From Sign Language to Speech using Artificial Intelligence

Andreas Holmer Bigom, Michael Alexander Harborg,
Niels Raunkjær Holm

Bachelor Project
June 19, 2023

Abstract

The ability to communicate is a defining characteristic of what constitutes being human. It allows us to express ourselves, cooperate, and foster new relationships. Being deaf or mute, however, presents unique challenges in the context of interpersonal communication. Sign language is the primary means of communication for over 72 million people worldwide [7]. Currently, the language barrier between signers and non-signers is bridged through scarce and costly human interpreters. Enabling communication between signers and non-signers at an on-demand basis could thus impact millions of people.

A domain within machine learning which addresses this is Sign Language Translation (SLT). However, SLT is still in its early phases which can in part be attributed to a lack of large, high quality datasets as well as having to bridge the modalities of video and text. In spite of these challenges, recent advances in the field have yielded substantial increases in model performance. Most notable is the current state-of-the-art (SOTA) SLT method [5], who address the issue of data scarcity through domain adaptation and progressive pre-training. In spite of the impressive results, a crucial bottleneck of this model is that it requires gloss annotations[1] for supervision.

This project revolves around assessing the feasibility of developing an application to allow for on-demand communication between signers and non-signers. To this end, we reproduce the current SOTA SLT method [5] by manually implementing it from scratch. Furthermore, we design an application pipeline whose architecture is centered around reducing latency during conversations. To accommodate language differences between users and improve the user experience, various external APIs are incorporated. Additionally, to assess the prospect of leveraging large SLT datasets, an ablation study is conducted where gloss annotations are approximated.

Upon developing the model as well as application, several tests were conducted to assess the individual performance of its components. This includes assessing the success of reproducing the model along with the effect on performance of approximating glosses. Additionally, the model's ability to generalize is assessed by evaluating model performance on new unseen data. Following this, the latency surrounding translation is tested on consumer-grade hardware. Furthermore, the effect on model performance when downsampling input videos along the temporal axis is assessed in conjunction with its effect on application latency.

Based on the analyses described above, it can be concluded that we were successfully able to reproduce the SOTA SLT model [5] within a reasonable margin of error. Furthermore, when approximating glosses we outperform existing approaches which do not rely on these annotations. Moreover, when assessing application latency on consumer-grade hardware, our pipeline approaches real-time SLT as defined in the context of this report. Future research should explore means of circumventing the need for expertly-annotated glosses. Additionally, considering the execution time of different components of the application pipeline, further work should experiment with the potential benefits from hosting the SLT model in the cloud.

[1] Glosses are word-for-word transcriptions of each sign in an input video which capture its semantic meaning.

Acknowledgements

Advisors

Morten Mørup, professor at DTU
Carsten Witt, professor at DTU

Contents

Abbreviation	Description
SLT	Sign language translation.
NMT	Neural machine translation.
ASR	Automatic Speech Recognition.
DGS	German sign language.
ASL	American sign language.
BSL	British sign language.
TSL	Turkish sign language.
CSL	Chinese sign language.
S3D	Separable 3D ConvNet
I3D	Inflated 3D ConvNet
Sign2Gloss	The process of translating a sign language video into a correspond sequence of glosses.
Gloss2Text	The process of translating a gloss sequence into natural language.
Sign2Text	Translating a sign language video directly into natural language.
Text2Speech	The process of synthesizing text to speech.
Speech2Text	The process of translating speech audio into text.
CTC	Connectionist temporal classification.
WER	Word Error Rate
BLEU	Bilingual Evaluation Understudy.
ROUGE	Recall-Oriented Understudy for Gisting Evaluation.
REST	Representational State Transfer
TCP	Transmission Control Protocol
FLOPS	Floating point operations

Table 1: Overview of the different abbreviations used throughout the project.

1 Introduction

An estimated 430 million people worldwide suffer from disabling hearing loss [21]. Despite workarounds to mitigate the effects of hearing impairment, over 72 million people globally use sign language as their primary form of communication [7]. The ability to communicate is paramount for development, both on an individual and a societal level, hence why sign language is no less complex than spoken language. Currently, the communication barrier between signers and non-signers is alleviated through human translators. However, this process is costly and the availability of translators is scarce.

Recent advances in Neural Machine Translation (NMT) have enabled the development of tools which help reduce language barriers globally. Most prominent is the transformer architecture introduced in Attention is All You Need [28], which, since its release, has revolutionized the field of machine learning as a whole. Coupled with advances in computer vision for video processing and the release of large sign language datasets, the prospect of bridging the gap between signers and non-signers through these technologies becomes evermore possible.

In spite of sharing similarities with the task of translating across spoken languages, Sign Language Translation (SLT) introduces several additional challenges. For one, the modalities of the input data and the output sequences are different, given the visual nature of sign language. Sign languages consist of complex syntaxes of visual gestures. Hand movements, body language, facial expressions, and the context in which signs occur all play a part in determining the precise meaning of the sentence. There are also significant differences in the grammatical structure of sign languages as opposed to spoken and written languages which complicates sentence alignment. For this reason, SLT researchers often utilize glosses[2] along with latent visual representations, as an intermediate step between the video input and the final translated sentence. This component, particularly, has paved the way for the current state-of-the-art methods. The current state-of-the-art (SOTA) SLT model [5], doubles the performance when compared to the current best method that omits glosses, TSPNet [16].

However, the reliance on gloss annotations also has its drawbacks, as only few datasets contain them. This is due to the annotation procedure requiring expert level domain knowledge and even then being heavily time-consuming [6]. Several authors, whose methods are reliant on gloss annotations, have seen substantial improvements in performance by employing various strategies to combat the lack of training data. BN-TIN-Transf. [31] alleviates data scarcity by supplementing training with synthetically generated data. The SOTA model [5] achieves state of the art performance by utilizing transfer learning from generic action sequences, hypothesizing that isolated signs are a special case of general actions. In spite of these advances, both of these papers addressed SLT in very narrow domains.

To alleviate the amount of resources required to obtain gloss annotations for large SLT datasets, several authors focus on developing methods for approximating

[2]Glosses are word-level transcriptions of signs which are closely associated with the signs meaning.

them. The authors behind BOBSL [1], the largest publicized SLT dataset, train several models to this end. They propose detecting signs based on mouthing as well as comparing the similarity between frames with queries from a database of isolated signs. Additionally, they localize signs in input videos using the attention mechanism of a transformer trained on SLT data. Another approach focuses on approximating gloss annotations through natural language translations of sign videos and expert knowledge regarding the linguistic differences between signed and spoken language [20].

This project aims to assess the current usefulness of SLT systems in a real world setting. To this end, we set out to reproduce the current SOTA SLT model [5] by manually implementing the proposed method *from scratch*. This method was chosen due to its high performance on typical natural language processing metrics, along with considerations of inference time in the context of deploying it in a user application. Additionally, we create such an application, which allows for two-way communication between signers and non-signers. During the application design and development, emphasis was placed on minimizing execution time and latency for all components required to process user input and use it for model inference.

Additionally, to assess the feasibility of general-domain SLT, an ablation study testing the approach for circumventing costly expertly annotated glosses in [20] is performed. Lastly, we evaluate the model's ability to generalize by testing its performance on unseen samples drawn from the DGS Corpus [10].

The repositories for the code bases developed throughout this project, along with the model checkpoints and application installation can be found through the links presented in table 20 in the appendix. Moreover, a table over the individual contributions for the project can be found in section 8.1 in the appendix.

1.1 Research Questions

In order to address the aforementioned problem statements, the following set of research questions were formulated, to concretize the objectives of the project.

- How well can we reproduce the state-of-the-art Sign Language Translation model [5] and is it possible to achieve similar performance, without the dependency on gloss annotated data?

In order to answer this research question, we aim to reproduce the state-of-the-art model architecture from scratch, solely through the description of the methods and implementation details formulated in their paper [5]. In relation to this, we build a GitHub organization, containing a set of repositories for our individual code bases.

Furthermore, we extend these methods to work with datasets not initially containing annotated gloss sequences, by generating them as approximations given a spoken language annotation.

- How do we design the application pipeline to be used in a real-time setting?

In this context, we direct our focus towards the pipeline going from user input to model inference and back to the user with the results. This pipeline constitutes the

infrastructure for interfacing with the ML models, and will thus be the determining factor in terms of execution speed and application latency, along with the models themselves. When choosing the technologies to implement the application, we prioritize computational efficiency and low communication overhead, since we aim to construct a piece of software that can be run on consumer level hardware, e.g. a laptop.

Furthermore, we conduct quantitative tests in order to assess the impact of decreasing the temporal resolution[3] as part of the application pipeline.

Finally, we evaluate the impact of using hardware acceleration for the ML models.

- How well does the application work in a real world setting?

To access how our application performs in a real world setting, we evaluate whether the proposed model is able to generalize to the distribution of never before seen general-domain data.

- How can we accommodate language differences across users and how do these implementations affect application latency?

To accommodate language differences across users, we implement a pre-existing natural language translation model, text-to-speech model and speech-to-text model in the backend model server of our application. To access how these implementations affect the latency of the application, we evaluate the relative contribution to latency of each component of the backend model server.

1.2 Report Outline

First, preliminary information about the SLT task is provided. Following this, various available datasets are presented along with their properties. This highlights the general availability of resources and serves as prerequisite knowledge for the remarks on achieving general-domain SLT. Moreover, ethical considerations surrounding SLT datasets are made.

Hereafter, the underlying methodology surrounding the SLT model, evaluation metrics and application dependencies are presented. The process of translating a sign language video into glosses and subsequently into natural language is explained in detail. Additionally, details surrounding implementation and training dependencies at various stages of the model pipeline are presented. Lastly, the methodical framework surrounding an ablation study to effect of omitting gloss annotations during SLT is presented.

Following this, the application and design choices during its development are presented. The architecture of the application is outlined along with a description of its components. Furthermore, the process of recording, streaming and processing video data within the application is explained. Lastly, as a means to reduce execution time of inference, temporal downsampling of input videos are introduced.

Hereafter, the results primary results of the project are presented and described. This includes model performance attained when reproducing the SOTA model [5]

[3] Lowering the frames per second by means of downsampling.

along with the performance when omitting gloss annotations. The performance of the system in a general-domain setting is also presented. Additionally, execution time benchmarks for inputs of varying sizes and user-settings are shown.

This is followed by further discussion of the obtained results in conjunction with the research questions. Here, remarks on reproducibility as well as the ability to generalize of the method are made. Following this, an assessment of deploying an SLT application in the real-world setting is made. In this context, the limitations of current SLT systems and steps to undertake which could lead to high performing, general-domain SLT models are discussed.

Lastly, the general findings of the project are summarized.

2 Data

This section introduces the domain of SLT and general format of datasets. An overview of the general data availability is given. Additionally, the datasets used within this project are described and their properties are summarized. Lastly, ethical considerations surrounding SLT datasets generally are presented.

2.1 Presentation and Summary Statistics

Sign languages consist of complex visual syntaxes and are completely distinct from their spoken language counterparts. They contain all of the fundamental properties of language e.g. word formatting, ordering and pronunciations. Fingerspelling is also a common component of sign languages often used to indicate proper names as well as the equivalent word for something in spoken language. Using neural methods to translate sign language into natural language is no easy feat. For one, the available datasets are orders of magnitude smaller compared to natural language translations. Furthermore, the input modality, i.e. video, increases the computational overhead required to train SLT models significantly. Adding to this, the mapping between signed sequences and words is generally a complex many-to-many mapping. Hence, determining the precise meaning of a signed sequence is heavily context dependent.

At a minimum, sign language translation datasets contain RGB videos of signers along with natural language translations. Some may also include keypoint video estimates of hand and face movements as well as gloss annotations. Glosses are approximate word-for-word transcriptions of each sign in a given video, where the semantic meaning of the given sign is captured.

2.1.1 PHOENIX-2014T

The main dataset used within this project is PHOENIX-2014T [2]. This is due to its widespread usage amongst SLT practitioners, allowing for meaningful model comparisons. The dataset contains signed sequences in the form of RGB clips, along with their corresponding gloss annotations and natural language translations. The videos have been collected from German weather forecasts and thus signed sequences and natural translations are in German sign language (DGS) and German, respectively. The general data format within SLT is illustrated in figure 1 using a sample from this dataset.

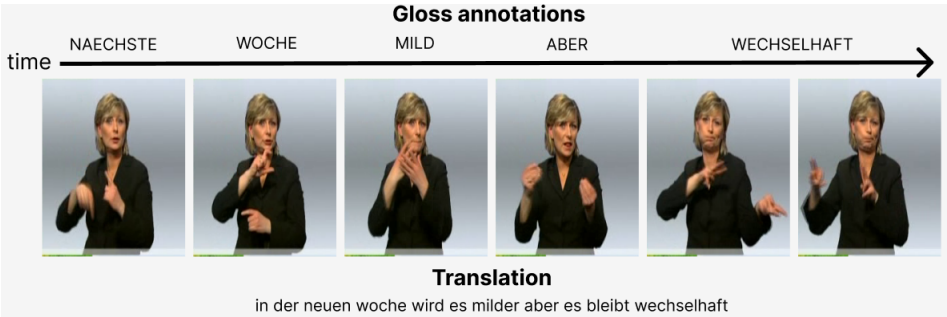


Figure 1: Sample from the PHOENIX-2014T dataset showcasing the corresponding gloss annotations and the German translation.

2.1.2 WLASL

Data scarcity is a general issue in the context of SLT. For this reason, the authors behind the state-of-the-art (SOTA) SLT method [5] pre-train the visual component of their model on the Word Level American Sign Language (WLASL) dataset [15]. It consists of RGB video clips of single signs along with their corresponding gloss annotation. Furthermore, this dataset has a large set of diverse signers which could help assert invariance with respect to the features of the person signing. Since part of this project focuses on reproducing the model presented in SOTA, WLASL is used for general domain pre-training prior to training on PHOENIX-2014T.

2.1.3 Kinetics-400

The Kinetics-400 dataset [12] is an action recognition dataset containing 306245 labeled videos of humans performing generic actions. The videos originate from YouTube and annotations were collected through workers from Amazon Mechanical Turk. Motivated by the data scarcity in SLT, SOTA argue that recognizing signs can be considered a special case of action recognition. For this reason, they initialize the visual component of their SLT model on Kinetics-400.

2.1.4 Properties of the Considered Datasets

To add further context to the datasets used within this project, key properties of them are presented in table 2 below. The size of splits for WLASL and PHOENIX-2014T are provided. Additionally, the number of words and glosses that are out of vocabulary i.e. present in the validation or test set but not in the training set are reported.

Dataset	Size			OOV glosses		OOV words	
	train	validation	test	validation	test	validation	test
WLASL	14k	4k	3k	0	0	N/A	N/A
PHOENIX-2014T	7096	519	642	12	18	57	58

Table 2: Properties of the datasets used within this project. Their sizes and the number of words and glosses that are out-of-vocabulary are reported.

As evident from table 2, the considered datasets have a negligible amount of words and glosses that are out-of-vocabulary. Additionally, the distribution of words and glosses are reported in figure 2 below.

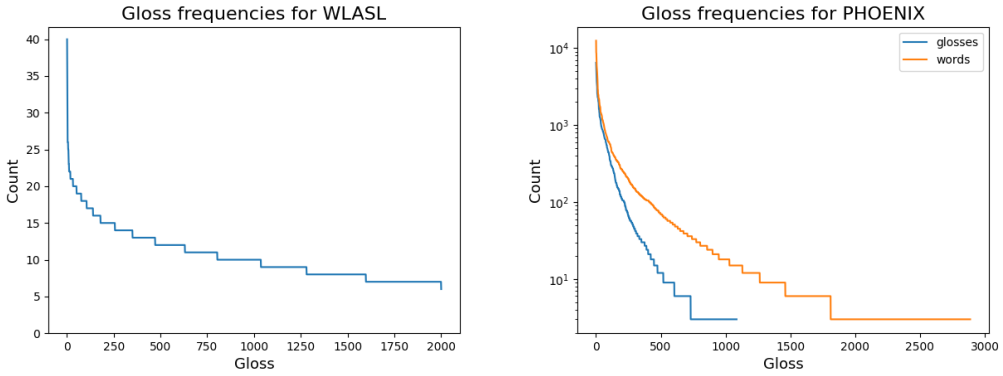


Figure 2: Gloss and word frequencies for PHOENIX-2014T and WLASL respectively.

Considering the word and gloss distributions illustrated in table 2, it is clear that WLASL has a relatively uniform distribution. The distribution of glosses and words for PHOENIX-2014T is more right-skewed. This is to be expected considering that PHOENIX-2014T contains sentences rather than words and thus reflects the general word-frequencies present in language. Supplementary information regarding PHOENIX-2014T and WLASL can be seen in table 3.

2.1.5 General Data Availability

To provide context about the general data availability in SLT, we present the most prominent resources currently available in table 3 below.

Name	Language[4]	Vocabulary size		Hours	Domain-type	N.O signers	Source
		Words	Glosses				
WLASL	ASL	N/A	2000	14	General	119	Web
PHOENIX-2014T	DGS	2887	1085	11	Specific	9	TV
DGS Corpus	DGS	42k	38k	∼50	General	330	Lab
How2Sign	ASL	16k	N/A	∼80	General	11	Lab
OpenASL	ASL	33k	N/A	288	General	∼220	Web
MSASL	ASL	N/A	1000	25	General	222	Web
CSL-daily	CSL	2343	2000	23	General	10	Lab
AUTSL	TSL	N/A	226	∼25	General	43	Web
BOBSL	BSL	78k	N/A*	1467	General	39	TV

Table 3: Overview of various SLT datasets and their properties. Note that N/A* in the context of BOBSL refers to the use of statistical means to annotate glosses. As such the gloss vocab size varies with the confidence threshold for annotations.

As evident from table 3, there are several large SLT datasets which span a general domain. However, the majority of large SLT datasets do not contain glosses, a prerequisite for training our model. This should come as no surprise considering

[4]American Sign Language (ASL), German Sign Language (DGS), Chinese Sign Language (CSL), Turkish Sign Language (TSL) and British Sign Language (BSL).

the resources required to collect gloss annotations. For reference, the authors of How2Sign [6] report that glosses were annotated at a rate of 90 seconds of video per hour.[5] The BOBSL datasets is an exception to this and resorts to statistical methods as a means to approximate them [1]. However, BOBSL is under particularly restrictive licensing and it was therefore deemed unfeasible to obtain clearance for its use in the context of this project.

2.2 Ethical Considerations

Several ethical concerns are raised when working with sign language datasets. Here, the concerns deemed most notable surrounding the development of SLT models and the utilized data are presented.

2.2.1 Privacy

The need for RGB videos with hand movements and visual cues can, without appropriate processing, lead to increased surveillance of signers. The lack of anonymity is also amplified by the fact that sign language communities are generally small, particularly for low resource sign languages. Additionally, information regarding disabilities of individuals are sensitive features which will be conveyed, unless accounted for.

2.2.2 Bias

Another potential byproduct of scarcity regarding SLT data is a lack of ethnic coverage, gender diversity and differences in body type of signers. This could lead to discrimination with respect to performance for different users. As highlighted in table 3 several of the SLT datasets, including PHOENIX-2014T, contain few signers. These concerns could be addressed by progressive pre-training using datasets with many signers present such as WLASL or MSASL [11].

[5] Gloss annotations for How2Sign are still underway and in spite of having reached out to the authors, we were not able to obtain a subset of these annotations.

3 Methods

This sections covers the underlying theory of the considered models along with implementation details. The theoretical framework of this project closely follows the state-of-the-art (SOTA) SLT model [5]. As the framework consists of three main stages, namely Sign2Gloss translation, Gloss2Text translation and lastly Sign2Text translation, these individual steps will be presented sequentially.

Additionally, the various evaluation metrics used to assess the performance of the models are presented. Lastly, the methodology surrounding the ablation study assessing the prospect of omitting expertly-annotated glosses is presented.

3.1 Proposed Model Architecture

This section introduces the architecture of the proposed SLT model introduced in [5] on a high level. Since the model is built to solve a multi-modal task, its architecture is divided into three components, illustrated in figure 3. The visual component of the model is responsible for creating a meaningful embedding of given sign videos. The modality mapper converts the features from the visual domain into language features. The language component is responsible for translating the language features into natural language text.

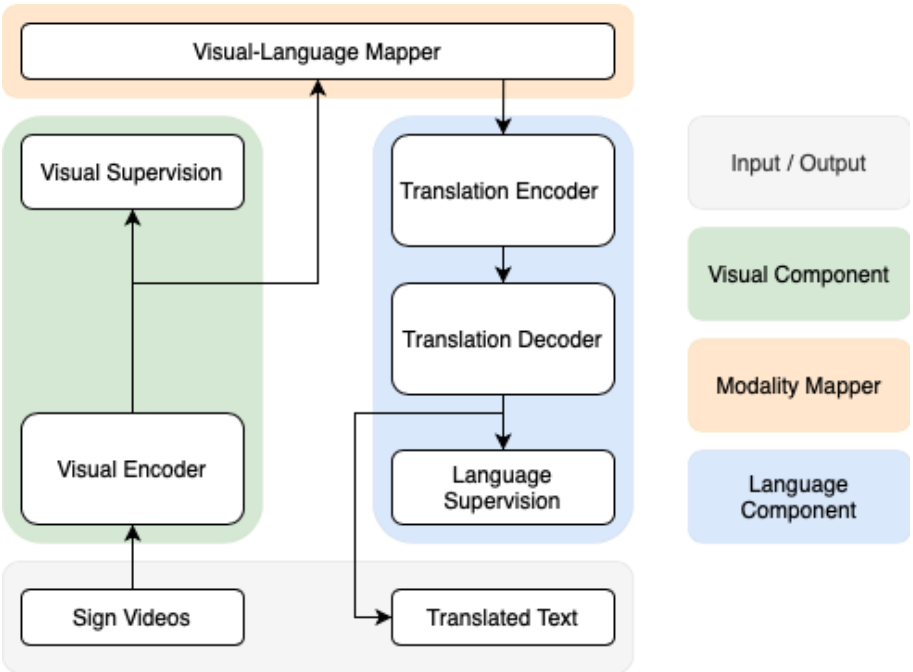


Figure 3: High level overview of the proposed Sign2Text model. The three main components of the model are decoupled into the visual component, language component and modality mapper.

