attention, FFN
Encoder Layer 7 Multi-Head Attention, FFN
Encoder Layer 8 Multi-Head Attention, FFN
Output Sequence Start
Embed & Encode
Embedding & Positional Encoding Decoder
Encoder Output to Decoder
Input to Decoder
Decoder
Decoder Layer 1 Masked Multi-Head Attention, FFN
Decoder Layer 2 Masked Multi-Head Attention, FFN
Output Sequence
Output

Figure F.9: Transformer Model

Appendix G. Implementation

fingerspelling_v2

May 13, 2024

```
[ ]: import pandas as pd
     import pyarrow.parquet as pq
     import math
     import torch
     import torch.nn as nn
     import json
     import torch._dynamo
     from torchdata.dataloader2 import DataLoader2, MultiProcessingReadingService
     import matplotlib.pyplot as plt
     from torchdata.datapipes.iter import (
         FileLister,
         FileOpener,
         TFRecordLoader,
         Mapper,
         Batcher,
         Collator,
         Shuffler,
     )
     import torch.nn.functional as F
     import multiprocessing as mp
     import tensorflow as tf
     import numpy as np
     from tqdm.notebook import (
         tqdm_notebook,
     )  # Assuming you only need one tqdm implementation
     from sklearn.preprocessing import StandardScaler
     from scipy.interpolate import interp1d
     import pytorch_lightning as pl
     from pytorch_lightning import Trainer
     from pytorch_lightning.loggers import TensorBoardLogger
     from pytorch_lightning.callbacks import (
         ModelCheckpoint,
         LearningRateMonitor,
         EarlyStopping,
         RichModelSummary,
     )
     from torchmetrics.text import EditDistance
```

```
# Set the default matmul precision to medium, or high/highest?
torch.backends.cuda.matmul.allow_tf32 = True
torch.backends.cudnn.allow_tf32 = True
torch.set_float32_matmul_precision("medium")
# Read the first CSV file
dataset_train_df = pd.read_csv("train.csv")

# Read the second CSV file
dataset_supplemental_df = pd.read_csv("supplemental_metadata.csv")

# Concatenate the two dataframes
dataset_df = pd.concat([dataset_train_df, dataset_supplemental_df],␣
 ↪ignore_index=True)

# Save the combined CSV file
# dataset_df.to_csv("train_full.csv", index=False)
dataset_df = dataset_df
```

[ ]:
```
# Read the first row of the DataFrame
path, sequence_id, file_id, phrase = dataset_df.iloc[0][
    ["path", "sequence_id", "file_id", "phrase"]
]
print(f"path: {path}, sequence_id: {sequence_id}, file_id: {file_id}, phrase:␣
 ↪{phrase}")

sample_sequence_df = pq.read_table(
    f"{str(path)}",
    filters=[
        [("sequence_id", "=", sequence_id)],
    ],
).to_pandas()
print("Full sequence dataset shape is {}".format(sample_sequence_df.shape))

# Calculate the length of each phrase by characters
df['phrase_length_chars'] = df['phrase'].apply(len)

# Create a histogram
ax = df['phrase_length_chars'].plot.hist(bins=30, color='grey', alpha=0.7)
plt.title('Distribution of Phrase Lengths in Characters')
plt.xlabel('Length of Phrases (characters)')
plt.ylabel('Frequency')
plt.grid(True)

# Calculate and display mean and median
mean_val = df['phrase_length_chars'].mean()
```

```
median_val = df['phrase_length_chars'].median()
plt.axvline(mean_val, color='k', linestyle='dashed', linewidth=1)
plt.axvline(median_val, color='c', linestyle='dashed', linewidth=1)
plt.legend({'Mean':mean_val, 'Median':median_val})

plt.text(mean_val + 10, 1000, f'Mean: {mean_val:.2f}', rotation=0)
plt.text(median_val + 10, 45, f'  Median: {median_val:.2f}', rotation=0)

plt.show()
```

```
[ ]: import os

def process_directory(directory):
  data = []
  for filename in os.listdir(directory):
    if filename.endswith(".parquet"):
      filepath = os.path.join(directory, filename)
      # Read the Parquet file
      df = pd.read_parquet(filepath)
      # Group by sequence_id and count the number of frames
      grouped_data = df.groupby('sequence_id')['frame'].count().reset_index()
      # Append the group data to the main list
      data.append(grouped_data)
  # Combine all group data into a single DataFrame
  result_df = pd.concat(data, ignore_index=True)
  return result_df

directory = "/home/jpinn/asl-fingerspelling-recognition/src/train_landmarks"

# Process the directory and get the DataFrame
result_df = process_directory(directory)
```

```
[ ]: # Create a histogram
plt.figure(figsize=(8, 6))  # Adjust figure size as needed
plt.hist(result_df['frame'], color='grey', bins=100)  # Adjust the number of␣
 ↪bins as needed
plt.xlabel('Number of Frames')
plt.ylabel('Number of Sequences')
plt.title('Distribution of Frames per Sequence')
plt.grid(True)  # Add gridlines for better readability

# Calculate and display mean and median
mean_val = result_df['frame'].mean()
median_val = result_df['frame'].median()
plt.axvline(mean_val, color='k', linestyle='dashed', linewidth=1)
plt.axvline(median_val, color='c', linestyle='dashed', linewidth=1)
plt.legend({'Mean':mean_val, 'Median':median_val})
plt.text(mean_val + 150, 1000, f'Mean: {mean_val:.2f}', rotation=0)
plt.text(median_val + 200, 500, f'  Median: {median_val:.2f}', rotation=0)

plt.tight_layout()
plt.show()
```