Each model component is responsible for different parts of the sign language translation. Thus, the three main tasks Sign2Gloss, Gloss2Text and Sign2Text, can be assigned different model components.

Sign2Gloss. This refers to translating a sign language video into a corresponding gloss sequence. Since the signed sequences are temporally aligned with their respective gloss annotated translations, the language component of the model is not needed in this context. Thus, for the Sign2Gloss translation task, only the visual component is used and trained.

Gloss2Text. This task revolves around translating a gloss sequence into a natural language translation. During this pre-training procedure, the ground-truth gloss annotations are given as input. As such, the language component of the model is trained in isolation for the Gloss2Text task.

Sign2Text. This task refers to translating a sign language video directly into its corresponding natural language translation. To facilitate this, the visual and language component are connected using the visual-language mapper. Thus, the mapper acts as a bridge between the two modalities.

3.2 Sign2Gloss Task

This section covers our approach for translating sign language videos into gloss sequences. As mentioned in section 3.1, for the Sign2Gloss task, only the visual component of the model is used. This component of the model consists of a convolution-based "visual backbone", tasked with extracting visual features from the input video, followed by a translational head network which maps these features to glosses.

As a means to combat the scarcity of within-domain data, the visual backbone is pre-trained multiple times, before addressing the Sign2Gloss task. Initially the visual backbone is pre-trained on the general domain Kinetics-400 [12] dataset. This is followed by training on the general domain word-level dataset, WLASL [15], and lastly the within-domain sequence-level PHOENIX2014-T dataset [2]. This sequential pre-training procedure is shown in figure 4.

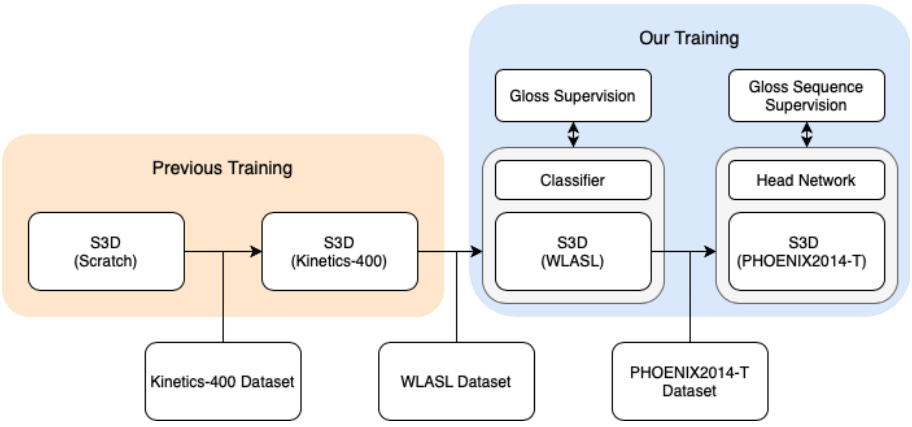


Figure 4: Overview of the sequential training procedure of the S3D backbone. Both Kinetics-400 and WLASL remains general-domain pretraining, while PHOENIX2014-T is within-domain.

The various dependencies as well as implementation details for the different training procedures outlined in figure 4 are presented in the following.

3.2.1 S3D

Following [5], we employ the Seperable 3D ConvNet (S3D) [29] as the visual backbone of the model. This model has been shown to provide consistently high performance across video datasets and requires fewer FLOPS[6] compared to other CNN architectures for video tasks. The S3D model is a modification on the Inflated 3D ConvNet (I3D) model [4].

I3D is based on the idea of pretraining 2D convolutional layers on an image dataset to learn spatial features. Then, copies of the resulting 2-dimensional kernels are stacked on top of each other, so as to "inflate" them into 3-dimensional kernels. The 3-dimensional kernels are then trained further on videos to learn temporal features as well. The end result is thus spatio-temporal 3D convolutional layers.

S3D abbreviates "separable 3D convolutions", which hints at the intuition behind the model. S3D separates the 3D spatio-temporal convolutions in I3D into 2D convolutions for the spatial features and 1D convolutions for the temporal features, and applies them in sequence.

Concretely, each 3D convolutional layer with kernel $f_{3D} = [k_t, k, k]$ is separated into two 3D convolutional layers with kernels $f_{2D} = [1, k, k]$ and $f_{1D} = [k_t, 1, 1]$, $k, k_t \in \mathbb{N}$. The singleton dimensions mean that these correspond to 2D and 1D convolutions, respectively, despite being defined in 3 dimensions.

Naturally, the separation of the 3D convolutions reduces the computational complexity of the model substantially.

Surprisingly however, the S3D authors observe improvements in performance when comparing S3D with I3D on both the Kinetics-400 [12] and Something-Something [8] datasets. They hypothesize that the separation of spatio-temporal convolutions reduces overfitting whilst retaining the quality of representations. This is backed up by the observation that simply reducing the number of model parameters of I3D does not yield an increase in performance. The full architecture of S3D is depicted in figures 5 and 6 below.

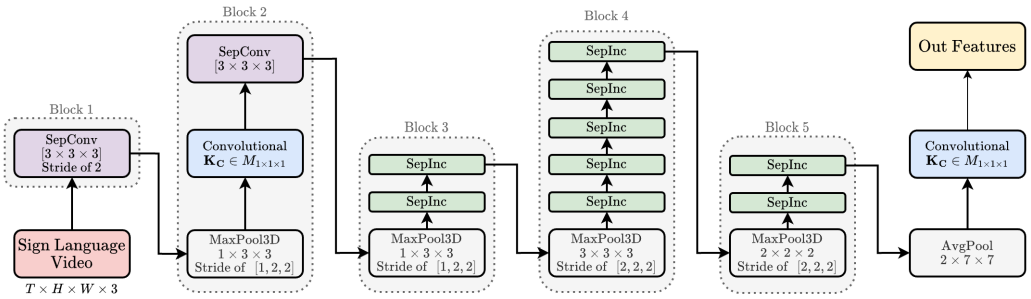
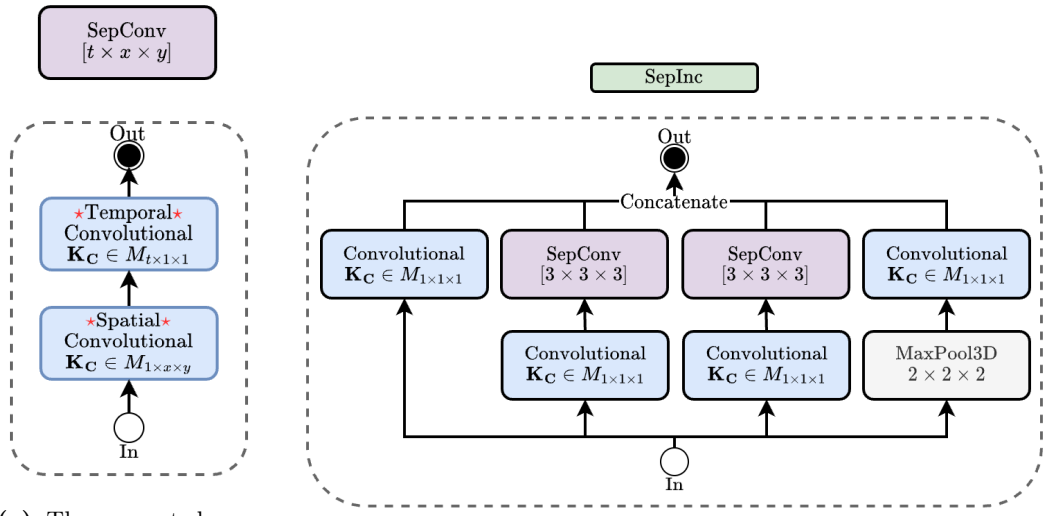


Figure 5: The full S3D architecture. The sub-blocks SepConv and SepInc are visualized in figure 6. Note that for our Sign2Gloss as well as full Sign2Text model pipeline, we only use blocks 1 through 4. $\mathbf{K_C}$ denotes the convolution kernel of the layer.

[6]Floating point operations.



(a) The separated convolutional block.

(b) The separated inception block.

Figure 6: Sub-blocks of S3D. $\mathbf{K_C}$ denotes the convolution kernel of the layer.

3.2.2 Classifier

To allow for predicting a single gloss of a given sign language video from the word-level WLASL dataset, the classification head of S3D is used. Firstly, average pooling is applied on the spatial dimension of the output of the final convolutional block of S3D, yielding a feature vector $\hat{\mathbf{y}} \in \mathbb{R}$. Following this, a single linear layer maps these features onto the gloss vocabulary as output logits. Lastly, the softmax function is applied to $\hat{\mathbf{y}}$ followed by argmax to obtain a gloss prediction corresponding to the input video.

3.2.3 Implementation Details - WLASL (Single Gloss)

Our pre-training procedure on WLASL largely follows [5]. The model is initialized with weights trained on the Kinetics-400 [12] and hereafter the model was trained for a total of 133 epochs using cross-entropy as the loss function. We use a batch size of 6, SGD as the optimizer with a momentum of 0.9 and an initial learning rate of 0.1. During training, we convert all videos to images, normalize them such that pixel values lie between [0, 1], and re-sample temporally to obtain a sequence length of 64. Several data augmentations are applied during training consistently for all images in a video. These include random horizontal flipping, random cropping to 224x224 pixels, random rotation within the interval $[-5°, 5°]$. Furthermore, a learning rate scheduler is used decaying the learning rate by a factor of 10 if no improvement in validation loss has occurred within 5 epochs. During inference, all frames of each input video are used. Additionally, random horizontal flipping and center cropping to 224x224 pixels is applied.

We find that the validation loss is saturated at epoch 91 and thus the weights of this model are used for subsequent fine-tuning.

3.2.4 Head Network

To allow for predicting a continuous sequence of glosses given a sign language video, we replace the last convolutional block of S3D as well as the previously mentioned classifier with a head network. Its architecture closely follows the one presented in [5]. As input, the head network takes a latent representation of the original sign language video $\mathbf{z} \in \mathbb{R}^{N \times T/4 \times 832}$ corresponding to the output of the penultimate convolutional block of S3D. Here N denotes the batch size and T is the number of frames in the video.

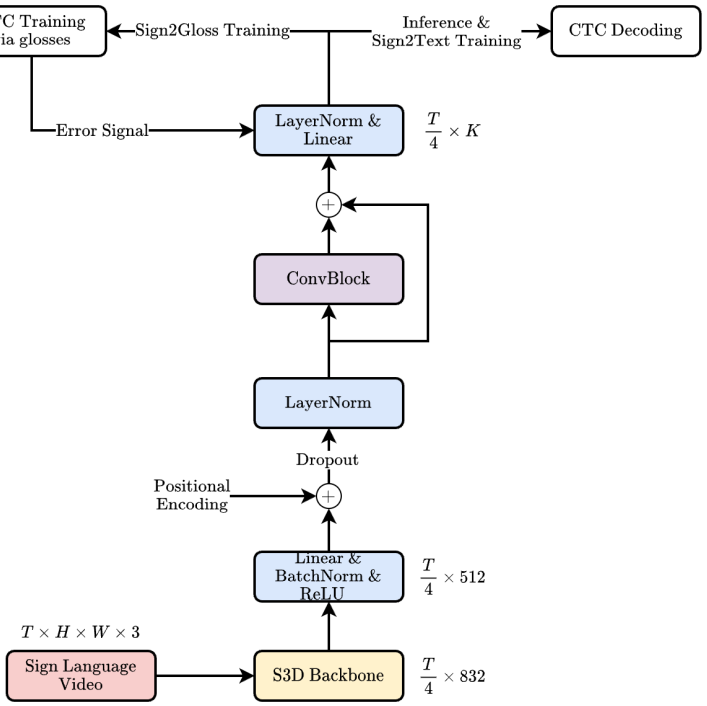
The first component of the head network is a projection block containing a linear layer followed by batch normalization and ReLU activation. Hereafter, positional encoding is applied as shown in equations 11 and 12 followed by a dropout layer.

Subsequently, layer normalization is applied and the input is passed through a convolutional block. This block consists of two convolutional layers with a kernel size of 3, ReLU activation and dropout in between. The first convolutional layer outputs features of dimension 2048 and the final output of this block has dimensions $N \times T/4 \times 512$. Additionally, resiudal connection is applied between the output of the projection layer and convolutional block.

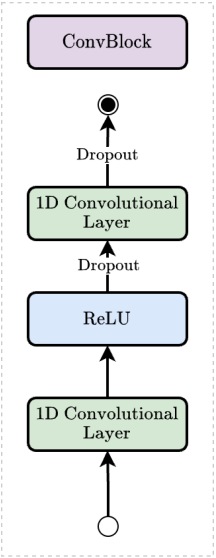
As its final component, the head network contains a translation layer. Here, layer normalization is first applied, followed by a linear projection from 512 to the size of the gloss vocabulary, K. The complete architecture of the vision model is summarized in figure 7 below.

Recall that the goal of the Sign2Gloss task is to predict a continuous gloss sequence given an input video. To facilitate this, we seek to temporally align the output features of the head network using the sentence level gloss labels. To this end, we employ Connectionist Temporal Classification (CTC) loss to supervise the visual encoder and CTC decoding for predicting gloss sequences. We will go into detail with CTC in the next section.

The architecture of the model responsible for the Sign2Gloss task is summarized in figure 7 below.



(a) Architecture of the the vision model.



(b) The ConvBlock composite function from (7a).

Figure 7: The complete Sign2Gloss architecture. Here T denotes the number of frames in the input video. At each step of the pipeline, dimensions are annotated should they change relative to the input. S3D backbone refers to blocks 1 through 4 of the architecture depicted in figure 5.

3.2.5 CTC Loss and Decoding

This section covers the theoretical framework surrounding CTC loss as well as decoding. Crucially, the objective of CTC is to find the most probable output gloss labeling for a given input. This corresponds to the labeling where the set of paths through the search space that lead to it have the highest cumulative probability.

A key assumption for CTC loss is that the order in which the labels are observed in the input signal is consistent with the order of the target labelings. This is exactly the case for our gloss labels, but not for our natural language labels. This detail is unfortunately also what limits this method, since the need for continuous gloss labels is the bottleneck for acquiring larger datasets.

CTC Problem Formulation. To use CTC, softmax is applied to the output of the translation layer returned from the head network. We denote this joint network as $\mathcal{N}_w$. We interpret the output from $\mathcal{N}_w$ as a $\tau = \frac{T}{4}$ long sequence of probability distributions of labels - one per time index. Note that our temporal dimension has been aggregated by the convolutional operations in $\mathcal{N}_w$, meaning that each index no longer directly corresponds to a single frame in the video.

We index the sequence of probability distributions y as $\left[y_k^t\right]$, meaning the probability of observing label k at time t. We will denote the input data used to compute

y as x, and the target label sequence for x as z.

Since we want a way to handle indices in the sequence where no gloss is being signed, we define an alphabet $A' = A \cup \{\text{blank}\}$, where A is our gloss vocabulary and blank denotes a label for the case of no gloss. A' will be used in an intermediate representation for computing the labeling probabilities.

For translating between our intermediate representation in A' and our target in A, we let $\mathcal{B}$ denote a map that filters blank-labels and collapses adjacent repeated labels in a sequence, e.g. $\mathcal{B}([a, b, b, \text{blank}, c, b,]) = [a, b, c, b]$.

We can now express the probability of a given label sequence $\mathbf{l} \in A^t$ by summing the probabilities of each possible alignment mapping to it from $(A')^{\geq t}$. We write this as

$$p(\mathbf{l}|x) = \sum_{\pi \in \mathcal{B}^{-1}(\mathbf{l})} p(\pi|x) \tag{1}$$

Here, $p(\pi|x)$ denotes the probability of a label sequence $\pi \in \mathcal{B}^{-1}(\mathbf{l}) \subseteq (A')^{\geq t}$, i.e. any label sequence that is mapped to $\mathbf{l}$ by $\mathcal{B}$. This probability can be expressed as a product of all the individual probabilities in the sequence:

$$p(\pi|x) = \prod_{t=1}^{\tau} y_{\pi_t}^t \tag{2}$$

As the set $\mathcal{B}^{-1}(\mathbf{l})$ quickly becomes extremely large, $p(\mathbf{l}|x)$ is clearly no trivial computation. Therefore, we will introduce a dynamic programming algorithm that includes only the needed terms when performing the computation.

Tractable computation of label sequence probability. The algorithm in question is the CTC Forward-Backward Algorithm and is first described in [9].

We first index a target label sequence $\mathbf{l}$ of length r by its first p and last p symbols, as $\mathbf{l}_{1:p}$ and $\mathbf{l}_{r-p:r}$, respectively.

Then we modify the sequence to allow blank labels as start and end tokens as well as in between all of the labels. We denote this modified sequence as $\mathbf{l}'$. The modified sequence will have length $|\mathbf{l}'| = 2|\mathbf{l}| + 1$.

We now consider a graphical model of $\mathbf{l}'$, where the nodes are the labels and the outgoing transitions signify the possible postfixes. The graph is partially ordered according to the time and ordering of the ground truth label.

A visualization of the general graph can be seen in figure 8. Furthermore, an example of such a graph for the sequence $\mathbf{l}'$ corresponding to $\mathbf{l}$ = [NAECHSTE, WOCHE, REGEN] is illustrated in figure 9.

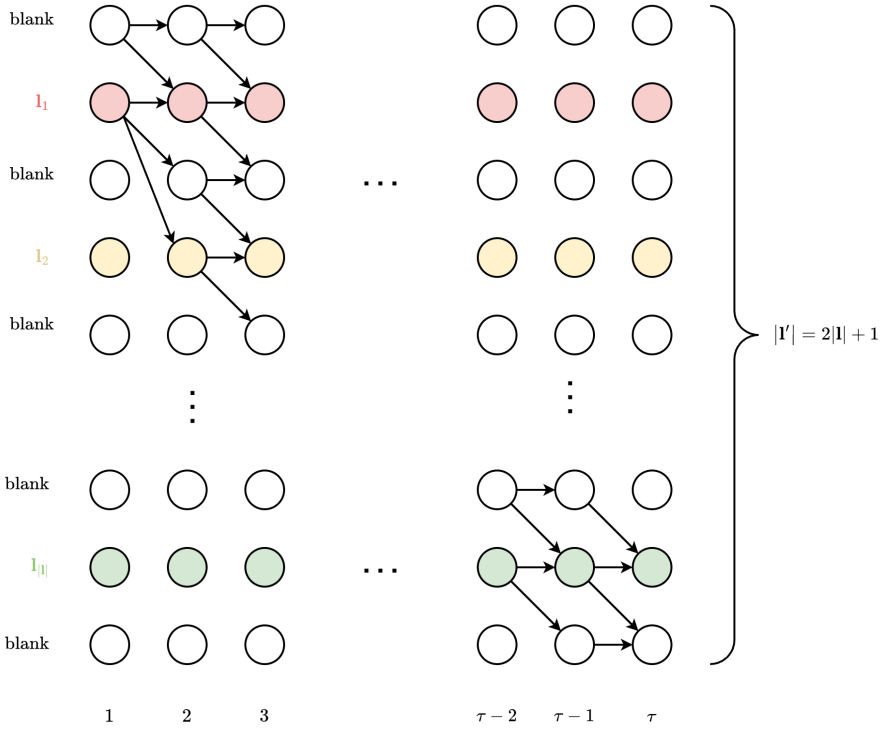


Figure 8: Visualization of the partially ordered graph for a gloss sequence $\mathbf{l}'$ over time length τ.

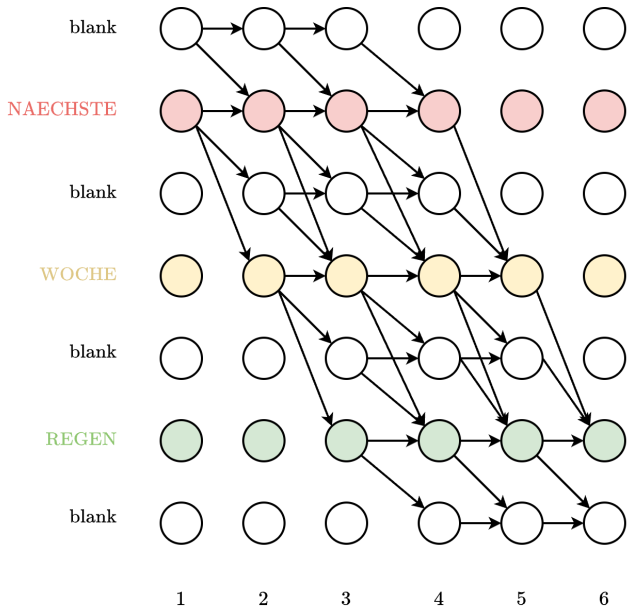


Figure 9: Visualization of the partially ordered graph corresponding to all paths that map to the gloss sequence [NAECHSTE, WOCHE, REGEN] given time horizon $\tau = 6$. Unconnected notes correspond to individual labels in impossible paths, given the label sequence $\mathbf{l}'$.

Note that the unconnected nodes represent labels that have probability 0 at the given time index. In these cases, the unconnected notes represent labels the sequence cannot start in and those where the full sequence is no longer reachable inside the timesteps.

Recalling that y_k^t is the probability of observing label k at time t, we can now define the *forward* variables, α, and *backward* variables, β, as

$$\alpha_t(s) = \sum_{\substack{\pi \in (A')^\tau : \\ \mathcal{B}(\pi_{1:t}) = \mathbf{l}_{1:s}}} \prod_{t'=1}^{t} y_{\pi_{t'}}^{t'} \tag{3}$$

$$\beta_t(s) = \sum_{\substack{\pi \in (A')^\tau : \\ \mathcal{B}(\pi_{t:\tau}) = \mathbf{l}_{s:|\mathbf{l}|}}} \prod_{t'=t}^{\tau} y_{\pi_{t'}}^{t'} \tag{4}$$

These variables express the total probability, at time t, of labels $\mathbf{l}_{1:s}$ for $\alpha_t(s)$ and $\mathbf{l}_{s:|\mathbf{l}|}$ for $\beta_t(s)$. They are the basis for constructing a recursive relationship that allows us to include only probabilities which we cannot be certain are zero in the computation of $p(\mathbf{l}|x)$.

Note that the probability of $\mathbf{l}$ can be found as the sum of $\mathbf{l}'$ with and without a blank at the final timestep:

$$p(\mathbf{l}|x) = \underbrace{\alpha_\tau(|\mathbf{l}'|)}_{\text{blank at } \tau} + \underbrace{\alpha_\tau(|\mathbf{l}'|-1)}_{\mathbf{l}_{|\mathbf{l}|} \text{ at } \tau} \tag{5}$$

From the graphical model we can derive the recursion initialization rules for α, from the probabilities at $t = 1$, as

$$\begin{aligned} \alpha_1(1) &= y_{\text{blank}}^1 \\ \alpha_1(2) &= y_{\mathbf{l}_1}^1 \\ \alpha_1(s) &= 0, \forall s > 2 \end{aligned}$$

and for β, from the probabilities at $t = \tau$, as

$$\begin{aligned} \beta_\tau(|\mathbf{l}'|) &= y_{\text{blank}}^\tau \\ \beta_\tau(|\mathbf{l}'|-1) &= y_{\mathbf{l}_{|\mathbf{l}|}}^\tau \\ \beta_\tau(s) &= 0, \forall s < |\mathbf{l}'|-1 \end{aligned}$$

Furthermore, we can express the recursion rules for computing $\alpha_t(s)$ and $\beta_t(s)$ for a given s. For α:

$$\alpha_t(s) = \begin{cases} \hat{\alpha}_t(s) y_{l'_s}^t & l'_s = \text{blank or } l'_{s-2} = l'_s \\ (\hat{\alpha}_t(s) + \alpha_{t-1}(s-2) y_{l'_s}^t) & \text{otherwise} \end{cases}$$

where

$$\hat{\alpha}_t(s) = \alpha_{t-1}(s) + \alpha_{t-1}(s-1)$$

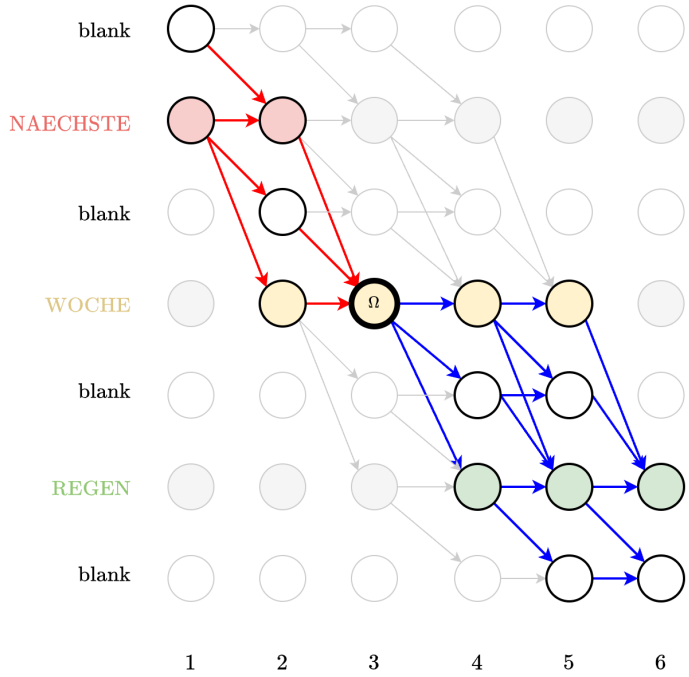


Figure 10: Visualization of the graph transitions corresponding to the sequences considered when calculating the *forward* variable $\alpha_t(s)$ (red transitions) and *backward* variable $\beta_t(s)$ (blue transitions) for the node $\Omega = (t, \mathbf{l}_s) = (3, \text{WOCHE})$ from the graph in figure 8

.

For β:

$$\beta_t(s) = \begin{cases} \hat{\beta}_t(s) y^t_{l'_s} & l'_s = \text{blank or } l'_{s+2} = l'_s \\ (\hat{\alpha}_t(s) + \alpha_{t+1}(s+2) y^t_{l'_s}) & \text{otherwise} \end{cases}$$

where

$$\hat{\beta}_t(s) = \beta_{t+1}(s) + \beta_{t+1}(s+1)$$

Figure 10 contains a visualization of paths considered in calculating the *forward* and *backward* variables.

Next, we use (3) and (4) to combine the *forward* and *backward* variables into a useful result:

$$\alpha_t(s)\beta_t(s) = \underbrace{\left(\sum_{\substack{\pi \in (A')^\tau : \\ \mathcal{B}(\pi_{1:t}) = \mathbf{l}_{1:s}}} \prod_{t'=1}^{t} y^{t'}_{\pi_{t'}} \right)}_{\substack{\text{Sum over subset of } (A')^\tau \\ \text{that map to } \mathbf{l}_{1:s}}} \underbrace{\left(\sum_{\substack{\pi \in (A')^\tau : \\ \mathcal{B}(\pi_{t:\tau}) = \mathbf{l}_{s:|\mathbf{l}|}}} \prod_{t'=t}^{\tau} y^{t'}_{\pi_{t'}} \right)}_{\substack{\text{Sum over subset of } (A')^\tau \\ \text{that map to } \mathbf{l}_{s:|l|}}} = \underbrace{\sum_{\substack{\pi \in \mathcal{B}^{-1}(\mathbf{l}) : \\ \pi_t = \mathbf{l}_s}} y^t_{\mathbf{l}_s} \prod_{t'=1}^{\tau} y^{t'}_{\pi_{t'}}}_{\substack{\text{Sum over all paths } \pi \\ \text{that map to } \mathbf{l} \\ \text{and have } \pi_t = \mathbf{l}_s}}$$

Rearranging this and using (2), we obtain

$$\frac{\alpha_t(s)\beta_t(s)}{y^t_{\mathbf{l}_s}} = \sum_{\substack{\pi \in \mathcal{B}^{-1}(\mathbf{l}) : \\ \pi_t = \mathbf{l}_s}} p(\pi|x) \tag{6}$$

This means that the sum of probabilities of π that map to $\mathbf{l}$ and has $\mathbf{l}_s$ at timestep t can be expressed as the product of $\alpha_t(s)$ and $\beta_t(s)$ divided by the probability of observing label $\mathbf{l}_s$ at time t.

Finally, we can insert (6) into (1) to express the probability of $\mathbf{l}$ from $\alpha_t(s)$ and $\beta_t(s)$:

$$p(\mathbf{l}|x) = \sum_{\pi \in \mathcal{B}^{-1}(\mathbf{l})} p(\pi|x) = \sum_{t=1}^{\tau} \sum_{s=1}^{|\mathbf{l}|} \frac{\alpha_t(s)\beta_t(s)}{y^t_{\mathbf{l}_s}} \tag{7}$$

This last expression (7) is crucial, as it provides a computationally tractable way to compute the probability of a labeling $\mathbf{l}$ given input data x, using the output sequences of probability distributions from our visual head network. In practice, the α and β variables are scaled to avoid numerical and computational issues, but this just involves some simple division terms along the way [9].

The probability can now be used for maximum likelihood training, i.e. minimizing the negative log-likelihood.

CTC Training Objective. For maximum likelihood training, we wish to maximize the likelihood of our visual encoder correctly predicting the ground truth label sequences, given all their corresponding paths.

Recall that $\mathcal{N}_w$ is the joint network that outputs a τ long sequence of probability distributions. Denoting our distribution of training examples as $S \subset \mathcal{D}_{x \times z}$, we can express the maximum likelihood objective O^{ML} as minimization of

$$O^{\text{ML}}(S, \mathcal{N}_w) = -\left(\sum_{(x,z) \in S} \log(p(z|x)) \right)$$

Since the training examples $(x, z) \in S$ are independent, a partial derivative corresponding to a component in the gradient can be expressed as

$$\frac{\partial O^{\text{ML}}(\{(x,z)\}, \mathcal{N}_w)}{\partial y^t_k} = -\frac{\partial \log(p(z|x))}{\partial y^t_k} \tag{8}$$

Using the chain rule on (8), we get

$$\frac{\partial O^{\text{ML}}(\{(x,z)\}, \mathcal{N}_w)}{\partial y^t_k} = -\frac{1}{p(z|x)} \frac{\partial p(z|x)}{\partial y^t_k} \tag{9}$$

We can express $\frac{\partial p(z|x)}{\partial y^t_k}$ by setting $\mathbf{l} = z$ in (7) and taking the partial derivative w.r.t. y^t_k:

$$\frac{\partial p(z|x)}{\partial y^t_k} = \frac{\partial}{\partial y^t_k} \left(\sum_{t=1}^{\tau} \sum_{s=1}^{|z|} \frac{\alpha_t(s)\beta_t(s)}{y^t_{z_s}} \right) = -\frac{1}{(y^t_k)^2} \sum_{s \in \{s' : z_{s'} = k\}} \alpha_t(s)\beta_t(s) \tag{10}$$

Here, $\sum_{s \in \{s' : z_{s'} = k\}}$ is a sum over the set of positions where k occurs.

Finally, we can set $z = \mathbf{l}$ in (5) and insert it, as well as (10), into (9) to differentiate the objective function and thus obtain the error signal propagated back into the head network.

For a given data point (x, z), this becomes:

$$\frac{\partial O^{\mathrm{ML}}(\{(x,z)\}, \mathcal{N}_w)}{\partial y_k^t} = \left(\alpha_\tau(|z'|) + \alpha_\tau(|z'|-1)\right)^{-1} \frac{1}{(y_k^t)^2} \sum_{s \in \{s' : z_{s'} = k\}} \alpha_t(s)\beta_t(s) \quad \square$$

CTC Decoding. After successfully training the network with CTC loss, the distance between the output gloss sequence probabilities and the ground truth glosses will be minimized. Hereby, we hope that these output distributions will align with the distribution of never before seen data.

We now want to turn these probability distributions into the most probable gloss sequence, such that these can be validated with the ground truth glosses.

Using the notation of the previous section, we can express the most probable gloss sequence given the network output as

$$h(x) = \underset{\mathbf{l} \in A^{\leq \tau}}{\operatorname{argmax}}\, p(\mathbf{l}|x)$$

Once again, this is non-trivial to compute, as the space we are searching when computing argmax quickly grows extremely large as vocabulary A and time horizon τ increase in size.

This can be done naïvely using the computationally trivial greedy decoding scheme Φ_{greedy}, which gives the concatenation of the most probable gloss for each timestep, i.e. Φ_{greedy} yields $\left[\forall t \in \tau : y_*^t = \underset{k}{\operatorname{argmax}}\, y_k^t\right]$.

However, Φ_{greedy} has the assumption that the most probable path corresponds to the most probable labeling, which is not necessarily the case. As can be seen in figure 9, we can trace multiple paths through the probability distributions that map to the same labeling via $\mathcal{B}$. The true probability of a labeling is the sum of all of these, as can be seen in (1).

An example of the greedy decoding algorithm giving a result differing from the true most likely labeling can be seen in the appendix files under `CTC_decoding.ipynb`.

However, since we do not want to compute the probability of all possible labelings for each sequence to find the most probable one, we use a "beam search" algorithm. What beam search does essentially, is to start with an empty sequence of prefixes and their probabilities σ, and then iterate over each timestep $t = 1, 2, \ldots, \tau$. For each step, we compute the probabilities of extensions of the prefixes in σ by the labels in our vocabulary and add them to σ. In order to keep the computational cost down, σ is usually pruned to support a constant number of prefixes at the end of each iteration.

The search algorithm used for the CTC decoding is prefix search, a special case of beam search. Prefix search decoding modifies the forward-backward algorithm for exactly this purpose. We exploit that the forward variable $\alpha_t(s)$ already has a

recursive relation to describe the probability of the prefix $\mathbf{l}_{1:s}$ at time-step t.

To sum it up, what all of these steps give us is an efficient computation of the most probable labeling $\mathbf{l}$, i.e. the most probable gloss sequence, for a given input video x.

3.2.6 Implementation Details - PHOENIX-2014T (Gloss-Sequence)

In alignment with the SOTA model [5], we train our visual encoder using the AdamW optimizer with a weight decay and initial learning rate of 10^{-3}. Additionally, cosine annealing is used as the learning rate scheduler with $T_{max} = n_{epochs}$. The model was trained for 100 epochs with a batch size of 6 using CTC loss as the objective.

During training, all images are initially upsampled pixel-wise to 298x240 using bi-linear interpolation. After this, the image is normalized such that pixel values lie between [0,1]. This is followed by color jitter where we randomly adjust the brightness, contrast and saturation all within the range [0.6, 1.6], and the hue within the range [-0.1, 0.1]. Hereafter, a 240x240 center crop is taken to ensure that hand and face movements remain present in the final video. Follwing this, the video is rotated randomly with a range of $[-5°, 5°]$ followed by random cropping to 224x224.

During inference the images are upsampled to 298 x 240, normalized and hereafter a 224 x 224 center crop is taken.

3.3 Gloss2Text Task

This section covers the methodology surrounding translation of a gloss sequence into natural language. As mentioned in section 3.1, only the language component of the model is utilized and trained for this task. As the language component of the model we use mBART [18], a denoising autoencoder, in alignment with SOTA [5]. This model is a variant of BART [14] that is pre-trained on CC-25 [18], a large corpus of multilingual data. BART consists of a bi-directional encoder and an autoregressive decoder. The underlying architecture of BART is a sequence-to-sequence transformer as first proposed in Attention is All You Need (AiAYN) [28]. The only deviations from the original architecture are that GeLU is used as the activation function as opposed to ReLU and the latent model dimension and number of layers are increased.

3.3.1 Transformers

Transformers take sequential data as input and, through learnable parameters, they have the ability to decide how much different parts of an input sequence should contribute to the generated output sequence.

The first step in the transformer pipeline is tokenization of the input sequence. Here sub-words in a sentence are mapped to integers, corresponding to their row indices in an embedding matrix. The tokenized input sentence is then projected using the embedding matrix to obtain an embedding vector for each token in the input sequence.

Let $\tilde{\mathbf{x}} \in R^{N \times K}$ and $\mathbf{E} \in \mathbb{R}^{K \times d_{model}}$ denote a one-hot encoded, tokenized sequence and an embedding matrix, respectively. Here K denotes the vocabulary size, N is

the length of the input sequence and d_{model} is the latent model dimension. The embedded sequence, in matrix-form with embeddings as rows, is then given as $\mathbf{z} = \tilde{\mathbf{x}}\mathbf{E}$.

Positional Encodings. One of the later steps in the Transformer pipeline, self-attention, is permutation invariant. This means that no temporal information from the ordering of the sequence is retained. Naturally, this is undesirable in the context of NLP since the ordering of the input contains a lot of syntactical and semantic information. Thus, in order to propagate information about the relative positions of elements in the input sequence, we add positional encodings to $\mathbf{z}$. Here [28] proposes to use sinusoidal functions on the form:

$$\mathbf{PE}_{p,2i+1} = \cos\left(\frac{p}{10000^{2i/d_{model}}}\right) \tag{11}$$

$$\mathbf{PE}_{p,2i} = \sin\left(\frac{p}{10000^{2i/d_{model}}}\right) \tag{12}$$

Where $p \in \{0, 1, \ldots N-1\}$ denotes the position in the sequence, i.e. row index in $\mathbf{z}$, and $i \in \{0, 1, \ldots, d_{model}-1\}$ refers to the embedding dimension. Since the positional encodings remain constant across input sequences, it allows the model to learn the positions as well as relative distances between elements. Hence, the input to the transformer becomes $\mathbf{x} = \mathbf{z} + \mathbf{PE}$.

Below, in figure 11, is a plot of the positional encodings to visualize how they are providing positional information.

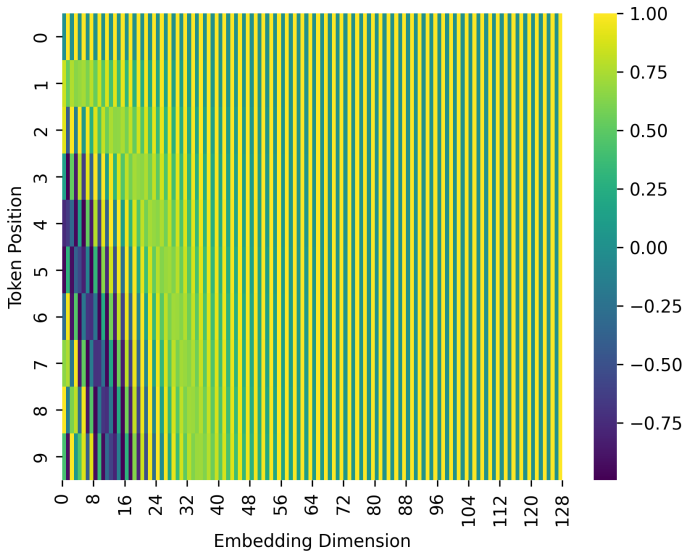


Figure 11: Visualization of the positional encodings. Note that the plot only depicts 128 embedding dimensions, whereas our model has 1024.

Attention. The backbone of the transformer architecture is the attention mechanism, in particular multi-head attention. To understand this, we first consider self-attention, i.e. what can be understood as single-head attention in this context. Given an embedded input sequence $\mathbf{x} \in \mathbb{R}^{N \times d_{model}}$, we compute three matrices,

namely the query, key and value matrices, as:

$$\mathbf{Q} = \mathbf{x}\mathbf{W}^Q \quad \mathbf{K} = \mathbf{x}\mathbf{W}^K \quad \mathbf{V} = \mathbf{x}\mathbf{W}^V$$

Here, $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d_{model} \times d_k}$ are learned projection matrices. Using these, the attention operation is computed as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\Big(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\Big)\mathbf{V} \tag{13}$$

Here the term $\mathbf{Q}\mathbf{K}^\top$ is a square matrix with dimension N containing the relevance score for each element in relation to every other element in the input sequence. These values are scaled by $\frac{1}{\sqrt{d_k}}$ to make the gradients more stable for large values of d_k. The softmax term converts the scores to a relative importance weighting, summing to exactly 1, and by taking the dot product with $\mathbf{V}$, a weighted average over the values is obtained. A visualization of the self-attention mechanism for a given gloss sequence is provided in figure 12.

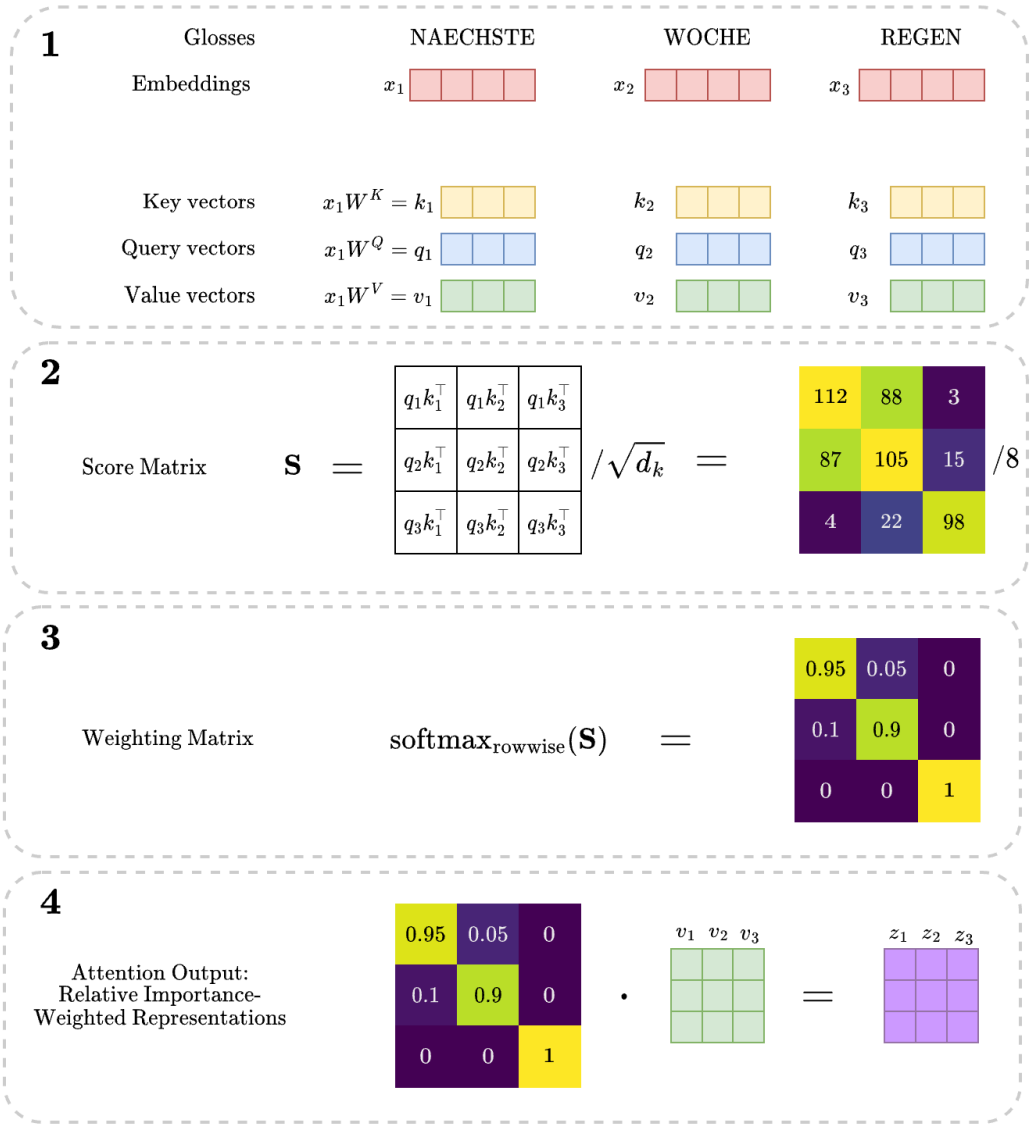


Figure 12: Visualization of self-attention applied on our example gloss sequence [NAECHSTE, WOCHE, REGEN]. The values for q_ik_j in pane 2 are made up, and d_k is selected to be 64 in line with [28]. Note that the dimension of the "vectors" is not specified - only the relative dimensions are important.