[ ]:
```
import pandas as pd
import matplotlib.pyplot as plt

# Function to load and concatenate CSV files
def load_and_concatenate_csv(file_paths):
    dataframes = []
    step_offset = 0  # Initialize step offset
    for file_path in file_paths:
        # Use delimiter='\t' to specify that the values are separated by tabs
        df = pd.read_csv(file_path)
        print(df.columns)  # This will print the column names to check them

        # Ensure 'Step' column exists
        if 'Step' not in df.columns:
            raise ValueError(f"Column 'Step' not found in {file_path}. Columns␣
 ↪found: {df.columns}")

        if step_offset > 0:  # Adjust 'Step' if it's not the first file
            df['Step'] += step_offset

        dataframes.append(df)
        step_offset = df['Step'].iloc[-1]  # Update step offset to the last␣
 ↪step of the current df

    concatenated_df = pd.concat(dataframes)
    return concatenated_df

# File paths for train and validation CSV files
train_csv_files = ['run-transformer_version_218-tag-train_loss_epoch.csv',␣
 ↪'run-transformer_version_220-tag-train_loss_epoch.csv']
val_csv_files = ['run-transformer_version_218-tag-val_loss_epoch.csv',␣
 ↪'run-transformer_version_220-tag-val_loss_epoch.csv']

# Load and concatenate data
train_data = load_and_concatenate_csv(train_csv_files)
val_data = load_and_concatenate_csv(val_csv_files)

# Plotting the training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(train_data['Step'], train_data['Value'], label='Train Loss')
```

```python
plt.plot(val_data['Step'], val_data['Value'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Step')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

[ ]:
```python
import pandas as pd

def parse_line(line):
    # Split each part of the line
    parts = line.split(', ')
    predicted = parts[0].split(': ')[1]
    target = parts[1].split(': ')[1]
    edit_distance = float(parts[2].split(': ')[1])
    return {'Predicted': predicted, 'Target': target, 'Edit Distance':␣
 ↪edit_distance}

def read_data(filepath):
    data_list = []
    with open(filepath, 'r') as file:
        for line in file:
            if line.strip():
                data_list.append(parse_line(line))
    return pd.DataFrame(data_list)

# Replace 'data.txt' with the path to your text file
df = read_data('edit_dists.txt')

total_edit_distance = df['Edit Distance'].mean()
print(total_edit_distance)

mean_target_length = df['Target'].apply(len).mean()
print(mean_target_length)
```

[ ]:
```python
# Read the total amount unique files
unique_paths = dataset_df["path"].unique()

sum = unique_paths.shape[0]

print("Total number of files: {}".format(sum))
```

[ ]:
```python
LIP = [
    61,
    185,
    40,
    39,
    37,
    267,
    269,
    270,
    409,
    291,
    146,
    91,
    181,
    84,
    17,
    314,
    405,
    321,
    375,
    78,
    191,
    80,
    81,
    82,
    13,
    312,
    311,
    310,
    415,
    95,
    88,
    178,
    87,
    14,
    317,
    402,
    318,
    324,
    308,
]

FACE = (
    [f"x_face_{i}" for i in LIP]
    + [f"y_face_{i}" for i in LIP]
    + [f"z_face_{i}" for i in LIP]
)
LHAND = (
    [f"x_left_hand_{i}" for i in range(21)]
    + [f"y_left_hand_{i}" for i in range(21)]
    + [f"z_left_hand_{i}" for i in range(21)]
```