Multi-head Attention. Multi-head attention improves on the attention mechanism by computing self-attention, as seen in (13), for h attention heads each with different sets of projection matrices $\{\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V\}_{i=1}^h$. Hence, h attention matrices $\mathbf{A}_i \in \mathbb{R}^{N\times d_k}$ are obtained and hereafter they are concatenated into $\tilde{\mathbf{A}} \in \mathbb{R}^{N\times h\cdot d_k}$. The final output of multi-head attention is then a linear projection given as

$$\text{MultiHeadAttention}(\tilde{\mathbf{A}}) = \tilde{\mathbf{A}}\mathbf{W}^O, \quad \mathbf{W}^O \in \mathbb{R}^{h\cdot d_k \times d_{model}} \tag{14}$$

Having multiple heads, allows the attention layer to learn several latent representations thereby enhancing the models ability to relate and connect different parts of the input. A key feature is also that attention for each of the heads can be computed

in parallel, leading to massive computational advantages over sequential model architectures.

Encoder. The transformer architecture is composed of an arbitrary amount of encoder stacks all of which contain of two layers. In between each layer, a residual skip connection is applied, followed by layer normalization. Hence, for an input $\mathbf{x}_{in}$ and output $\mathbf{x}_{out}$ of a layer, the computation is LayerNorm($\mathbf{x}_{in} + \mathbf{x}_{out}$). The first layer of the encoder is multi-head attention and the second is a standard fully-connected feed forward neural network with two layers, ReLU activation in between and a hidden dimension of d_{ff}. The output of the feed forward network is a matrix of size $\mathbb{R}^{N \times d_{model}}$.

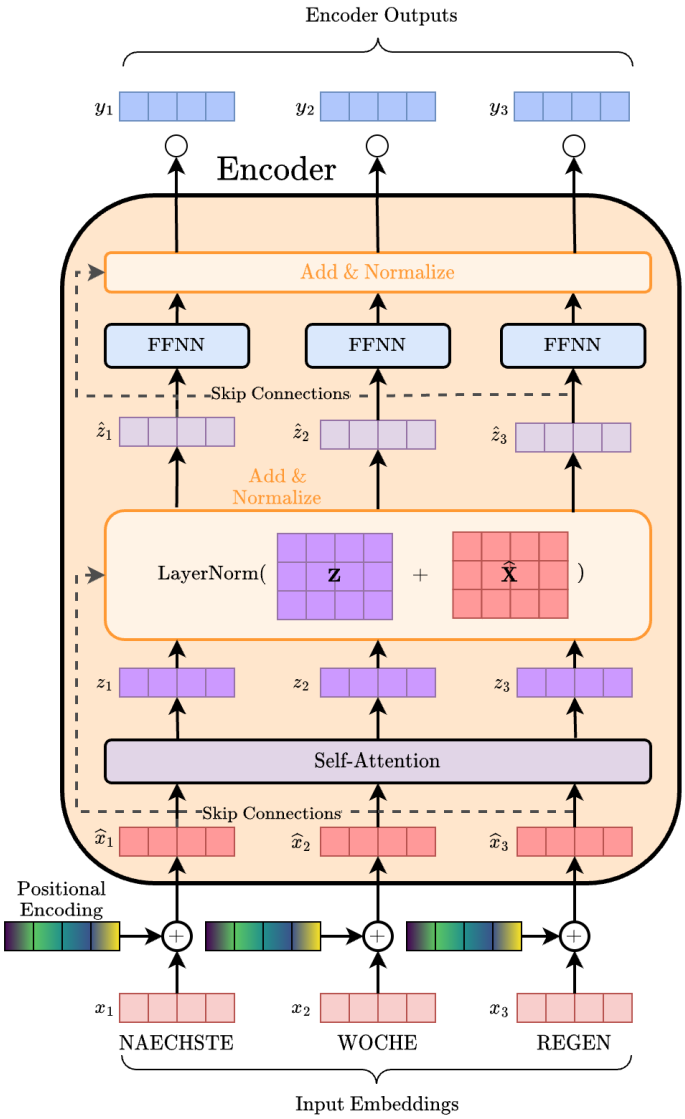


Figure 13: Diagram of the transformer encoder. FFNN denotes a feed forward neural network.

Decoder. The decoder of a transformer shares the components and structure of

the encoder with the exception of a third layer that computes multi-head attention over the final output of the encoder stack. Also, to ensure that the model does not attend to parts of the input at future time-steps, the decoder input is offset by one position and masking is applied. The latter is implemented as a sum of the score matrix (see Figure 12) and a strictly upper triangular matrix with $-\infty$ in its non-zero indices. This corresponds to setting subsequent time-steps to $-\infty$ relative to the current time-step when applying softmax during attention, as each row corresponds to a later time-step.

The masking step is illustrated in Figure 15.

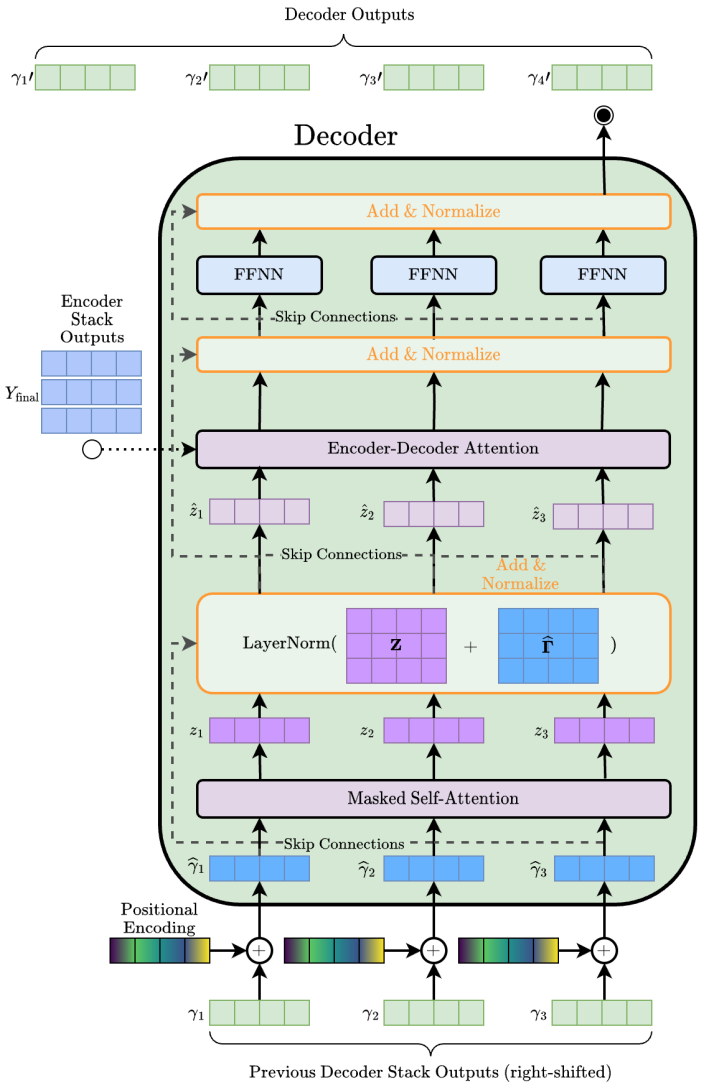


Figure 14: Diagram of the transformer decoder. FFNN denotes a feed forward neural network. Y_{final} denotes the output embedding sequence from the encoder stack. Encoder-Decoder Attention denotes the multi-head attention over the final output of the encoder stack. The decoders each output embeddings one at a time. Each decoder uses the sequence of previous outputs of the decoder below it in the stack as inputs. The first decoder uses the previous outputs of the full decoder stack, however - in the first step, it receives a beginning-of-sentence token, corresponding to a 'right-shift'.

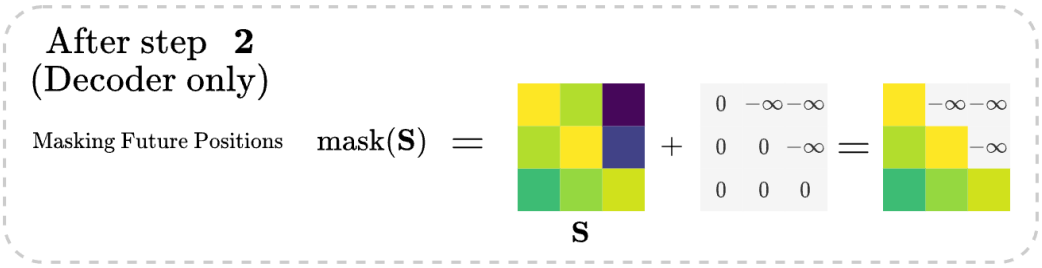


Figure 15: Visualization of the masking step in the decoder masked self-attention. Step 2 refers to the steps in figure 12.

Finally, the architecture of the transformer is summarized in figure 16 below. Here, the relation between encoder and decoder stacks is illustrated, along with how they work together in the context of sequence to sequence mapping.

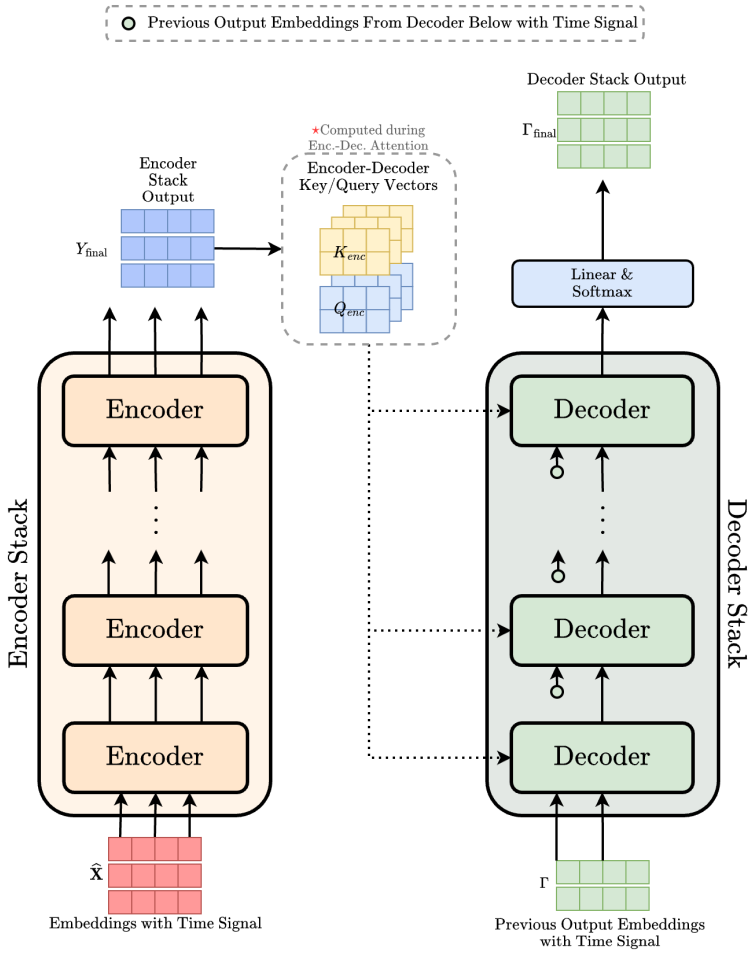


Figure 16: Diagram of the structure of the full transformer and how the encoder and decoder stacks are connected.

3.3.2 mBART

As noted earlier, mBART consists of a bi-directional encoder (stack) and an autoregressive decoder (stack).

In this context, bidirectionality is simply an explicit reference to the fact that attention is computed for the entire input sequence at once within each encoder. This is identical to the procedure described above and outlined in AiAYN [28].

The decoder being autoregressive refers to the fact that the decoder stack outputs a single embedding at a time, which is then fed back into the bottom-most decoder. This means that the each new output is conditioned on (all of) its previous outputs.

The parameters of the mBART architecture are summarized in table 4 below.

Parameter	Value	Description
d_{model}	1024	Embedding dimension
d_{ff}	4096	Latent dimension in feed forward networks.
d_k	64	Scaling factor in self-attention
h	16	Number of attention heads in Multi-head attention.
n_{stack}	12	Size of stacks of encoders/decoders.
n_{params}	∼680M	Total number of model parameters.

Table 4: The parameters of mBART-cc25, used as the language component for our model.

In spite of the architectural similarities between mBART and AiAYN [28], there are also notable differences in their respective training procedure. mBART follows BART [14] in terms of pre-training objectives. Thus, the model is pre-trained to reconstruct documents, purposefully corrupted by noise. This allows mBART to achieve high performance on both the sentence level, as well as document level.

One of the pre-training tasks is text infilling, where spans of text are randomly sampled from the input and replaced with a single <MASK> token. The length of each span is determined by drawing samples from a Poisson distribution with $\lambda = 3.5$. The model is then tasked with predicting the correct word(s) in each span.

Sentence permutation is also used which involves splitting a document into sentences and randomly shuffling them. Then, the model is tasked with reconstructing the correct order of sentences.

During pre-training, the input consists of several sentences from the same document each separated by an end-of-sentence <EOS> token. Sentences are sampled from a document until either the maximal sequence length of 512 is reached or the document has ended. To mark the end of an instance, a language id <LID> token is added as the final token, potentially followed by padding.

Let $\mathcal{D} = \{\mathcal{D}_1, ..., \mathcal{D}_n\}$ denote a multilingual corpus of text where $\mathcal{D}_i$ is a monolingual document. The objective of the training is then to maximize the following w.r.t. the model

$$\mathcal{L}(\mathbf{x}) = \sum_{\mathcal{D}_i \in \mathcal{D}} \sum_{\mathbf{x} \in \mathcal{D}_i} \ln(P(\mathbf{x}|g(\mathbf{x})) \tag{15}$$

Where $\mathbf{x}$ is a span of text in $\mathcal{D}_i$ and g is a noising function for corrupting the input text. Hence, g can either correspond to text infilling or sentence permutation.

During pre-training, the authors of mBART investigate the effect of pre-training on different languages. The variant of mBART we use has been pre-trained on all languages present in the cc-25 dataset [18]. During fine-tuning, the encoder of the model is given an single input sentence $\mathbf{x}$ in the source language with the format:

$$\mathbf{x} \text{ <EOS> <LID>}$$

The decoder is tasked with autoregressively predicting the next token as described in the previous section. The input to the decoder at the first time-step is the <LID> corresponding to the target language. The format of the ground-truth translation $\mathbf{y}$ in this context becomes

$$\text{<LID> } \mathbf{y} \text{ <EOS>}$$

When fine-tuning mBART for NMT, the cross entropy between complete output of the decoder and ground-truth translation is minimized.

3.3.3 Beam Search Decoding

To find an optimal translation given an input video in a computationally efficient manner, we once again employ a method of beam search decoding. This variation beam search can be thought of as a pruned BFS where the branching factor is constrained to a fixed number of competing hypotheses, k. Let N denote the length of a sequence and $\mathbf{e}$ the output of the encoder. The goal is to find a sequence of words $\mathbf{y}$ that maximizes $\Pi_{t=1}^{N} P(y_t|y_1, ..., y_{t-1}, \mathbf{e})$.

At each time-step t, the algorithm outputs the k most probable hypotheses conditioned on the previously predicted words $y_1, ...y_{t-1}$ and $\mathbf{e}$. Recall that the decoder of a transformer is autoregressive and computes exactly this . Hence, at each time-step, the decoder computes $P_\theta(y_t|y_1, .., y_{t-1}, \mathbf{e})$ where θ denotes its parameters. P_θ is thus used to score the hypotheses at each time-step of the beam search decoding process.

Algorithm 1 Beam search decoding

Goal: Find a sequence $\mathcal{Y} = \{y_1, ..., y_N\}$ maximizing $\Pi_{t=1}^{N} P(y_t|y_1, ..., y_{t-1}, \mathbf{e})$
Recall the decoder computes $P_\theta(y_t|y_1, .., y_{t-1}, \mathbf{e})$, where θ denotes its parameters.
Let $k \in \mathbb{N}$ denote the number of competing hypotheses.
Let $\circ$ denote a string concatenation operation.
Let $\mathbf{e}$ denote the output of the encoder of the model.
Let $\mathcal{V}$ denote the vocabulary of the language model.
Let $\mathcal{Y}$ denote a set $\{\mathbf{y}_i\}_{i=1}^{k}$ of sequences initialized with the the BOS token.

expandHypotheses(Y, k, $\mathbf{e}$)
  $k_{best} \leftarrow \emptyset$
  $scores \leftarrow \{\}$ // define empty map
  for i from 1 to k **do**
    for v in $\mathcal{V}$ **do**
      scores$[Y_t^k \circ v] \leftarrow P_\theta(v|Y_k, \mathbf{e})$
  $scores' \leftarrow$ **argsort**($scores.values()$) // sort scores in descending order
  for j from 1 to k **do**
    $sequence$, $score \leftarrow scores'$.pop()
    k_{best}.add($sequence$) // add sequence
  return k_{best}

BeamSearch($\mathbf{e}$, k)
  for t from 1 to n_{max} **do**
    $\mathcal{Y} \leftarrow$ **expandHypotheses**$(\mathcal{Y}, k, \mathbf{e})$
    if $\forall\ \mathcal{Y}_k^t =$ <EOS>
      break
  return $\mathcal{Y}$

3.3.4 Implementation Details PHOENIX-2014T

In alignment with the SOTA model [5], we implement a transformer based translation network using the mBART-cc25 model. Hence, this is a variant of mBART that is pre-trained on all source languages present in its training data.

To alleviate computational overhead the embedding matrix of mBART is pruned. Hence, all rows of the embedding matrix corresponding to tokens that are not present in the vocabulary are removed. Additionally, the embedding matrix is frozen during training in alignment with [5], to retain semantically meaningful embeddings.

Individual glosses may be split up into several tokens. As a result of this, the tokens corresponding to a single gloss will have several associated embedding vectors. Prior to being inputted to the encoder, the mean of the embedding vectors corresponding to a single gloss is computed. Hence, we end up with a single embedding vector for each gloss. Additionally, the output layer is pruned such that weights that do not correspond to tokens in the vocabulary are removed.

As input to the encoder, the model receives a sequence of gloss-embeddings, computed in accordance to the description above. Following the conventions of mBART [18], a new language code is introduced for glosses "de_GLS" after the

<EOS> token. This serves to help the model distinguish between input modalities.

Additionally, label smoothing is applied for the targets in the objective function in alignment with the implementation of [5]. Hence, for a one-hot encoded target sequence $\mathbf{y} \in \mathbb{N}^{N \times K}$, the probability mass of the target word becomes $1 - s$ where $s < 1$ denotes the smoothing factor. For other words in the vocabulary, the probability mass is set to $\frac{s}{K}$ and for padding tokens the probability mass is 0. During training we minimize the Kullback-Leibler divergence (KL divergence) between the predicted sequence and smoothened target sequence. Prior to being inputted to the loss function, the sequences of probability distributions are flattened, meaning that the rows of the matrices $\mathbf{x}$ and $\mathbf{y}$ are stacked yielding vectors of dimension $\mathbb{R}^{N \cdot K}$. This can be precisely described as applying the matrix operator $\text{vec}^\top$[7].

The KL divergence can be described by

$$\mathcal{L}_{KL}(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^{N} \text{vec}^\top(\mathbf{y})_i \cdot \ln\Big(\frac{\text{vec}^\top(\mathbf{y})_i}{\text{vec}^\top(\mathbf{x})_i}\Big) \tag{16}$$

where N denotes the padded sequence length and $\mathbf{x} \in \mathbb{R}^{N \times K}$ $\mathbf{y} \in \mathbb{R}^{N \times K}$ are the prediction and target respectively.

The model is trained on the Gloss2Text task for 80 epochs, with a batch size of 8. The AdamW optimizer is used with an initial learning rate of 10^{-5}, a weight decay of 10^{-3} and a linear learning rate scheduler. As a learning objective, the KL divergence loss function is applied with a smoothing factor of 0.2. In order to prevent overfitting, a dropout rate of 0.3 along with an attention dropout of 0.1.

When evaluating model performance at each epoch, beam search decoding is used with a beam size of 4 and length penalty of 1. The weights of the model with the highest BLEU-4 score on the validation set are used for subsequent fine-tuning.

3.4 Sign2Text Task

This section covers bridging the modalities of video and text. Doing so allows for directly translating a sign language video into text and thus circumvents the need for a cascading approach. As mentioned in section 3.1, the entire proposed model, namely the visual component, language component and modality mapper is utilized for this task. Additionally, this section outlines the full translation pipeline as well as details surrounding its training.

3.4.1 VL-mapper

As noted earlier, visual cues are a factor when it comes to translating sign language to natural language. Therefore, information will likely be lost when a cascading approach is taken i.e. translating an input video into a corresponding gloss sequence and subsequently translating this into text. For this reason, we implement a Visual-Language mapper (VL-mapper) in alignment with [5]. It consists of two linear layers and ReLU activation in between. The VL-mapper takes gloss representations as

[7] Let $A = [a_{i,j}]$ be an $m \times n$ matrix, where $i = 1, \ldots, m$ and $j = 1, \ldots, n$.
Then $\text{vec}^\top A = \begin{bmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,n} & a_{2,1} & \ldots & a_{m,n} \end{bmatrix}^\top \in M_{mn}$.

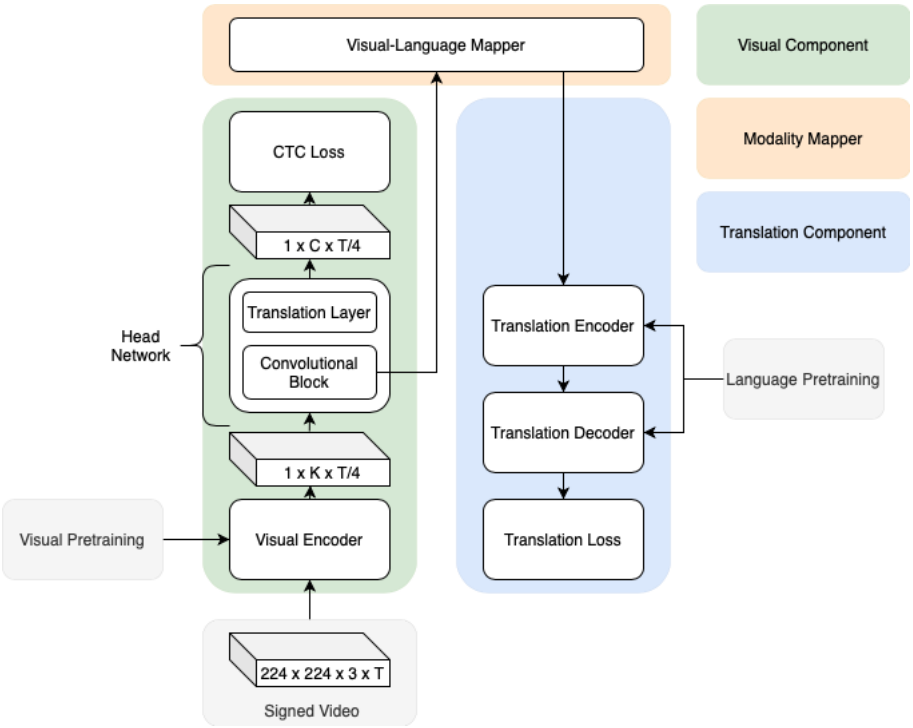


Figure 17: Overview of the full proposed Sign2Text pipeline. For simplicity the three components of the model are decoupled into the visual component, the translation component and the modality component connecting the two modalities. Both the visual encoder and translation encoder-decoder are pretrained as described in earlier sections.

input i.e. latent features of dimension $\mathbf{z} \in \mathbb{R}^{T/4 \times 512}$. That is, the input to the VL-mapper is the output of the convolutional block in the head network (see figure 7a). The first layer projects the features to dimension $d_{model} = 1024$, and the last layer does not change the dimensionality. This effectively corresponds to learning a new embedding matrix for the visual domain as opposed to tuning the one of mBART trained to encode the semantics of text input.

3.4.2 Full Model Pipeline

With the addition of the VL-mapper, we are able to directly translate a sign language video into text. First, the video is fed through the S3D backbone yielding a latent representation $\mathbf{z} \in \mathbb{R}^{T/4 \times 832}$. This is then passed through the convolutional block of the head network changing the latent dimension to 512. Hereafter, the VL-mapper converts this dimension to 1024, thereby matching d_{model} of mBART. Lastly, the embedded sign language video is passed through the mBART pipeline yielding a probability distribution $\mathcal{P} \in \mathbb{R}^{N \times K}$ where N is the length of the outputted sequence and K is the size of the vocabulary. This sequence is then decoded into a natural language translation through beam search decoding. An illustration of the end-to-end model pipeline can be seen figure 17.

3.4.3 Implementation Details - PHOENIX-2014T

The model is trained on the Sign2Text task for 40 epochs with a batch size of 6. AdamW is used as the optimizer with a weight decay factor of 10^{-3}. For the visual component of the model as well as the VL-mapper, the initial learning rate is set to 10^{-3} whereas the language component's learning rate is initialized as 10^{-5}.

Additionally, a cosine annealing learning rate scheduler is used with $T_{max} = n_{epochs}$. The augmentations used during training and validation remain identical to the ones used for Sign2Gloss pretraining as described in section 3.2.6. During training, the weights of the S3D backbone are frozen to alleviate computational overhead. Thus, the head network, VL-mapper and weights of mBART constitute the trainable parameters. For mBART, the dropout is set to 0.3 along with an attention dropout of 0.1. For the head network, all dropout layers are set to 0.1 as well. The head network is supervised by CTC loss and the objective function of mBART is KL divergence with label smoothing, as described in section 3.3.4. Each of the two loss terms are weighted equally during training.

3.5 Evaluation Metrics

In the following, we introduce the various evaluation metrics used to assess the effectiveness of the model at different training stages.

3.5.1 Word Error Rate

For assessing the quality of gloss prediction, Word Error Rate (WER) is used. This metric computes the number of insertions, deletions or substitutions necessary to reproduce a reference sentence from an input sentence divided by the length of the reference. Hence, given a sequence of input and reference words x and y, respectively, the WER is given as

$$\mathrm{WER}(x, y) = \frac{\mathrm{Ins}(x, y) + \mathrm{Sub}(x, y) + \mathrm{Del}(x, y)}{|y|}$$

3.5.2 ROUGE

The ROUGE-N metric introduced by [17] computes the number of overlapping n-grams between an input and a reference sentence.

ROUGE-L improved on this by considering the longest common subsequence (LCS) between the input and reference. Formally, the LCS between two sequences $\mathbf{x} \in \mathbb{N}^N$ and $\mathbf{y} \in \mathbb{N}^M$ is the longest sequence of strictly increasing indices $\mathbf{I} \in \mathbb{N}^K$ where $\mathbf{x}_i = \mathbf{y}_i \;\; \forall_i \in \mathbf{I}$. Thus, the ROUGE-L measure becomes

$$\mathit{ROUGE\text{-}L}(\mathbf{x}, \mathbf{y}) = \frac{(1 + \beta^2) \cdot R_{LCS}(\mathbf{x}, \mathbf{y}) \cdot P_{LCS}(\mathbf{x}, \mathbf{y})}{R_{LCS}(\mathbf{x}, \mathbf{y}) + \beta^2 \cdot P_{LCS}(\mathbf{x}, \mathbf{y})}$$

where $\beta = \frac{P_{LCS}}{R_{LCS}}$ and R_{LCS}, P_{LCS} are defined as standard recall and precision measures, respectively, taking the LCS into account. We can express this as

$$R_{LCS}(\mathbf{x}, \mathbf{y}) = \frac{LCS(\mathbf{x}, \mathbf{y})}{|\mathbf{y}|} \qquad P_{LCS}(\mathbf{x}, \mathbf{y}) = \frac{LCS(\mathbf{x}, \mathbf{y})}{|\mathbf{x}|}$$

3.5.3 BLEU

The BLEU metric [22] is a popular choice amongst NLP practitioners for evaluating the corrected of predicted sequences against references. As shown in [22], the metric has a high correlation with human evaluations in the context of machine translation.

Recall the standard precision measure which, in the context of translation, is given as the number of unigrams in a predicted sentence that are also present in the reference sentence divided by the number of words in the predicted sentence. This measure can result in high precision scores in spite of the translation being poor e.g. cases where $n = 1$ and the prediction contains a single word in the reference duplicated an arbitrary amount of times. To this end, the authors introduce a modified n-gram precision which for a prediction p and reference r is defined as:

$$MP_n(p, r) = \frac{\sum_{n\text{-gram} \in p} \min(count_p(n\text{-gram}),\ count_r(n\text{-gram}))}{\sum_{n\text{-gram}' \in p} count_p(n\text{-gram}')}$$

Hence, multiple occurrences of the same n-gram in a prediction will only be counted the same amount of times that it occurs in the reference.

Additionally, the authors introduce a brevity penality (BP) to combat cases where the prediction is shorter than the reference. [8] Given a predicted sentence p and its corresponds reference r, the brevity penalty is defined as

$$BP(p, r) = \begin{cases} 1 & \text{if } |p| \geq |r| \\ \mathrm{e}^{\frac{1-|r|}{|p|}} & \text{else} \end{cases}$$

To combine modified n-gram precisions for different values of n, the authors propose computing the geometric mean. However, the modified n-gram precision decays approximately exponentially with a factor of n. To accommodate this, the logarithm of modified n-gram precision is computed. With the above definitions in mind, the BLEU metric is given as

$$BLEU(p, r, N) = BP(p, r) \cdot \exp\left(\sum_{n=1}^{N} \frac{1}{N} \ln(MP_n(p, r))\right)$$

Note that the exponentiated term is the geometric mean as expressed in log-scale:

$$\mu_{geo}(A) = \underbrace{\left(\prod_{i=1}^{n} a_i\right)^{\frac{1}{n}}}_{\text{Std. } \mu_{geo} \text{ def.}} = \exp\left(\frac{1}{n}\sum_{i=1}^{n} \ln(a_i)\right)$$

for the list of numbers $A = a_1, a_2, \ldots, a_n$.

3.6 Omitting Glosses - Ablation Study

As stated earlier in the context of How2Sign, collecting gloss annotations is heavily time-consuming. To address this, we decided to conduct a preliminary ablation study

[8] Note that the reverse i.e. where the prediction is longer than the reference case is already penalized by modified n-gram precision.

to assess the effect of omitting expertly annotated gloss. Hence, we seek to assess whether learning a temporally aligned latent representation of sign language videos is feasible in a weakly-supervised manner. In doing so, it would be possible to leverage massive SLT datasets containing only videos and natural language translations e.g. [6] and [25].

3.6.1 Statistical Gloss Approximation

To obtain gloss annotations, we approximate the ground truth glosses using linguistic rules of DGS and German. Due to lack of expertise regarding DGS and German we resort to the proposed method by [20]. Following the outlined approach, SpaCy is used for extracting relevant information about the natural language translation. We wish to identify the part-of-speech for each word, detect named entities, determine the grammatical structure of the sentence and lemmatize all words. The approach of approximating glosses is summarized in algorithm 2 below.

Algorithm 2 Gloss approximation procedure

Goal: Approximate ground truth gloss annotations
Let s denote a sentence represented as a list of words.
Let m denote a NLP model capable of extracting linguistic information from sentences.
Let **Swap**(s, w_1, w_2) denote a function that swaps the position of words w_1 and w_2 in a sentence s.
Let **ExtractSVO**(s) denote a function capable of extracting a list of (subject, verb, object) tuples from a sentence s.

```
SyntheticGlosses(s, m)
  s' ← m(s) // extract relevant information
  SVO ← ExtractSVO(s')
  if length (SVO) > 0
    for (S, V, O) in SVO
      s ← Swap(s, V, O) // Swap VERB and OBJECT
  for word in s'
    If word.POS ∉ {NOUN, VERB, ADJECTIVE, ADVERB, NUMERAL}
      s.remove(word.text) // Remove word from s
    if word.NER = LOCATION
      s.remove(word.text)
      s.prepend(word.lemma) // Move to start of s
    if word.DEP = NEGATION
      s.remove(word.text)
      s.append(word.lemma) Move to end of s
    else
      s.replace(word.text, word.lemma)
  return s
```

The procedure highlighted in algorithm 2 is performed for all natural language trans-

lations for the train, test and validation set, respectively. To assess the error associated with approximating glosses as opposed to using the ground-truth annotations, the WER between them is reported in table 5 below.

WER		
Train	Dev	Test
87.26	87.64	85.65

Table 5: WER between the ground truth gloss annotations and those obtained by running algorithm 2 for the different data splits on PHOENIX-2014T.

As seen in table 5, there is a significant error associated with the proposed method. Generating synthetic glosses leaves us with a gloss vocabulary size of 2391 excluding the blank token added for CTC training. Thus, it is reasonable to assume that the approximated glosses will contain more redundant words as well as inconsistencies compared to the gold-standard annotations. Upon inspection, there are some differences in the lemmatization of words computed by SpaCy compared to the ground-truth glosses. Thus, the same signs may have different yet consistent annotations when comparing between the ground truth and approximated glosses. We account for this, to a small extent by converting cardinal directions into single universal labels e.g. norden, nördlich and other variants are always mapped to nord (north in German). Both this and filtering for stop words is applied post-hoc to combat inconsistencies as well as redundancy within annotations.

4 Software Product

In order to enable users to interact with our model, we need to integrate it into a user application, and provide a user interface. Our design choices for this application reflect a wish for modularity and decoupling of the different components, but also resource constraints, since we are developing with the end goal of running it on consumer-level hardware. Both our ML model pipeline and the modalities of the data it operates on, i.e. video and audio, are quite resource intensive in nature. The resource usage is not lessened by the fact that we need to feed our model the video data in a format resembling the `rawvideo` format, i.e. as an uncompressed, multidimensional tensor.

To facilitate a satisfactory user experience, we will thus need to have a simple user interface, an efficient video processing pipeline and finally our model needs to work sufficiently fast.

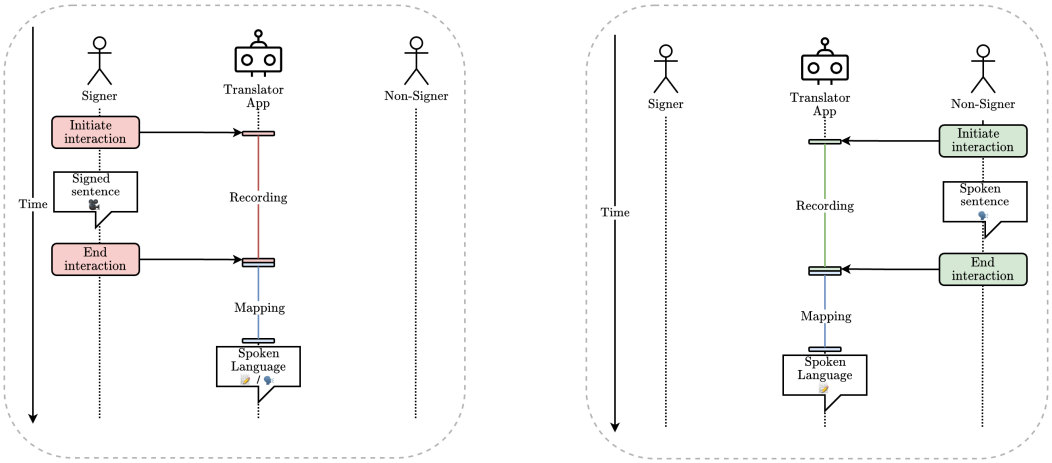
4.1 Problem Formulations

4.1.1 Overall Problem

Our product needs to enable conversation between a person understanding sign language and text and a person understanding spoken language and text.

The core problem can be broken down into translating between a sign language and a spoken language, mapping between different modalities of spoken language and optionally, translating between spoken languages. We have chosen to design our software product primarily around the first of these tasks, i.e. the translation of sign language to spoken language, as it is of course our own model that we have built and trained.

Our model works in a way that limits its interface to take in full chunks of video only, which means that the only way to get meaningful output is to perform inference on the entirety of a signed sentence. This has led us to model general conversations between a signer and a non-signer in our app as a sequence of interactions using protocols as shown in figure 18.



(a) Protocol for when the signer is signing. Note that the output spoken language sentence is read aloud.

(b) Protocol for when the non-signer is speaking. Note that the output spoken language sentence is only displayed.

Figure 18: Conversation protocols for the application.

In these protocols, both the spoken and signed sentences are recorded in chunks, which the users specifies when pressing start and stop. This means that the conversation flow will consist of one person communicating a sentence and only after this will the processing begin.

4.1.2 Subproblems

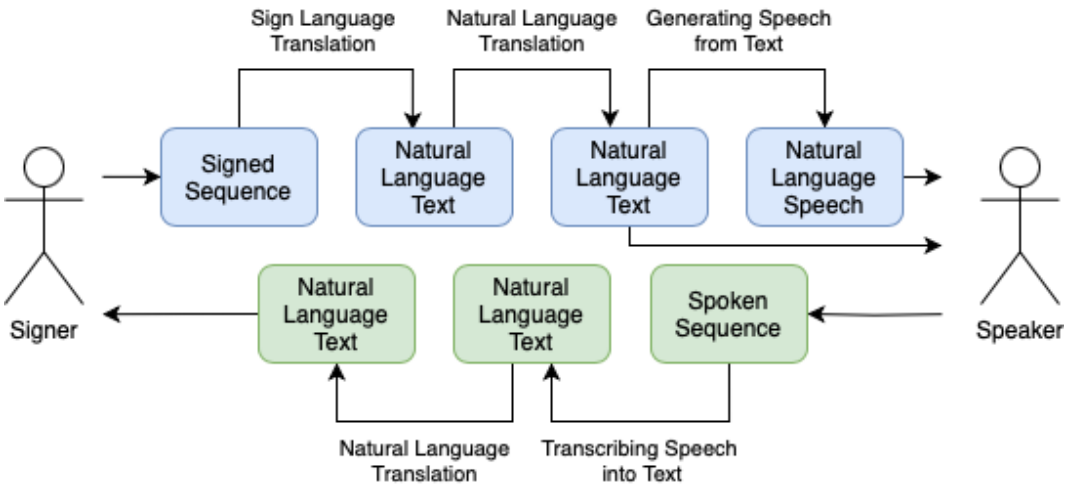


Figure 19: High level overview of the application translation functionality. Note that natural language translation as well as text-to-speech are optional.

In figure 19, a high level overview of the application translation functionalities are visualized. Following these functionalities, we have broken the core problem described above down into the following components:

1. Translate sign language videos into text.

2. Generate speech from text.
3. Transcribe speech into text.
4. Translate text between languages.
5. Ensure inference execution time is sufficiently close to real-time.

Our design take these subproblems into account as follows; 1. should be handled by our SLT model. 2. should be handled by WaveNet [27], an external API provided by Google. 3. should be handled by a local instance of OpenAI Whisper [24]. 4. should similarly to 2, be handled by an external API, provided by Google. 5. is not currently formulated as concretely as the first three specifications, but we will get more into that. It will also serve as a useful evaluation metric for the full product in terms of usability. Furthermore, it is a parameter that can be used to trade off translation accuracy with efficiency.

4.2 The Real-Time Problem

Before going into the solutions to the subproblems, we will attempt to give a precise definition of what 'real-time' performance means in the context of our application.

Within computing, a real-time process or operation usually refers to there being a guarantee of completion within a certain time. This deadline is highly domain specific, and will be defined in the context of our application. Recall that the core aim of our product is to enable conversations between signers and non-signers. The 'protocols' for such a conversation is illustrated in figure 18.

Our application performs the translation between languages and mapping between modalities needed to facilitate the conversations. On a high level, it does so by letting the users record the required input data and then processing the input to fit the different interfaces of the different models, and then runs inference and present the results to the users. In terms of what can be deemed as real-time in this context, the ideal to strive for intuitively corresponds to having a regular conversation.

This is optimistic, however, since the model is unable to perform online, continuous translation, but needs the full context to begin inference. Thus, we set a milder constraint, and view the role of the app to be that of an translator or interpreter. By doing so, we allow for the full sentence to be communicated before we start the timer, so to speak. In other words, latency is only counted from when the user signals to the application that they are finished with a sentence.

With this definition of latency in mind, we define true real-time performance for the role of interpreter as less than ~ 1 second of latency. We set the following definitions in order to clearly state our performance metrics as well as our target:

App Latency :	Time measured between when a language sequence is finished and the translation result is presented.

Real-time Performance :	Less than ~ 1 second of **App Latency**.

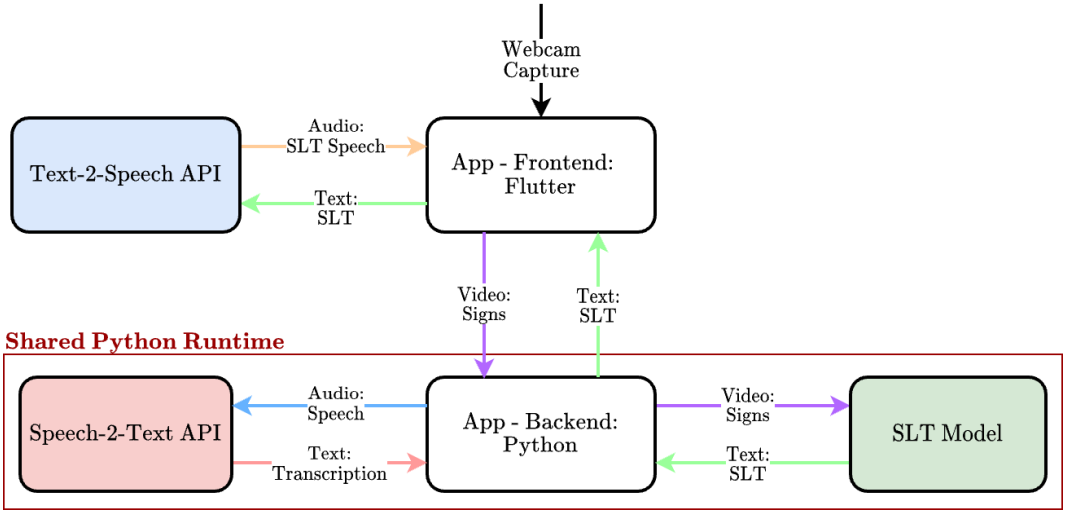


Figure 20: High level architecture of the software product.