```
)
RHAND = (
    [f"x_right_hand_{i}" for i in range(21)]
    + [f"y_right_hand_{i}" for i in range(21)]
    + [f"z_right_hand_{i}" for i in range(21)]
)
POSE = (
    [f"x_pose_{i}" for i in range(0, 23)]
    + [f"y_pose_{i}" for i in range(0, 23)]
    + [f"z_pose_{i}" for i in range(0, 23)]
)

SEL_COLS = FACE + LHAND + RHAND + POSE
FRAME_LEN = 384  # 384
```

[ ]:
```
# Read the existing data
with open("character_to_prediction_index.json", "r") as f:
    json_chars = json.load(f)

# Define the new entries
new_entries = [
    "<",
    ">",
    "P",
]

# Add the new entries starting from index 59, only if they don't already exist
for i, entry in enumerate(new_entries, start=59):
    if entry not in json_chars:
        json_chars[entry] = i

# Write the updated data back to the file
with open("character_to_prediction_index.json", "w") as f:
    json.dump(json_chars, f, indent=4)

start_token_idx = 59
end_token_idx = 60
pad_token_idx = 61
```

[ ]:
```
from multiprocessing import Manager, Pool


tf.config.set_visible_devices([], "GPU")  # Disable GPU for Tensorflow
# Create a Manager object for the progress_queue

manager = Manager()
progress_queue = manager.Queue()
```

```
def process_file(file_id):
    file_df = dataset_df.loc[dataset_df["file_id"] == file_id]
    path = file_df["path"].values[0]
    parquet_df = pq.read_table(path, columns=["sequence_id"] + SEL_COLS).
 ↪to_pandas()
    features = [FACE, LHAND, RHAND, POSE]
    for feature in features:
        scaler = StandardScaler(with_mean=True, with_std=True)
        parquet_df[feature] = scaler.fit_transform(parquet_df[feature])
    tf_file = f"preprocessed/{file_id}.tfrecord"
    parquet_numpy = parquet_df.to_numpy(copy=False)
    col_to_index = {col: i for i, col in enumerate(parquet_df.columns)}
    LHAND_indices = [col_to_index[col] for col in LHAND]
    RHAND_indices = [col_to_index[col] for col in RHAND]
    buffer_size = 1000  # Adjust as needed
    buffer = []
    with tf.io.TFRecordWriter(tf_file) as file_writer:
        for seq_id, phrase in zip(file_df["sequence_id"], file_df["phrase"]):
            frames = parquet_numpy[parquet_df.index == seq_id]
            progress_queue.put(
                f"Process: {mp.current_process().name}, File: {file_id},␣
 ↪Sequence: {seq_id}"
            )
            if frames.shape[0] > FRAME_LEN:
                itp = interp1d(
                    np.linspace(0, 1, len(frames)),
                    frames,
                    axis=0,
                    kind="linear",
                    fill_value="extrapolate",
                )
                # Generate the new index array and apply interpolation
                frames = itp(np.linspace(0, 1, FRAME_LEN))
            # Calculate the number of NaN values in each hand landmark
            r_nonan = np.sum(np.sum(np.isnan(frames[:, RHAND_indices]), axis=1)␣
 ↪== 0)
            l_nonan = np.sum(np.sum(np.isnan(frames[:, LHAND_indices]), axis=1)␣
 ↪== 0)
            no_nan = max(r_nonan, l_nonan)
            frames = np.nan_to_num(frames, nan=0)
            num_hand_frames = np.sum(
                np.any(frames[:, LHAND_indices + RHAND_indices] != 0, axis=1)
            )
            if frames.shape[0] < 50 and num_hand_frames < 3:
                phrase = "2 a-e -aroe"
            if 2 * len(phrase) < no_nan:
                features = {
                    COL: tf.train.Feature(
                        float_list=tf.train.FloatList(
                            value=frames[:, col_to_index[COL]]
                        )
                    )
                    for COL in SEL_COLS
                }
                features["phrase"] = tf.train.Feature(
                    bytes_list=tf.train.BytesList(value=[bytes(phrase,␣
 ↪"utf-8")])
                )
                example = tf.train.Example(features=tf.train.
 ↪Features(feature=features))
                record_bytes = example.SerializeToString()
                buffer.append(record_bytes)
                if len(buffer) == buffer_size:
                    for record in buffer:
                        file_writer.write(record)
                        buffer = []
        if buffer:
            for record in buffer:
                file_writer.write(record)
    # gc.collect()


cpu_count = int(mp.cpu_count() / 2)
cpu_count = 6  # 8"""  """
with Pool(cpu_count) as pool:
    progress_bars = [
        tqdm_notebook(desc=f"Process {i + 1}", unit="seq") for i in␣
 ↪range(cpu_count)
    ]
    for result in pool.imap(
        process_file,
        dataset_df["file_id"].unique(),
    ):
        progress_updates = []
        while not progress_queue.empty():
            progress_updates.append(progress_queue.get())
        for update, bar in zip(progress_updates, progress_bars):
            bar.set_description(update)
            bar.update()
print("All parquets processed to TFRecords")
```

[ ]:
```python
import os
import random