4.3 Backend Architecture

For our application, we choose an architecture with a clear split between user application logic and data processing logic.

This allows for more flexibility in choosing the technologies for the user application, and also allows for hosting the model and data processing sever separately. This could be relevant in case we want to interface with the model via a device that cannot run inference on its own, such as a mobile device.

Our focus however, will be on running the model and data processing application on the same machine as the user application, in this case an Apple MacBook Pro 2021, as an example of consumer grade hardware. Additionally, to test the application when using hardware acceleration, we also conduct tests on a separate system equipped with an RTX-3060 GPU.

Given this constraint and the fact that a design goal of our application is to reach true or close to real-time performance (previously defined as $<\sim$ 1 second of latency), the architecture needs to be designed around efficiency and performance. From pilot tests of the technologies which we will describe below, we determined model inference to be the biggest bottleneck in terms of latency. We do not have many parameters to control inference speed - in fact, since the spatial resolution of the input data is fixed at the dimensions demanded by our visual encoder network, we can only tweak the temporal resolution for this purpose. We will investigate the implications of this later on in this report.

The user application module (client) serves as the interface to the user. From the user application, video and audio are recorded via the device's built-in camera and microphone. These collections of data are then transferred via WebSocket to the data processing and model inference module (model server). The client then receives responses back containing the text predictions with corresponding synthesized audio or transcriptions of the streamed video and audio, respectively. This architecture is visualized as a high level diagram in figure 20.

The model server is packaged as a server application, which exposes an API, which can be consumed via the user application. The API has REST and WebSocket endpoints, which are used for different purposes.

4.3.1 REST

REST is an acronym for **RE**presentional **S**tate **T**ransfer, and the concrete implementation in our case is through HTTP. The user application sends HTTP requests for simple configuration changes on the server-side, such as changing target or source language for the translation functions.

4.3.2 WebSocket

WebSocket is a communications protocol that enables duplex channels over a single TCP connection, i.e. two-way communication. The WebSocket protocol has lower data transfer overhead than HTTP, and is in fact designed to facilitate real time data transfer between client and server.

4.4 Efficient Pipeline for Video Streaming and Processing

In order to improve the real-time aspect of the application, we need to take into account both the computationally expensive work as well as the latency caused by transferring data via network protocols. Figure 20 shows an overview of the full application inference pipeline, which will be described in detail in the following.

In accordance with our aim to build a web application-based client in Google Chrome, we utilize that the video recorded via webcam is stored in the `.webm`-format, which is already compressed via video codec VP8 or VP9, and audio codec Opus. This means that it will be substantially less expensive to transfer to our model server.

We stream the serialized compressed video data from the application to our model server via a WebSocket. The model server receives the data and spawns a subprocess running the multimedia utility library FFMPEG [26], which it communicates with using inter-process communication (IPC) pipes. FFMPEG converts the `.webm` video into `rawvideo`, which we are then able to load directly into a tensor object in the Python runtime.

Now that we have the video stored in memory as a tensor, we can perform a series of video processing steps to ensure the tensor has correct dimensions, value ranges and format for our ML model. This ensures that the model is able to handle the inputs. Additionally, through this novel inputs will resemble the model's training data as much as possible.

The video processing steps are modularized into the `VideoPipeline` class. An overview of this procedure is given in figure 21 below. Moreover, an overview of the entire workflow described above is visualized in figure 22.

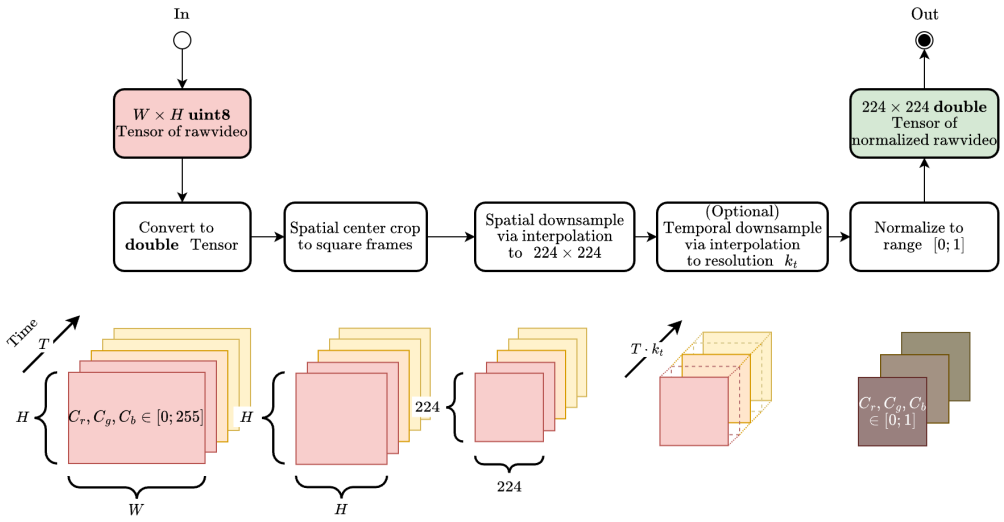


Figure 21: Video processing steps for converting from input dimensions into the parameters specified by our model's API. Implemented in the modular custom class VideoPipeline.

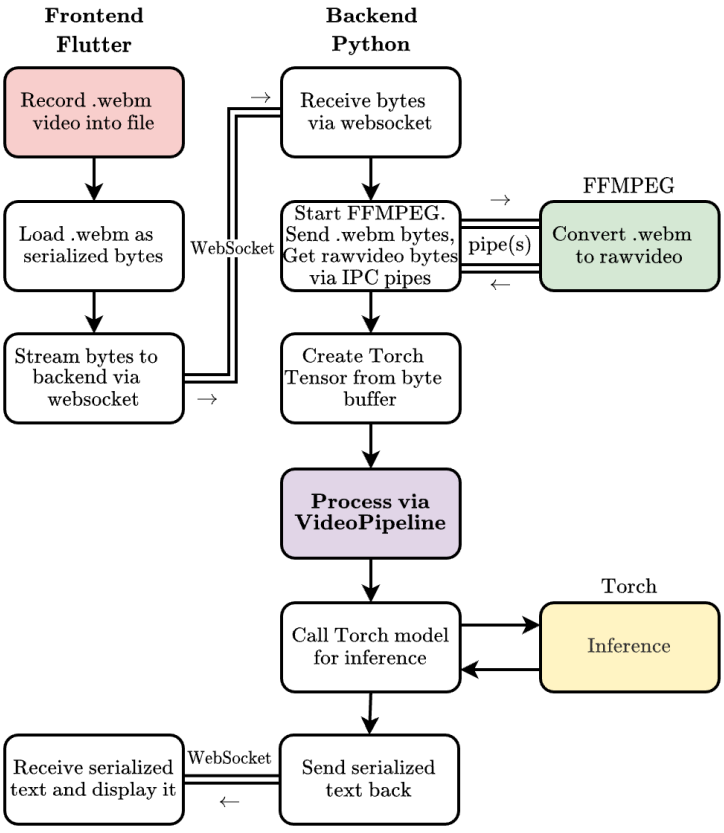


Figure 22: Overview of the workflow for recording, streaming and processing video, running SLT inference and then receiving the inferred translation.

4.4.1 Temporal Downsampling as a Processing Step

A way that we can tune the execution time of model inference is to apply a temporal downsample. Hence, we can interpolate the frame sequence along the time axis of the input. This effectively corresponds to lowering the temporal resolution, potentially improving the inference time of the model, as it only has to process a shorter frame sequence.

Obviously, lowering the temporal resolution results in less information that our model can use. [5] performed an ablation study to assess exactly this. They trained each part of the model pipeline of the videos with specific downsampling rates. From the results of [5], as well as our own experiments, which will be introduced later, it is evident that the model performance degrades notably when temporal downsampling is applied. That being said, the gain associated with faster inference time may be desirable in some use-cases. Observations made in this context, which will be presented and discussed later, serve as motivation for incorporating an option for users to trade-off between inference time and performance.

To this end, we extend our `VideoPipeline`-class with an optional temporal downsampling step. Concretely, it applies interpolation along the temporal dimension of the video tensor, separately for each of the three color channels. We will get into two interpolation modes, linear and nearest, i.e. picking the frame from the old tensor nearest to the new index along the temporal axis.

In our frontend application, we implement a slider widget that allows the end user to interpolate between 3 performance modes, corresponding to 3 levels of temporal resolution (denoted k_t), 1, $\frac{1}{2}$ and $\frac{1}{3}$ of the original resolution, respectively. These are the same levels as [5] used for their ablation study.

4.5 Model Integrations

As mentioned in section 4.1.2, the core components of our application, besides translating sign language into text, is generating speech from text, transcribing speech into text and lastly enabling cross language communication through text to text language translation. Due to the scope of the project, we chose to integrate these three components using established pre-existing models.

Figures 23 and 24 shows a high level overview of these model integrations and how they are called in sequence, when translating a signed and spoken sequence, respectively.

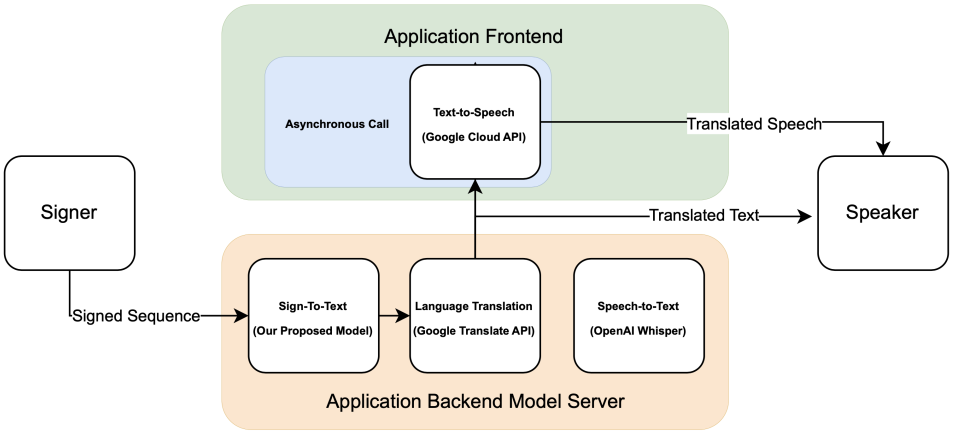


Figure 23: High level overview of the sequential calls of the model integrations, when translating a signed sequence.

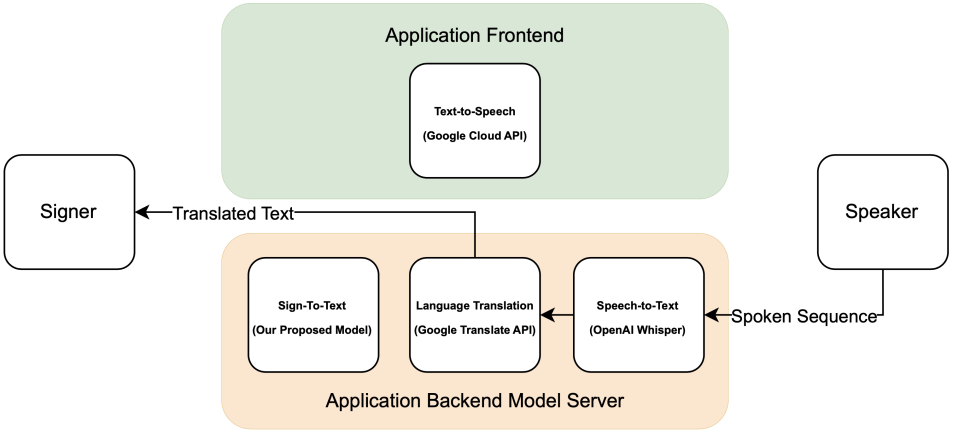


Figure 24: High level overview of the sequential calls of the model integrations, when translating a spoken sequence.

4.5.1 Google Cloud Text-to-Speech API

The text to speech component was implemented through the Google Cloud Text-to-speech API, using the Wavenet engine [27]. This integration was chosen, due to it's highly natural sounding capabilities and low latency. Furthermore, as the component is integrated through an external API, we omit depleting the limited resources of the consumer hardware.

As visualized in figure 20, the text-to-speech API calls were implemented in the application frontend, in contrast to all other signal processing components of the application. This placement was chosen as a means to avoid any unnecessary latency, when translating sign language in the backend model server of the application. Thus, the user is presented with the sign-to-text translation, as soon as the application receives the response from the backend model server. Following this, the frontend calls the Text-to-Speech API asynchronously through an HTTP request and plays the received audio response.

4.5.2 Google Translate API

Similarly, the Google Translate API is used to accommodate language differences amongst end users. This was implemented due to its efficiency as well as to reduce the impact on the end-user hardware. As the language translation is necessary to present the end-user with a response, this component was implemented in the backend model server, along with the other signal processing components.

4.5.3 OpenAI Whisper

The last integrated pre-trained component, the speech-to-text transcription, was implemented using a locally run instance of the multilingual OpenAI Whisper Speech-to-Text base-model. In contrast to the previously mentioned components, where external API connections where implemented, this component is run locally on the backend model server. By running the model locally we avoid the latency in relation to calling external API connections. Furthermore, as this variant of the Whisper model is relatively efficient and does not require a considerable amount of computational power, this implementation was found to be the most suitable for low latency transcription.

4.5.4 Our Proposed Sign2Text Model

Similarly, our proposed Sign2Text model was implemented as an instance run locally on the backend model server to avoid the latency in relation to calling external API connections. However, as this model is relatively large and expensive to run, this implementation has a significant impact on both the real-time aspect of the application, along with the end-user hardware requirements. These implications will be discussed further in later sections.

4.6 Application Architecture

The main objectives with regards to the architecture of the application is simplicity, efficiency and cross-platform capabilities. As the end-user of the application does not necessarily have any particular experience with advanced software interfaces, it is essential for the application interface to be simple and intuitive. Moreover, in alignment with our desire to reduce the latency of the backend model server, we also aim to minimize the amount of time the end-users spend accessing the translation interface. Thus, the translation setup should be simple. Lastly, in order to make the application accessible to as many potential end-users as possible, cross-platform capabilities are crucial. The above mentioned objectives are at the core of the application architecture and the following design choices where made to satisfy these.

4.6.1 Flutter Development Kit

Flutter is an open-source UI software development kit created by Google. One of the key capabilities and advantages of Flutter is the ability to develop cross-platform

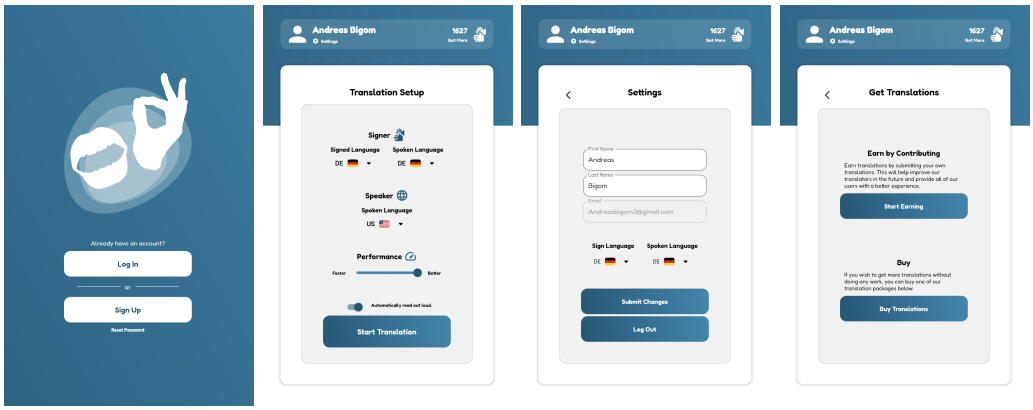
applications from a single code base. Thus, by writing the application using the Flutter development kit, only minor changes have to be implemented throughout the code base, in order to reach the end-users on platforms such as Android, iOS, Linux, macOS, Windows, and the web.

Furthermore, Flutter has a large set of pre-written packages available for common integrations, such as accessing the on-device media-recorders, connecting to databases and performing web requests. Integrating these packages reduces the workload of the application design process.

4.6.2 Firebase Services

Firebase is a collection of backend cloud computing services such as NoSQL databases and user-authentication, provided by Google. As Firebase and Flutter are both created by Google, the integration of Firebase in the Flutter application is automatic and well supported. Furthermore, the Firebase Cloud Firestore NoSQL database provides fast and reliable real-time services.

For these reasons, the Firebase Authentication and Firebase Cloud Firestore services were chosen as the user-authentication and backend database for the application architecture, respectively.



(a) Welcome Screen. **(b)** Main Menu. **(c)** Settings. **(d)** Get Translations.

Figure 25: Application interface design for configurational screens.

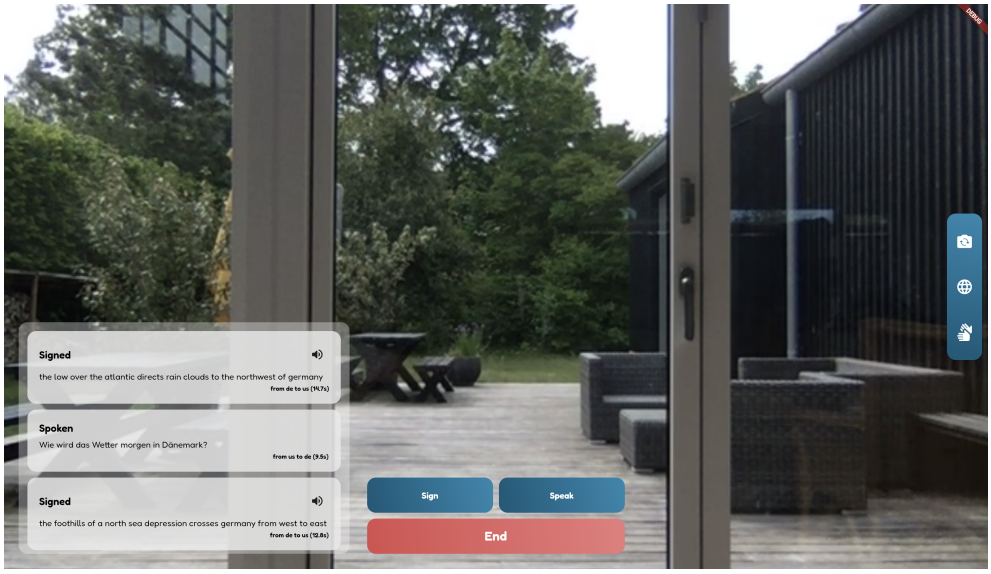


Figure 26: Application interface design for two-way communication / translation screen.

4.6.3 Application Interface

Lastly, but equally important, the application interface was designed to meet the objectives of the application architecture. Firstly, the application only consists of five simple screens as shown in figure 25 and 26. 1) The welcome screen, where users can create an account or log on to an existing account (figure 25a). 2) The main screen, where users can configure the settings of and begin a translation (figure 25b). 3) The settings, where users are able to change their default settings (figure 25c). 4) The Get Translations screen, where users are able to get more translations through contribution or pruchases (figure 25d). 5) The translation screen, where users can perform two-way communication between signers and non-signers (figure 26).

Furthermore, as the translation configuration screen is used as the main screen, we minimize the time spend by the user to configure and begin a translation. Additionally, the default translation configuration, such as the specific sign-language and spoken language, can be set by the user. Following this, the application will use these as the default configurations when initializing a translation. Thus, the user is able to begin a translation rapidly with a few to no steps from opening the application.

4.7 Performance Analysis

In this section, we will go through the experiments and analyses we wish to conduct, in order to find performance bottlenecks and possible ways of mitigating these issues.

4.7.1 Benchmarking the Application

In order to quantitatively evaluate the performance of the application, we perform rigorous tests of the various computationally intensive operations performed. The main focus in this context is the execution time and total latency induced. To ensure fair comparisons across tests and hardware specifications, the same 100 samples,

which were initially selected at random from the PHOENIX-2014T test-split, are used throughout benchmarking. We develop a few different testing suites that monitor the execution time of each subcomponent of the application pipeline.

First and foremost, we will be using various built-in Python timer instruments to perform the actual measurements. We use and compare the metrics `time.time`, `time.perf_counter`, `time.process_time`. `time.time` simply uses the system Real-Time Clock to get timestamps. `time.perf_counter` uses the value of a performance counter, i.e. the highest available time resolution to measure a short duration. The value of a single such measurement is machine-specific, and only the deltas between two measurements are informative for us. `time.process_time` returns the CPU time of the current process, and pauses for any I/O calls and similar. `time.process_time` is not optimal for measuring Pytorch execution times, as it does not reset its counter when running the test suite. This might be because Pytorch reuses the C++ backend process that performs the computations. Because of the unreliable results of `time.process_time` for Pytorch, we will mainly be using it to measure the latency caused by waiting for asynchronous calls to the external Google Translate API. The test suites are structured as follows:

1. `test_suite_slt`
    - Convert `.webm` bytes to video tensor.
    - **[GPU only]** Transfer video tensor from RAM to GPU VRAM.
    - Process video tensor by running it through `VideoPipeline`.
    - Run Sign2Text inference on processed video tensor.
    - Call external Google Translate API to translate the output sentence.
2. `test_suite_stt`
    - Convert `.webm` bytes to waveform.
    - Run Whisper Speech2Text inference on waveform.
    - Call external Google Translate API to translate the output sentence.

Note that Text2Speech inference is not included in the SLT performance test. This component was excluded, as the Text2Speech API is called in parallel with presenting the end-user with the text-based result. Thus, the true inference latency is not affected by this step. Each step of the test suites are wrapped in a performance monitoring function, elaborated in algorithm 3 below.

Algorithm 3 Performance monitor wrapper function for components.

Input: *component*: function, *component_args*: dynamic // ***function to be tested, and its (variable) input arguments.***
Result: (*perf_metrics*: DataFrame) is extended with performance data.

Let *res* be a variable for storing function (*component*) output.
Let *start*, *end* and *metrics* be auxiliary variables for storing the three different time measurements.
Let **StartTimer**(), **EndTimer**() and **ComputeMetrics**() be auxiliary functions for getting start times, end times and computing deltas, respectively.

$start \leftarrow$ **StartTimer**()
$res \leftarrow component$.**call**($component\_args$)
$end \leftarrow$ **EndTimer**()
$metrics \leftarrow$ **ComputeMetrics**($start, end$)

append($perf\_metrics, metrics$) // ***Append computed metrics to DataFrame.***

We perform all aforementioned experiments on CPU, in order to validate how our application performs on average consumer level hardware. Furthermore, in order to test how our application performs using hardware acceleration, we perform identical experiments on a system with a discrete GPU.

Results of the experiments are presented in section 5.2.

4.7.2 Analyzing the Effect of Temporal Downsampling

To allow for comparisons between the effect of temporal downsampling and other benchmarks on execution time, the same samples described in section 4.7.1 are used, along with identical procedures in terms of timing. Furthermore, latency is assessed on the same examples of consumer-grade hardware as highlighted in section 4.3. Additionally, to draw comparisons between the ablation study conducted by [5] surrounding temporal downsampling, the same downsampling factors are used i.e. $\frac{1}{2}$ and $\frac{1}{3}$.

4.7.3 Modeling and Estimation of Execution Time

When analyzing the results of the measurements from running the test suites, we obtain a lot of samples of execution times when inputting a different number of frames. We thus want to use a simple statistical model to get an estimate of the relative contributions to execution time for each component of the app pipeline.

The two pipelines, for Sign2Text and Speech2Text respectively, both consist of a some theoretically simple components in terms of their runtime relative to input size, as well as some much more complex ones, namely our Pytorch model and OpenAI Whisper.

One approach to estimate runtimes would be to calculate theoretical runtime complexities in terms of $\mathcal{O}(n)$ for our model. The layers and other parameters of the model are all fixed at inference time.

The variability in execution time comes from the input size to the convolutional layers, the CTC decoding operation, the subsequent input size to the transformer, the autoregressive output sequence generation of the transformer and finally, the beam search decoding operation on the logits of the transformer decoder.

In addition to the variability in the models themselves, the hardware and compilers used to create and run them also plays a large role in the actual run time. The compiler might have vectorized some of the basic operations used, and some higher-level operations, such as self-attention[9] in the transformer, is massively parallellizable when using a GPU.

All of these factors make it very complex to create a precise, general estimator, so this approach is out of scope for this project.

Instead, we focus on an easily interpretable approach, and use linear regression on the measurements, removing any extreme outliers. An example of regression applied to the execution times of the Sign2Text pipeline on the GPU can be seen in figure 31 in section 8.3 in the appendix.

[9]Specifically, the matrix multiplication operations in dot-product attention.

5 Results

This section presents our model performance at different training stages along with results of the ablation study. The reported benchmarks on model performance are based on a single training due to the required computational overhead. We acknowledge that training should ideally be conducted several times with the same configuration to account for deviations. Moreover, the effect on performance when temporally downsampling input videos is assessed. We also assess the generalizability of the Sign2Text model by testing performance on never before-seen samples drawn from the DGS corpus.

Additionally, benchmarks on the runtime for Sign2Text as well as Speech2Text are shown with varying input lengths and configurations. Lastly, the contributions of different components to the app latency are highlighted. All results surrounding latency are, unless otherwise specified, run on a 2021 Apple MacBook Pro with an M1 Pro processor. GPU tests are performed using a Nvidia RTX 3060 GPU and an Intel Core i7-10700 CPU.

5.1 Model Performance

Here the obtained results surrounding model performance are outlined and briefly described.

5.1.1 Sign2Gloss

Below, various benchmarks on the test set of WLASL are presented in terms of top-k accuracy.

Method	top-1	top-5	top-10
VGG-GRU [15]	8.44	23.58	32.58
Pose-TGCN [15]	23.65	51.75	62.24
Pose-GRU [15]	22.54	49.81	61.38
I3D [15]	32.48	57.31	66.31
S3D (ours)	30.16	60.39	70.95
NLA-SLR [32]	**61.26**	**90.91**	N/A

Table 6: Top-k accuracy of different models on the test set of WLASL [15].

Below, comparisons between the state-of-the-art (SOTA) model [5] and our results following Sign2Gloss training are made on the validation and test sets of PHOENIX-2014T.

Initialization		WER val		WER test	
Kinetics	WLASL	Ours	Org. author	Ours	Org. author
		N/A	27.25	N/A	28.06
✓		**22.99**	23.05	23.64	**23.50**
✓	✓	33.43	**21.90**	33.88	**22.45**

Table 7: Performance benchmarks on the Sign2Gloss task for PHOENIX-2014T [2].

As evident from the results in table 6, our model performance on WLASL is on par with other benchmarks, aside from NLA-SLR [32]. This paper, however, is the current state-of-the-art method on several sign language recognition datasets, including WLASL. In spite of the relatively competitive performance obtained by our model on the test set of WLASL, the pre-training by SOTA [5] presumably yielded better results. This is backed up by the unsatisfactory results when further training our model on the Sign2Gloss task using our WLASL weights as initialization (see table 7). Surprisingly, our model with WLASL initialization achieves a higher WER compared to the 28.06 WER obtained by SOTA when training from scratch.

That being said, our results are on par with SOTA when initializing the S3D backbone with weights trained on the Kinetics-400 dataset. Thus, the correctness of the implementation is still verified in spite of the unsuccessful pre-training when initializing the model with weights trained on WLASL. This is in alignment with the findings of SOTA that weight initialization has a large effect on model performance. To verify the correctness of the pipeline during Sign2Text training, the weights obtained during Sign2Gloss training with Kinetics-400 initialization are used. This allows us to still make direct performance comparisons with SOTA.

5.1.2 Gloss2Text

Here, the results after training our model on the Gloss2Text task are reported and compared to SOTA [5]. For the validation set, we test the model performance on beam sizes varying between [1, ..., 10] and report the results where the highest BLEU-4 score is observed. On the test set, the same number of beams are used as for the reported validation results.

Validation					
Method	**ROUGE-L**	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**
Org. authors	53.79	**54.01**	**41.41**	**33.50**	**28.19**
Ours	**55.00**	51.34	40.28	32.86	27.65
Test					
Method	**ROUGE-L**	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**
Org. authors	52.54	**52.65**	**39.99**	**32.07**	**26.70**
Ours	**52.91**	49.58	38.77	31.21	25.94

Table 8: Performance on the Gloss2Text task for SOTA [5] and our implementation. Results are reported for the test set of PHOENIX-2014T.

As shown in table 8, our obtained results when performing Gloss2Text pretraining are slightly lower compared to SOTA. The only exception to this is the ROUGE-L metric with which we obtain a slightly better performance. However, our results are reported for a single training session and as such the slight performance differences could be attributed to natural training variability.

5.1.3 Sign2Text

Below our results on the validation and test sets of PHOENIX-2014T [2] after Sign2Text training are presented along with the results of the SOTA [5] model. The same procedure for determining the appropriate beam size in Gloss2Text is used here.

Validation					
Method	**ROUGE-L**	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**
Org. author	53.10	53.95	41.12	33.14	27.61
Org. author w/o WLASL	53.24	53.99	41.47	33.63	28.19
Ours	51.84	48.59	37.91	30.61	25.44
Test					
Method	**ROUGE-L**	**BLEU-1**	**BLEU-2**	**BLEU-3**	**BLEU-4**
Org. author	**52.65**	**53.97**	**42.01**	**33.90**	**28.39**
Org. author w/o WLASL	52.42	53.66	41.27	33.36	27.91
Ours	52.07	50.24	39.24	31.78	26.52

Table 9: Benchmark performances on the Sign2Text task for PHOENIX-2014T [2].

As evident from table 9, the performance of our Sign2Text model is slightly lower compared to SOTA. This could perhaps be attributed to differences witihn the Gloss2Text training degrading our model performance. However, this is likely not the sole cause considering that the SOTA model achieves a BLEU-4 score of 26.95 when initializing mBART with weights trained on cc25 [18]. Another plausible explanation could be slight deviations in implementations, perhaps in conjunction with the aforementioned cause.

5.1.4 Approximating Glosses

Here, the results of our ablation study surrounding the approximation of glosses are presented. The performance benchmarks are compared to to external literature in which glosses are either approximated or entirely omitted from the model pipeline.

Sign2Gloss		**Gloss2Text**	
WER		**BLEU-4**	
`Val`	`Test`	`Val`	`Test`
70.03	71.20	70.09	70.51

Table 10: Sign2Gloss and Gloss2Text performance after training with approximated glosses.

As evident from table 10, there is a substantial increase in the WER when performing Sign2Gloss pre-training. This is expected given that glosses are now approximated from the natural language translations resulting in less consistency and more redundancy in annotations. This is backed up by considering the significant gap in the Gloss2Text results compared to those of table 8. Hence, the approximated gloss annotations bear a much closer resemblance to the ground-truth translations. Below,

we present the results of our Sign2Text model with approximated glosses and compare it to relevant benchmarks. The components of the model are initialized with the weights corresponding to the results observed in table 10.

Method	ROUGE-L	BLEU-1	BLEU-2	BLEU-3	BLEU-4
TSPNet* [16]	34.96	**36.10**	23.12	16.88	13.41
SL-Luong* [3]	31.80	32.24	19.03	12.83	9.58
Ours w. gloss approx.	**36.77**	35.15	**24.95**	**18.97**	**15.14**

Table 11: Comparison of our model performance when gloss annotations have been approximated with other benchmarks. * denotes models whose architecture does not rely on glosses.

As seen in table 11, our model outperforms existing benchmarks who also omit expertly annotated glosses. This is surprising given the relatively simplistic approach to approximating gloss labels. That being said, the performance is of the Sign2Text model is significantly reduced when compared to the model trained on expertly-annotated glosses (see table 9).

5.1.5 Evaluating Real-World Capabilities

Below, our results for the evaluation of the generalization capabilities of our models are presented. To access this, model performance is evaluated on never before seen general-domain data sampled from the DGS Corpus dataset [10].

Dataset	Sample Strategy	OOV %	ROUGE-L	BLEU-1	BLEU-2	BLEU-3	BLEU-4
PHOENIX-2014T	None	2.01%	52.07	50.24	39.24	31.78	26.52
DGS Corpus	Domain-Relevant	25.58%	5.48	4.53	0.57	0.01	0.00
DGS Corpus	Random	50.11%	4.18	3.81	0.47	0.00	0.00

Table 12: Evaluation of the Sign2Text model on never before seen general-domain data. OOV denotes the proportion of words that are out-of-vocabulary within the samples. The first row shows our performance on the within-domain PHOENIX-2014T [2] test set as comparison. The domain-relevant sample strategy denotes sampling data with respect to the overlap of vocabulary between the natural language translations and the vocabulary of the PHOENIX-2014T dataset. The random sample strategy denotes sampling data at random, without regards for vocabulary overlap. In both cases, 1000 samples were drawn from the DGS Corpus [10].

As evident from the results shown in table 12, our model performs significantly worse on the new distribution of the general-domain data. This worsening of performance is to be expected, as the model is trained on the very specific vocabulary and domain of the PHOENIX-2014T dataset, along with the relatively limited amount of data it contains. This illustrates the lack of generalizability present in the current model and puts into question the validity of the training procedure in a novel user context.

5.1.6 Effect of Temporal Downsampling on Performance

Below, we present the effect on model performance when temporally downsampling inputs with varying factors k_t and methods of interpolation.

k_t	**Interpolation**	**BLEU-4**
1	N/A (original)	**26.52**
1/2	Linear	21.23
1/2	Nearest	22.27
1/3	Linear	16.57
1/3	Nearest	16.68

Table 13: The effect of temporally downsampling videos on model performance as measured by BLEU-4 score, without retraining. k_t denotes downsampling rate, i.e. the relative temporal resolution compared to the original. The BLEU-4 scores are computed over the entire test set of PHOENIX-2014T. The interpolation mode Nearest corresponds to copying the existing frame temporally closest to the interpolated frame. This is what the tests below use.

As seen in table 13, downsampling the input temporally affects translation accuracy considerably. Surprisingly, we observe a similar decrease in model performance compared to the ablation study conducted by SOTA [5] in spite of not training the model on downsampled input videos.

5.2 Application Performance

In this section we will present the results of running our test suites for benchmarking the execution times and attempting to quantify the contributions to app latency of the individual components.

5.2.1 Sign2Text

We will first concern ourselves with analyzing the execution-time of the Sign2Text pipeline. Figure 31 of section 8.3 in the appendix displays two scatter plots over total execution time of our Sign2Text procedure, given input length on CPU and GPU, respectively. The sampling of these videos are in accordance with section 4.7.1. The data points in each scatter plot are fitted using linear regression to estimate the trend of the relation.

These models show a linear trend for the execution time given the input length, for both the CPU and GPU results, but in the case of the GPU, there is significantly more variance.

Given the linear models, we now extrapolate some benchmark values for different reasonable inputs. These benchmarks are reported in subtables (**a**) and (**b**) in table 14 below. The subtables show the total execution time, execution times per component, as well as how large a percentage of the total execution time each component corresponds to, given an input of length 100, 200, and 300 frames, respectively.

CPU									
n_{frames}	WebmByteToTensor		VideoPipeline		Sign2Text		GoogleTranslate		**Total**
100	0.06s	2.07%	0.03s	0.91%	2.84s	92.04%	0.15s	4.98%	**3.09s**
200	0.09s	1.73%	0.05s	1.04%	4.89s	94.27%	0.15s	2.96%	**5.19s**
300	0.12s	1.58%	0.08s	1.09%	6.94s	95.21%	0.15s	2.11%	**7.29s**

(a) Performance on CPU.

GPU											
n_{frames}	WebmByteToTensor		ToGPU		VideoPipeline		Sign2Text		GoogleTranslate		**Total**
100	0.07s	7.23%	0.01s	0.56%	3e-4s	0.03%	0.61s	60.96%	0.31s	31.22%	**1.01s**
200	0.11s	6.32%	0.01s	0.33%	3e-4s	0.02%	1.29s	75.06%	0.31s	18.28%	**1.72s**
300	0.14s	5.94%	0.01s	0.23%	4e-4s	0.01%	1.97s	80.89%	0.31s	12.92%	**2.43s**

(b) Performance on GPU.

Table 14: Examples of execution times and the percentage of the total time they correspond to, broken down in the pipeline components, according to our linear model in figure 31. Input sizes are 100, 200 and 300 frames respectively.

Below we further illustrate the execution time of individual application components. This is shown in figure 27 in the form of a waterfall diagram depicting CPU and GPU execution times, given the mean length of the sample videos (130 frames). The diagram is a stacked horizontal bar chart that is sorted according to the order in which the components occur in the pipelines.

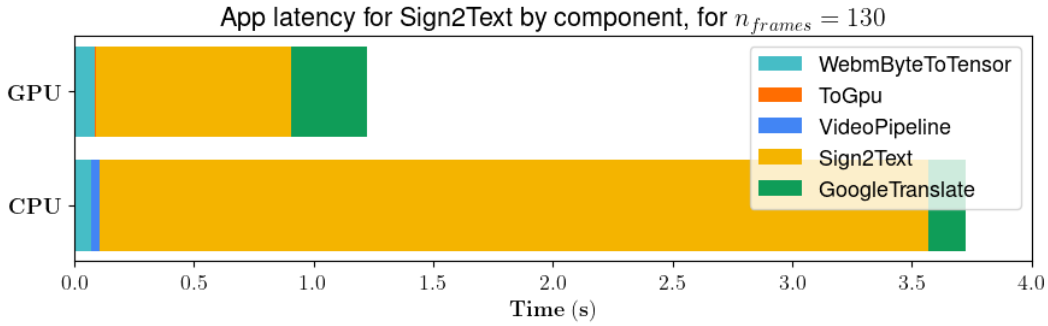


Figure 27: Waterfall diagram for visualizing the contribution to latency by each individual component. The ordering along the temporal axis is the same they are called in the model server.

5.2.2 Speech2Text

For Speech2Text, we evaluate the execution times of the pipeline on both CPU and GPU, using 36 samples of videos of speech, recorded by ourselves.

In figure 32 in section 8.4 of the appendix, we present scatter plots of total execution times and the corresponding input lengths, along with a plot of the function resulting from applying linear regression to the data points.

As with our Sign2Text execution time analysis, we use the linear models to extrapolate some benchmark values for different reasonable inputs. These benchmarks are reported in subtables (**a**) and (**b**) in table 15 below. The table presents the total execution time, execution times per component, as well as how large a percentage of the total execution time each component corresponds to, given an input of length 5, 10, and 15 seconds, respectively.

CPU							
Input length	WebmToWaveform		Speech2Text		GoogleTranslate		**Total**
5s	0.052s	2.89%	1.574s	87.30%	0.177s	9.81%	**1.80s**
10s	0.052s	1.79%	2.687s	92.14%	0.177s	6.07%	**2.92s**
15s	0.052s	1.29%	3.799s	94.31%	0.177s	4.39%	**4.03s**

(a) Performance on CPU.

GPU							
Input length	WebmToWaveform		Speech2Text		GoogleTranslate		**Total**
5s	0.029s	2.82%	0.561s	55.17%	0.428s	42.02%	**1.02s**
10s	0.029s	2.38%	0.749s	62.15%	0.428s	35.48%	**1.21s**
15s	0.029s	2.06%	0.937s	67.24%	0.428s	30.70%	**1.39s**

(b) Performance on GPU.

Table 15: Examples of execution times broken down in the Speech2Text pipeline components, according to our linear model in figure 31. Input sizes are 5, 10 and 15 seconds respectively.

Like with Sign2Text, we further illustrate the execution time of individual application components. This is shown in figure 28 in the form of a waterfall diagram depicting CPU and GPU execution times, given the mean length of the sample videos (10.84 seconds). The diagram is a stacked horizontal bar chart that is sorted according to the order in which the components occur in the pipelines.

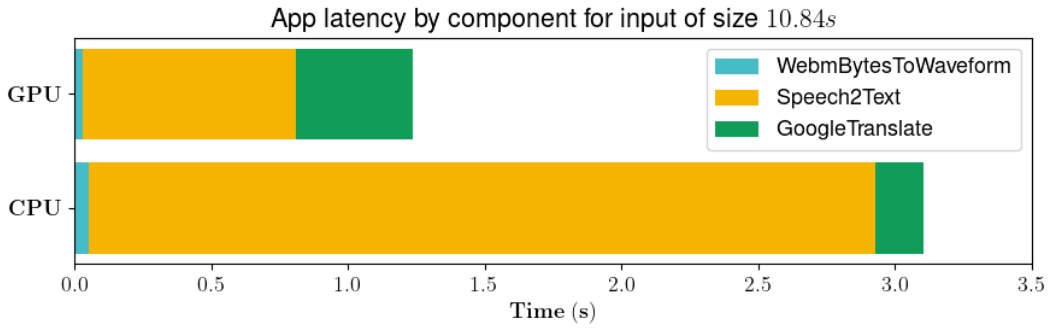


Figure 28: Waterfall diagram for visualizing the contribution to latency by each individual component. The ordering along the temporal axis is the same they are called in the model server.

5.2.3 Effect of Temporal Downsampling on Latency

The results below present the speedup accomplished on both CPU and GPU by using temporal downsampling and thus lowering the temporal resolution.
CPU. Firstly, we present the results from the latency results associated with temporal downsampling on CPU.

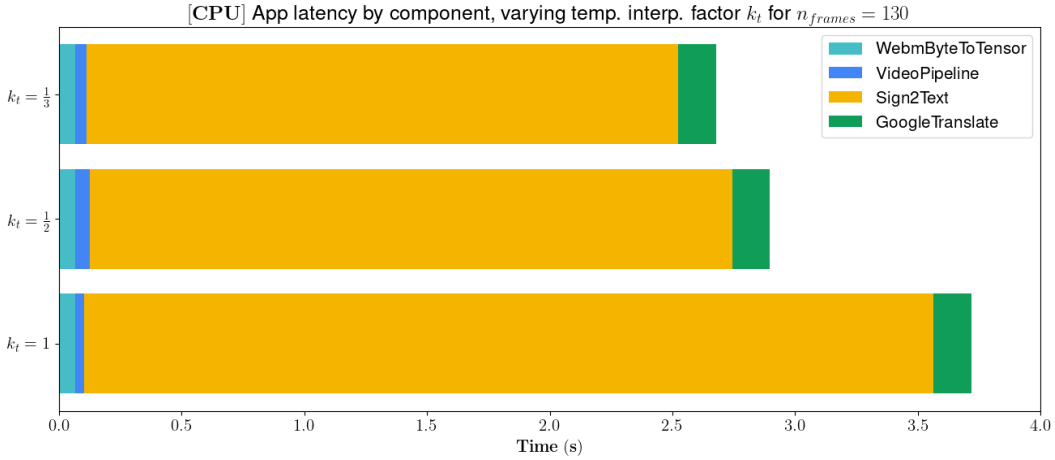


Figure 29: Waterfall diagram breaking down the time contribution to app latency by each component for a 130 frame long video, which is the mean of the subset we tested on. **WebmByteToTensor**: conversion from video bytes to tensor. **VideoPipeline**: described in section 4.4 and illustrated in figure 21. See table 16 for further breakdown of the latency. **Sign2Text**: running model inference.