with open("character_to_prediction_index.json", "r") as file:
    vocab = json.load(file)


def tokenize_string(text):
    # Tokenize the string using the provided vocabulary
    token_ids = [vocab[char] for char in text if char in vocab]
    return token_ids


def detokenize_batch(batch, INFERENCE=False):
    # Create a reverse vocabulary
    reverse_vocab = {v: k for k, v in vocab.items()}

    # Convert the token IDs back to characters for each sequence in the batch
    texts = []
    for seq in batch:
        text = []
        for id in seq:
            char = reverse_vocab[id.item()]
            if INFERENCE and char == ">":
                break  # Stop adding characters when '>' is found during␣
 ↪inference
            if char == "<":
                continue
            text.append(char)
        texts.append("".join(text))

    return texts


# Encodes phrase into a tensor of tokens
def tokenize_phrase(example):
    phrase = example["phrase"][0].decode(
        "utf-8"
    )  # Decode the byte string into a regular string
    phrase = "<" + phrase + ">"
    token_ids = tokenize_string(phrase)
    example["phrase"] = torch.tensor(
        token_ids
    )  # Replace the byte string with a list of integers
    return example
```

```
def collate_fn(batch):
    # Separate phrases and sequence lengths
    phrases = [seq.pop("phrase") for seq in batch]
    landmarks = [seq for seq in batch]

    sequence_lengths = [len(next(iter(landmark.values()))) for landmark in␣
 ↪landmarks]
    phrase_lengths = [len(phrase) for phrase in phrases]

    # Pad sequences and phrases
    padded_batch = [
        torch.stack(
            [
                F.pad(
                    input=tensor,
                    pad=(0, FRAME_LEN - tensor.shape[0]),
                    mode="constant",
                    value=0,
                )
                for tensor in seq.values()
            ],
            dim=-1,
        )
        for seq in batch
    ]

    stacked_landmarks = torch.stack(padded_batch, dim=0)

    padded_phrases = [
        F.pad(
            input=phrase,
            pad=(0, 64 - len(phrase)),
            mode="constant",
            value=61,
        )
        for phrase in phrases
    ]

    stacked_phrases = torch.stack(padded_phrases, dim=0)

    return (
        stacked_landmarks,
        stacked_phrases,
        torch.tensor(sequence_lengths),
        torch.tensor(phrase_lengths),
    )
```

```
# Compute the split index
tf_records = dataset_df.file_id.map(
    lambda x: f"/home/jpinn/asl-fingerspelling-recognition/src/preprocessed/{x}.
 ↪tfrecord"
).unique()

# Sample 20% of the TFRecords
# sample_size = int(0.2 * len(tf_records))  # Calculate 20% of the total records
# tf_records = random.sample(list(tf_records), sample_size)

split_index = int(0.8 * len(tf_records))
tf_records_len = len(tf_records)

print(f"Split index: {split_index}" f"\nTotal number of TFRecords:␣
 ↪{tf_records_len}")


def build_pipe(batch_size, drop_last, start, end, shuffle=True):
    datapipe = FileLister(tf_records[start:end])

    if shuffle:
        datapipe = Shuffler(
            datapipe, buffer_size=len(tf_records[start:end])
        )  # Shuffle the dataset

    datapipe = FileOpener(datapipe, mode="b")
    datapipe = TFRecordLoader(datapipe)
    datapipe = Mapper(datapipe, tokenize_phrase)
    datapipe = Batcher(datapipe, batch_size=batch_size, drop_last=drop_last)
    datapipe = Collator(datapipe, collate_fn=collate_fn)
    return datapipe
```

```
[ ]: class LightningDataModule(pl.LightningDataModule):
         def __init__(self, batch_size=64, shuffle=True):
             super().__init__()
             self.batch_size = batch_size
             self.shuffle = shuffle

         def train_dataloader(self):
             train_datapipe = build_pipe(
                 batch_size=self.batch_size,
                 drop_last=True,
                 start=0,
                 end=split_index,
                 shuffle=self.shuffle,
             )
```

```python
        return DataLoader2(
            datapipe=train_datapipe,
            reading_service=MultiProcessingReadingService(num_workers=6),
        )

    def val_dataloader(self):
        val_datapipe = build_pipe(
            batch_size=self.batch_size,
            drop_last=True,
            start=split_index+1,
            end=tf_records_len-5,
            shuffle=False,
        )
        return DataLoader2(
            datapipe=val_datapipe,
            reading_service=MultiProcessingReadingService(num_workers=6),
        )

    def predict_dataloader(self):
        predict_datapipe = build_pipe(
            batch_size=self.batch_size,
            drop_last=True,
            start=tf_records_len-4,
            end=tf_records_len,
            shuffle=False,
        )
        return DataLoader2(
            datapipe=predict_datapipe,
            reading_service=MultiProcessingReadingService(num_workers=2),
        )
```

[ ]:
```python
class TokenEmbedding(nn.Module):
    def __init__(self, num_vocab=62, maxlen=64, d_model=312):
        super(TokenEmbedding, self).__init__()
        self.emb = nn.Embedding(num_vocab, d_model, padding_idx=61)
        self.pos_emb = nn.Embedding(maxlen, d_model)

    def forward(self, x):
        maxlen = x.shape[-1]

        x = self.emb(x)

        # Generate positions
        positions = torch.arange(start=0, end=maxlen, dtype=torch.long,␣
 ↪device=x.device)
        pos_emb = self.pos_emb(positions)
        pos_emb = pos_emb.unsqueeze(0).expand(x.size(0), -1, -1)

        return x + pos_emb


class LandmarkEmbedding(nn.Module):
    def __init__(self, d_model=312, maxlen=FRAME_LEN, device="cuda", dropout=0.
 ↪1):
        super(LandmarkEmbedding, self).__init__()

        self.pos_emb = nn.Embedding(maxlen, d_model)

        # Define Conv1d layers
        self.conv1 = nn.Conv1d(
            in_channels=d_model, out_channels=d_model, kernel_size=11, padding=5
        )  # padding to maintain sequence length
        self.conv2 = nn.Conv1d(
            in_channels=d_model, out_channels=d_model, kernel_size=11, padding=5
        )
        self.conv3 = nn.Conv1d(
            in_channels=d_model, out_channels=d_model, kernel_size=11, padding=5
        )

        # Batch normalization layers
        self.bn1 = nn.BatchNorm1d(d_model)
        self.bn2 = nn.BatchNorm1d(d_model)
        self.bn3 = nn.BatchNorm1d(d_model)

        self.dropout = nn.Dropout(dropout)

        # Move to the specified device
        self.to(device)

    def forward(self, x):

        # Permute to fit Conv1d input requirements: [batch_size, channels,␣
 ↪seq_len]
        x = x.permute(0, 2, 1)

        # Apply Conv1d layers with ReLU activations and batch normalization
        x = self.dropout(F.silu(self.bn1(self.conv1(x))))
        x = self.dropout(F.silu(self.bn2(self.conv2(x))))
        x = self.dropout(F.silu(self.bn3(self.conv3(x))))

        # Permute back to [batch_size, seq_len, features]
        x = x.permute(0, 2, 1)
        # Generate positions
        positions = torch.arange(
            start=0, end=x.shape[1], dtype=torch.long, device=x.device
        )
        pos_emb = self.pos_emb(positions)

        pos_emb = pos_emb.unsqueeze(0).expand(x.size(0), -1, -1)

        return x + pos_emb
```

[ ]:
```
import csv


class LightningTransformer(pl.LightningModule):
    def __init__(self, config):
        super().__init__()
        self.save_hyperparameters(config)
        self.config = config
        self.batch_size = config["batch_size"]
        self.learning_rate = config["learning_rate"]
        self.enc_emb = LandmarkEmbedding(config["d_model"],␣
 ↪config["src_maxlen"])
        self.dec_emb = TokenEmbedding(
            config["num_classes"], config["tgt_maxlen"], config["d_model"]
        )
        self.transformer = nn.Transformer(
            d_model=config["d_model"],
            nhead=config["nhead"],
            dropout=config["dropout"],
            num_encoder_layers=config["num_encoder_layers"],
            num_decoder_layers=config["num_decoder_layers"],
            batch_first=True,
        )

        self.linear = nn.Linear(self.transformer.d_model, 62)

        self.metric = EditDistance()
        self.predictions_log = []
        self.loss = nn.CrossEntropyLoss(ignore_index=61)

    def create_mask(self, batch_size, max_length, real_length):
        """Create a boolean mask for sequences based on lengths."""
        key_padding_mask = torch.ones(
            (batch_size, max_length), device=self.device, dtype=torch.bfloat16
        )
        for i, length in enumerate(real_length):
            key_padding_mask[i, 0:length] = False
```

```python
        return key_padding_mask

    def forward(self, src, tgt, src_key_padding_mask, tgt_key_padding_mask):
        src = self.enc_emb(src)
        tgt = self.dec_emb(tgt)
        # Encodes source and decodes target sequences
        tgt_mask = self.transformer.generate_square_subsequent_mask(
            63, dtype=torch.bfloat16, device=self.device
        )
        output = self.transformer(
            src,
            tgt,
            tgt_mask=tgt_mask,
            src_key_padding_mask=src_key_padding_mask,
            tgt_key_padding_mask=tgt_key_padding_mask,
            tgt_is_causal=True,
        )
        return self.linear(output)

    def training_step(self, batch, batch_idx):
        source, target, src_lengths, tgt_lengths = batch

        tgt_input = target[:, :-1]  # Shifted right for input
        tgt_output = target[:, 1:]  # Real target without the first token

        src_key_padding_mask = self.create_mask(
            source.size(0), source.size(1), src_lengths
        )

        tgt_key_padding_mask = self.create_mask(
            target.size(0),
            target.size(1),
            tgt_lengths,
        )

        # Get model output
        output = self(
            source, tgt_input, src_key_padding_mask, tgt_key_padding_mask[:, :
↪-1]
        )

        # Compute loss; CrossEntropyLoss expects outputs of size (N, C, L) and␣
↪target of size (N, L)
        loss = self.loss(output.transpose(1, 2), tgt_output)

        self.log("train_loss", loss, on_step=True, on_epoch=True, prog_bar=True)
```

Appendix H. Testing

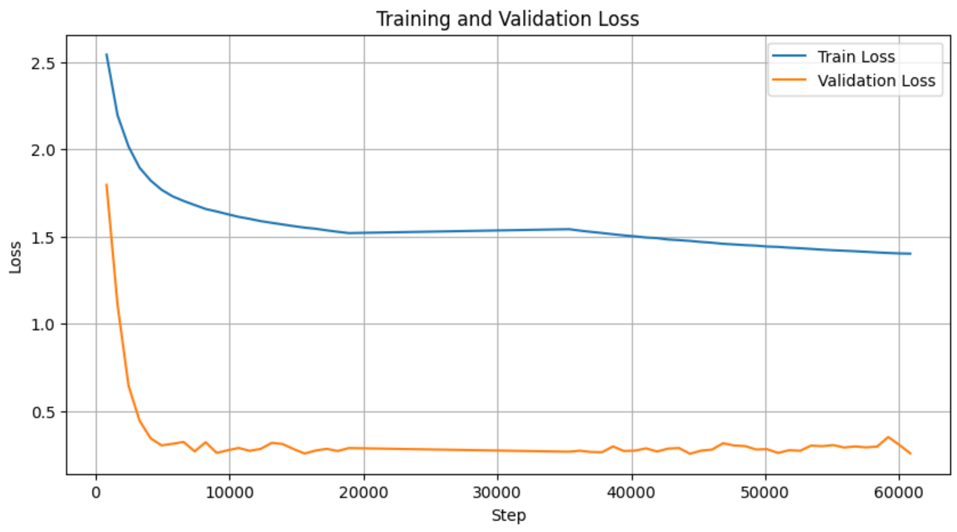

Figure H.10: Train Loss vs Validation Loss 16 Hours

Appendix I. User Guide

Not applicable.

Appendix J. Project Management

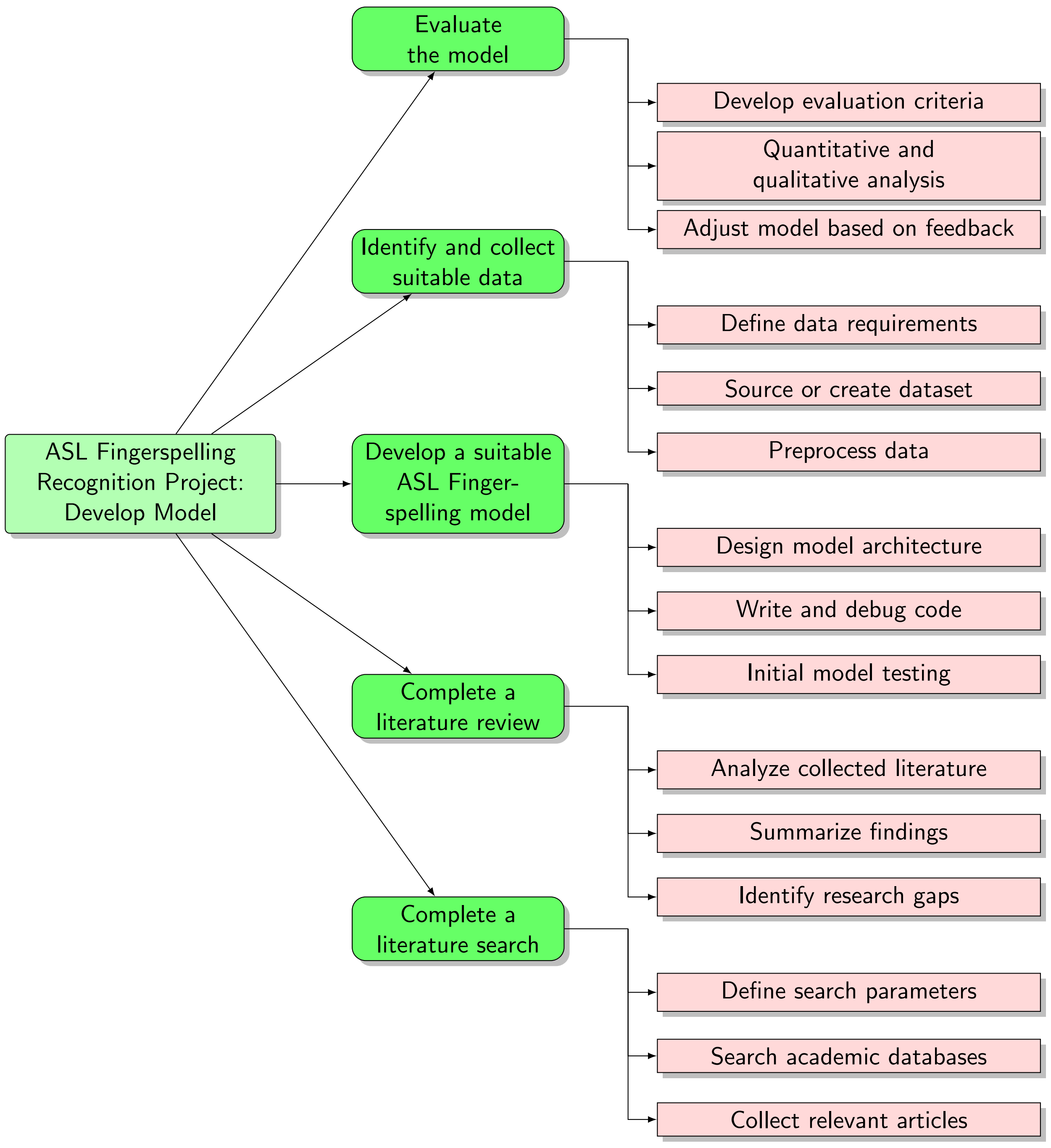
Evaluate the model
Develop evaluation criteria
Quantitative and qualitative analysis
Adjust model based on feedback
Identify and collect suitable data
Define data requirements
Source or create dataset
Preprocess data
ASL Fingerspelling Recognition Project: Develop Model
Develop a suitable ASL Finger-spelling model
Design model architecture
Write and debug code
Initial model testing
Complete a literature review
Analyze collected literature
Summarize findings
Identify research gaps
Complete a literature search
Define search parameters
Search academic databases
Collect relevant articles

Figure J.11: Work Breakdown Structure: Develop Model

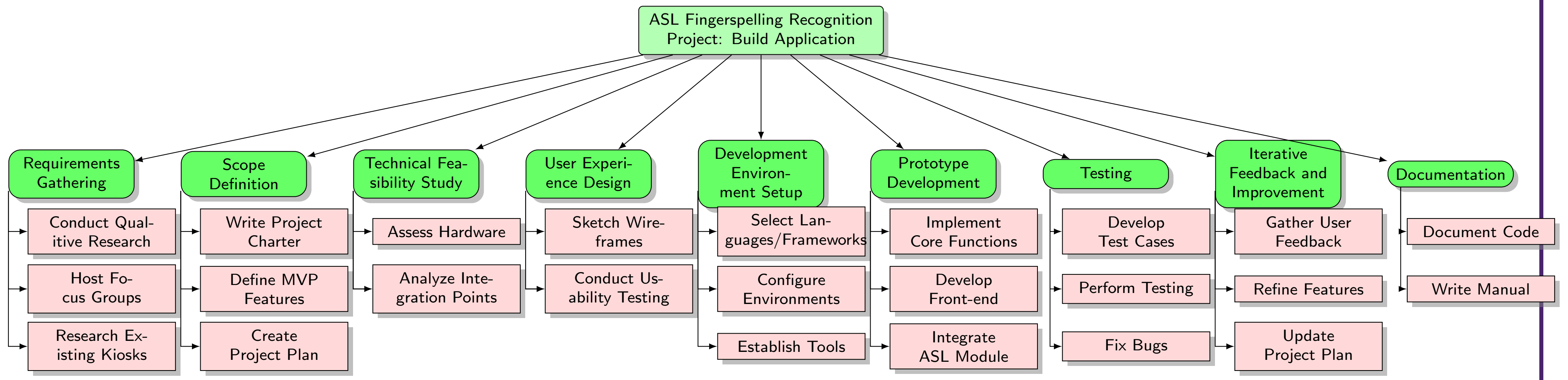

Figure J.12: Work Breakdown Structure: Build Application

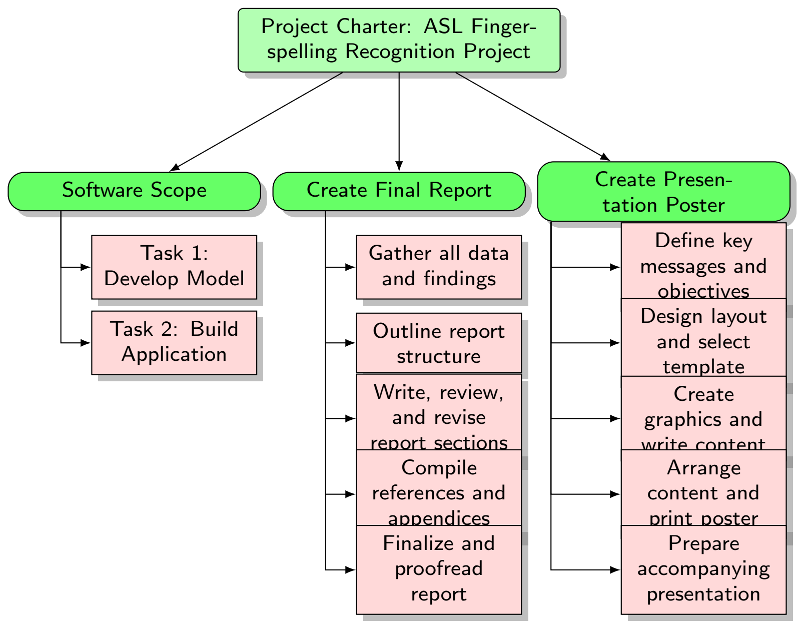
Project Charter: ASL Finger-spelling Recognition Project
Software Scope
Task 1: Develop Model
Task 2: Build Application
Create Final Report
Gather all data and findings
Outline report structure
Write, review, and revise report sections
Compile references and appendices
Finalize and proofread report
Create Presentation Poster
Define key messages and objectives
Design layout and select template
Create graphics and write content
Arrange content and print poster
Prepare accompanying presentation

Figure J.13: Work Breakdown Structure

Gantt chart placeholder
(Refer to the accompanying zipped file for the full chart)

Figure J.14: Gantt Chart

In this report's accompanying zipped file, a detailed Gantt chart is included as a separate document. Due to its extensive size and complexity, it is provided as an individual file to facilitate detailed review and ensure clarity. Please refer to the zipped file named gantt-project.png for the complete Gantt chart, which offers an in-depth view of the project timeline and milestones.

Risk	**Impact**	**Probability**	**Status**	**Mitigation Strategy**
Inaccurate ASL Recognition	High	Medium	Open	Enhance data collection and improve algorithm accuracy.
Data Privacy Concerns	High	High	Open	Implement GDPR compliant data handling processes.
Loss of Data	High	Low	Open	Implement GoogleDrive and GitHub repository.
Loss of Project Supervisor	High	Low	Open	Maintain regular communication with supervisor.
Project Delays	Medium	High	Open	Develop a schedule with buffers and regularly update it.
User Adoption Challenges	Medium	High	Open	Engage with users early and incorporate feedback.
Technology Integration Issues	Medium	Medium	Open	Conduct compatibility testing.

Figure J.15: Risk register

Appendix K. Meetings With Supervisor

Date	**Time**	**Location**	**Purpose**	**Description and Actions**
15 November 2023	13:00-14:00	CCCU - Lg33	Discuss project and literature research	Reviewed current status, discussed challenges, and agreed on next steps including further research on ASL recognition and user-centric approach to requirements. Contact charities
29 November 2023	13:00-14:00	CCCU - Lg33	Discuss third party inclusion, user centric requirement	Update on contacted charities and double diamond requirements
13 December 2023	13:00-13:30	Online	Catchup and talk poster presentation	Spoke about markschemes, poster style, sharing work

Appendix L. Agile Development: Timebox 1

Not applicable.

Appendix M. Agile Development: Timebox 2

Not applicable.

Appendix N. Agile Development: Timebox 3

Not applicable.