CPU									
k_t	WebmByteToTensor		VideoPipeline		Sign2Text		GoogleTranslate		**Total**
1	0.067s	1.81%	0.036s	0.96%	3.462s	93.09%	0.154s	4.14%	**3.72s**
$\frac{1}{2}$	0.067s	2.31%	0.058s	2.00%	2.619s	90.39%	0.154s	5.31%	**2.90s**
$\frac{1}{3}$	0.067s	2.51%	0.047s	1.74%	2.411s	90.01%	0.154s	5.74%	**2.68s**

Table 16: Absolute (in seconds) and relative (as a percentage) contribution to latency by component in the Sign2Text translation pipeline as it occurs in the application. The exact measurement uses an input video of 130 frames, which is the mean of the subset we tested on. Measured using different temporal interpolation factors, k_t. Tests are performed on a subset of size 100 with varying lengths, using a 2021 MacBook Pro with a 8-core M1 Pro Processor.

GPU. Hereafter, we present the results from the latency results associated with temporal downsampling on GPU.

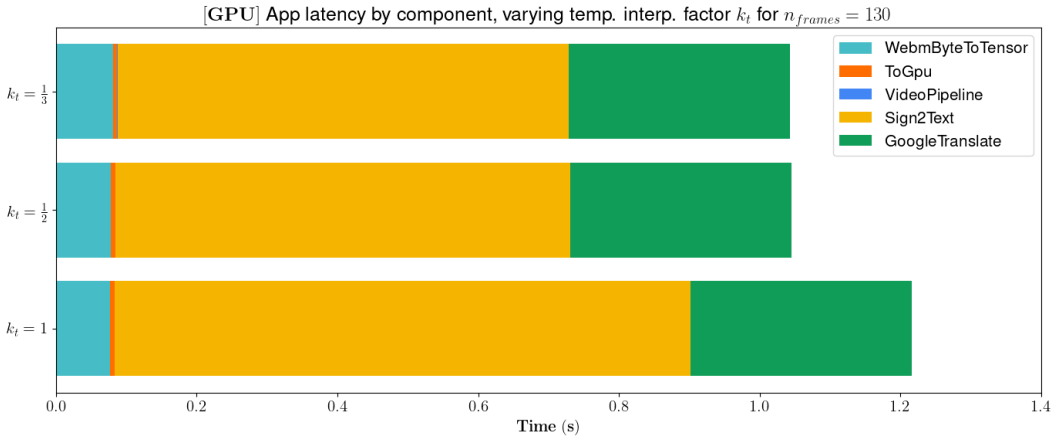


Figure 30: Waterfall diagram breaking down the time contribution to app latency by each component for a 130 frame long video, which is the mean of the subset we tested on. **ToGpu**: sending tensor from CPU memory to the GPU memory. See table 17 for further breakdown of the latency.

	GPU										
k_t	WebmByteToTensor		ToGPU		VideoPipeline		Sign2Text		GoogleTranslate		**Total**
1	0.077s	6.37%	6e-3s	0.46%	3e-4s	0.03%	0.818s	67.29%	0.314s	25.85%	**1.22s**
$\frac{1}{2}$	0.079s	7.51%	6e-3s	0.53%	5e-4s	0.05%	0.646s	61.84%	0.314s	30.07%	**1.05s**
$\frac{1}{3}$	0.082s	7.82%	5e-3s	0.53%	5e-4s	0.04%	0.641s	61.48%	0.314s	30.13%	**1.04s**

Table 17: Absolute and relative contribution to latency by component in the Sign2Text translation pipeline as it occurs in the application. Measured using different temporal interpolation factors, k_t. Tests are performed on a subset of size 100 with varying lengths, using a RTX 3060 graphics processing unit (GPU). Note that the time-scale is much smaller, due to the large speedup by the GPU. Note that the significantly larger latency from Google Translate is dependent on external factors and is thus inherently uncontrollable.

As can be seen in figure 30, when using a combination of temporal downsampling and running on a GPU, we get very close to our definition of true real-time in the context of our application, which is a latency of < 1 second. In fact, if we do not perform the spoken language translation step by calling the Google Translate API, we do achieve real-time performance.

Also, variance in the latency of Google Translate means that even when we use it, we might still achieve true real-time performance.

6 Discussion

Here, the results outlined in section 5 are discussed in detail in the context of our research questions. Additionally, we present ideas for future work both with respect to modeling and the application.

6.1 The Sign2Text Model

Recall that one of the goals of this project was to reproduce the current state-of-the-art (SOTA) method [5] through manual implementation from scratch. Considering the results of the Sign2Text model in table 9, reproducing the performance was not fully possible. This could be attributed to random deviations during training. As noted earlier, it was not feasible to train the model several times due to constraints on computational resources, as well as time. Another plausible explanation for the slight decrease in performance is minor deviations in implementation compared to SOTA, as a result of manually implementing the outlined method from scratch. Such deviations would likely lie within either the Gloss2Text pre-training, Sign2Text training, or a combination of the two. That being said, the achieved performances during the pre-training steps as well as the final Sign2Text training are either close to or on par with the results of SOTA.

Although SOTA achieve state-of-the-art performance on PHOENIX-2014T as well as CSL Daily [31], it is worth noting that the outlined training procedure is extremely prone to overfitting. As highlighted earlier, both the visual backbone and head network as well as mBART are trained twice on the dataset in question. This is backed up by evaluating the model performance on samples drawn from the DGS corpus [10] as seen in table 12. From this, it is clear that the model has a hard time generalizing to unseen data even when the majority of words are present in the vocabulary of the model. Considering some of the observed failure cases of the model, it is clear that one of the model's shortcomings is predicting dates. This is also in line with the findings of SOTA [5] and can in part be attributed to the low frequency of each date within the training set. Furthermore, there is a large degree of overlap in the phrase being uttered prior to a date within the training set of PHOENIX-2014T. As such, the model may have overfitted to the sub-sentence prior to the date and then it subsequently hallucinates or guesses some date. This tendency to learn entire phrases that are likely to occur within the training data could explain the unsatisfactory results when evaluating performance on the DGS corpus.

6.1.1 Reproducibility

A general trend when comparing our results at various stages against those obtained by SOTA[5] is that our performance is slightly lower. Several underlying factors could play a part in this. For one, SOTA simply referenced a previous paper regarding WLASL pretraining details, and spite of following the outlined approach, the results during Sign2Gloss training could not be replicated. Additionally, SOTA published their code half way through this project, and their repository contains code

for several SLT papers. This made differentiating between which code-snippets were used for what project cumbersome. Also, several of their implementations were either not described in the paper or only briefly touched upon. This includes the usage of dropout layers as well as latent dimension in the convolutional block in the head network, the usage of KL divergence as the loss function of mBART and the procedure of applying label smoothing. Additionally, when comparing the implementation details described in SOTA to the configuration files in the associated repository there are clear omissions and discrepancies. This lack of specification lead to additional missteps during implementation and added to the difficulty of reproducing the model from scratch.

6.1.2 Approximating Glosses

Although SOTA [5] achieve state-of-the-art performance, the reliance on glosses constitutes a major bottleneck with respect to data availability. Considering the conducted ablation study, the Sign2Gloss and Gloss2Text results, as seen in table 10, indicate that there is a large degree of noise present in the generated gloss labels. Naturally, the Gloss2Text task becomes significantly easier since the gloss annotations and natural language sequences bear more resemblance to each other. On the contrary, the Sign2Gloss mapping is made significantly harder which can be attributed to redundancy in and inconsistencies with the labels. As noted earlier, a key assumption with the CTC training objective is temporal consistency between the input sequence and target labels which is very likely to be violated in most appoximated gloss labelings.

That being said, the results obtained during Sign2Text training when approximating glosses, as seen in table 11, are encouraging. Comparing the model performances with external benchmarks may suggest that approximating glosses and leveraging supervised methods is beneficial over existing approach which entirely omit them. The relative simplicity of the approximation method further adds to this. Generating glosses based on natural language translations and linguistic rules of the sign language in question is, of course, universal. However, downstream performance will likely vary based on the linguistic properties of the sign language.

6.2 Assessing the Real-Time Problem

A key aspect of facilitating efficient two-way communication between users is latency. Many of the design choices surrounding the development of the application were made precisely with this in mind.

As evident from the results in table 14 as well as figures 31(a) and 31(b), the app latency scales approximately linearly with the number of input frames. This trend is most prevalent when running on CPU. This can be attributed to the application pipeline, up until the beam search decoding of mBART, being sped up significantly. As such, fluctuations in inference time are primarily dependent on beam search decoding which is run on CPU. The latency associated with this step will have a large variance since it is dependent on the signal in the input video. Additionally,

the effect of applying Text2Speech and post-hoc translation on the app latency is negligible.

The results in 14 show that the our application performance for both Sign2Text and Speech2Text does approach realtime performance, as we have defined it, but only when using hardware acceleration by running model inference on the GPU. The rate at which model inference time grows as a function of the input size is simply much higher on CPU.

As noted earlier, [5] conducted an ablation study on the effect on performance when training the various stages of the pipeline of the model on temporally downsampled inputs temporally. They find that downsampling with rates $k_t = \frac{1}{2} \mapsto 50\%$ and $k_t = \frac{1}{3} \mapsto 66.7\%$ yields BLEU-4 scores of 25.59 and 21.89, respectively. Surprisingly, we observe a similar level of degrade in performance when simply running inference on downsampled videos without re-training the model, see table 13. Coupling this with the results seen in table 14, there is an apparent trade-off between performance and app latency which may be desirable in some use-cases. This served as motivation for incorporating the option of specifying this trade-off within the application as illustrated in figure 25 (b). Thus, incorporating temporal downsampling in the video processing pipeline could in fact prove to be useful for improving the real-time performance of the application.

6.3 Creating a Real World Application

To assess the usefulness of the developed model in an end-user context, we test the Sign2Text model on samples from the DGS Corpus [10]. Although the ideal scenario would be evaluating with a native DGS speaker it was, in spite of numerous attempts, not possible to obtain a contact. Naturally, PHOENIX-2014T is confined to the specific-domain of weather forecasts whereas DGS corpus consists of conversations that span a general-domain. For this reason, we assessed performance both on cases where samples are randomly drawn from the DGS Corpus as well as samples where we account for whether words are present in the vocabulary. The former is used to assess the effectiveness of the current system in a general user setting. The latter is used as a proxy to assess how well a model with identical architecture whose vocabulary spans a general-domain would perform on new users.

From the results of this experiment, as seen in table 12, it is evident that the Sign2Text model is unable to generalize to new samples. Although, the proportions of words that are out-of-vocabulary is relatively high, this cannot adequately explain the decrease in model performance. A more plausible explanation lies with the training procedure and its proneness to overfitting as touched upon earlier. Although the fps of the DGS Corpus is 50, twice the amount of PHOENIX-2014T, temporally downsampling the samples has a negligible difference on performance. Coupling this, with the results in table 13, this is presumably not the deciding factor behind the unsatisfactory results.

Although these results put into question the validity of the approach, this experiment cannot assess the generalizability of a Sign2Text model trained on a large, general-domain corpus.

6.4 Accommodating Language Differences

When aiming to enable two-way communication between signers and non-signers in an application setting, it might be beneficial to accommodate both differences in modalities as well as language. To access whether it is possible to accommodate these differences, without compromising latency and accuracy, we conduct the tests presented in 5.2.

6.4.1 Associated Latency

As a means to access whether it is possible to accommodate language differences in a real world sign language translation application, we discuss upon the results from 5.2.

Accommodating language differences. As can be seen by the results of table 14, the vast majority of the latency associated with performing SLT inference on our application is caused by the Sign2Text model. On both the tests conducted on CPU and on GPU, the natural language translation model only accounts for 0.15s and 0.31s latency, respectively. The difference in latency for the two tests, might be due to differences in internet connection. However, while the natural language translation only accounts for $4-5\%$ of the inference time on CPU, it almost accounts for one third of the inference time on GPU. Hereby, SLT inference on GPU increases from below to above the true real-time criteria specified in section 4.2. Thus, it might be difficult to achieve true real-time performance, while accommodating language differences, given our current application architecture.

Accommodating modality differences. As can be seen by the results of table 15, the ability to enable true two-way communication through Speech2Text translation, comes at a cost of latency. Even though this latency is relatively insignificant, the Speech2Text inference still surpasses the real-time criteria specified in section 4.2. However, it can be argued that omitting this feature would result in a worsening of the end-user experience, as the two-way communication between signer and non-signers would be compromised.

6.4.2 Accumulating Inaccuracy

Moreover, as each of the models implemented in the application architecture have an associated error, we also introduce greater inaccuracies into our pipeline, by accommodating the aforementioned language differences. As the application inference pipeline depends on 2 and 3 independent models for Speech2Text and Sign2Speech respectively, the inaccuracy of each model will accumulate downstream. However, while the chosen models all achieve near state-of-the-art performance, it is still important to keep these accumulating inaccuracies in mind.

Naturally models within the domain of neural machine translation, speech synthesis and automatic speech recognition will improve over time. However, as we designed the application pipeline modularized, replacing each model independently will require few adjustments to the architecture.

6.5 Future Research - Towards Commercial SLT

Several additional challenges present themselves when considering to launch a SLT application in a real-world setting. For one, SLT datasets are under restrictive licensing and do not allow for commercial usage. Thereby, preliminary data collection will be required and ideally some extraordinary license agreement with data publishers. Due to the scarcity of data, it would be reasonable to incorporate data collection into the application pipeline. Additionally, the generalizability of the state-of-the-art approach introduced in [5] remains to be determined. Moreover the latency surrounding translations entails that communication between signers and non-signers will have a noticeable delay.

6.5.1 General domain SLT

Being able to conduct SLT in a general domain will first and foremost require a large and varying vocabulary. To leverage datasets which fulfill this prerequisite, the omission of expert annotated glosses is necessary as seen in table 3. This is also backed up by the remarks from the authors of How2Sign [6] regarding annotation time. In the following, we consider different means to achieve this.

Gloss approximation. A general approach to combating the requirement for gold-standard annotations is approximation. Within other domains of computer vision, resorting to weakly supervised learning as a means to increase data quantities has been demonstrated to greatly increase the generalizability and robustness of algorithms [19] [13]. Although the outlined approach to approximating glosses yields encouraging results on the PHOENIX-2014T dataset, it is evident that there remains a significant gap in comparison to relying on ground-truth annotations.

Future research could focus on enhancing gloss approximations and perhaps omitting the requirement of expert knowledge regarding the properties of the sign language in question. A potentially useful resource in the context of gloss approximation is the ASLG-PC12 dataset [30], a large artificially generated corpus containing parallel English sentences and ASL glosses. Considering this in conjunction with OpenASL [25] and How2Sign [6] could help facilitate general-domain SLT models.

Omitting glosses entirely. Another way of leveraging available SLT datasets could be training an autoencoder to reconstruct the raw videos and using this as the visual backbone. Although this could satisfy temporal alignment of the latent sign representations, it will introduce a large degree noise. Without some intervention, the embedding space of such a model might capture more information about the environment and features of the person signing rather than the actual signs and facial expressions. Thus, an important component of such a system would be invariance towards body type, race and background. If not accounted for, inductive biases unrelated to the signed sequence may weigh in on the final prediction.

Assessing whether this criteria is fulfilled could be possible through WLASL [15] considering its diverse signer background and gloss distribution as seen in table 2 and figure 2 respectively. Given that a set of $K = \{1, ..., M\}$ distinct signers all

sign the same phrase, it would be reasonable to want the model to yield identical outputs i.e. satisfy that $P(\hat{y}|K = i) = P(\hat{y}|K = j)\ \forall_{i,j} \in K$. However, this does not take into account whether the errors of the model are the same. Perhaps a better heuristic would be separation and thereby conditioning on the ground truth along with the signer ID i.e. $P(\hat{y}|y, K = i) = P(\hat{y}|y, K = j)\ \forall_{i,j} \in K$. This constraint is equivalent to equalizing the TPR and FPR of the model.

Perhaps bias and noise mitigation in the context of having an autoencoder as the visual backbone could be achieved through spatial masking of the input features as a preprocessing step. Such a method should preserve temporal features as well as information about the depths of objects present in the video. Such an approach could simultaneously address the concerns raised surrounding privacy.

6.5.2 Approaching Real-Time Through Cloud Hosting

Considering the contribution of different pipeline components on the app latency as shown in table 14, it is clear that Sign2Text model inference is the primary bottleneck. Although inference on the Google Translate model is called through an API and therefore acquire the latency related to this, the associated latency is orders of magnitude lower then that of the sign2text model.

These findings suggest, that our application inference latency might benefit from moving all local model instances to a capable cloud hosting service. This is further supported by the results from figure 31(b), where even a consumer level accelerated hardware show significant improvements in inference latency, even approaching real-time as defined in 4.2.

By assuming that our Sign2Text model could obtain similar inference time as the Google Translate model, when moved to a cloud hosting service, this restructuring of our pipeline could potentially enable the application to achieve real-time performance as defined in section 4.2. Furthermore, by moving our local model instances to a cloud hosting service, we also reduce the requirements for the end-user hardware.

Moreover, as we designed the application architecture modularized, this implementation change would require few changes to the source code.

However, the data format of the sign language videos are represented by vastly more bytes, than both natural text and audio speech. This could increase the latency of model significantly when having to send the sign videos through an API connection.

A way to combat this, would be to implement the sign2text model as a hybrid between a local instance and a cloud hosted API. For this setup, the visual encoder of the model could run as a local instance, while the modality mapper and language model was running on a cloud hosting service. Thus, the visual encoder would compress the information of the sign videos into embeddings. These embeddings would significantly reduce the amount of bytes sent through the API by a factor of 1176.

6.5.3 Benefits of a Modularized Architecture

Lastly, as mentioned previously in section 6.4.2, the fields of SLT, neural machine translation, speech synthesis and automatic speech recognition will evolve in the future. Considering future improvements of both our own SLT model, in addition to external dependencies, the translation pipeline of our application might become obsolete in the near future.

An example of these improvements of external dependencies, is the newly published paper Scaling Speech Technology to 1,000+ Languages from Meta AI [23]. Here, the authors achieve a more than double in performance of their Speech2Text model compared to OpenAI Whisper [24]. Thus, our pipeline might will likely see improvements by replacing this component.

However, as we designed the application architecture modularized, replacing each individual component requires few implementation details. Thus, by modularizing the pipeline, we ensure an extendable and easily maintainable application.

7 Conclusion

To conclude the project, we have successfully reproduced the state-of-the-art sign language translation model introduced in SOTA [5]. The slight deviations in performance is likely a result of minor deviations within implementation as well as noise in measurements. Additionally, we have assessed the effect of omitting expertly-annotated glosses within the model framework. In spite of the simplicity of the approach, we outperform existing approaches in which glosses are entirely omitted. This suggests that gloss approximation may be a fruitful avenue for future research, as a means to leverage large datasets within SLT.

Furthermore, we have developed an application which approaches real-time communication, as defined in section 4.2, between signers and non-signers on consumer-grade hardware. In this context, a trade-off between app latency and performance was observed by means of temporal downsampling. To accommodate use-cases in which this may be desirable, user-specification of this parameter was enabled. Additionally, language barriers between users are alleviated through the usage of various external APIs.

Lastly, we have assessed the generalizability of the developed model by drawing samples from the general-domain DGS corpus. This proxy task illustrates that the developed model fails to generalize to new samples. This suggests that the training procedure of the model is highly prone to overfitting. Assessing whether this inability to generalize would hold had the model been trained on a large SLT dataset is, however, not accounted for through this experiment. As such, we leave this as an open-ended research question.

References

[1] Samuel Albanie, Gül Varol, Liliane Momeni, Hannah Bull, Triantafyllos Afouras, Himel Chowdhury, Neil Fox, Bencie Woll, Rob Cooper, Andrew McParland, and Andrew Zisserman. BOBSL: BBC-Oxford British Sign Language Dataset. 2021.

[2] Necati Cihan Camgöz, Simon Hadfield, Oscar Koller, Hermann Ney, and R. Bowden. Neural sign language translation. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7784–7793, 2018.

[3] Necati Cihan Camgöz, Simon Hadfield, Oscar Koller, Hermann Ney, and R. Bowden. Neural sign language translation. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7784–7793, 2018.

[4] Joao Carreira and Andrew Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset, 2018.

[5] Yutong Chen, Fangyun Wei, Xiao Sun, Zhirong Wu, and Stephen Lin. A simple multi-modality transfer learning baseline for sign language translation. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5110–5120, 2022.

[6] Amanda Duarte, Shruti Palaskar, Lucas Ventura, Deepti Ghadiyaram, Kenneth DeHaan, Florian Metze, Jordi Torres, and Xavier Giro-i Nieto. How2Sign: A Large-scale Multimodal Dataset for Continuous American Sign Language. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[7] National Geographic. Sign language. `https://education.nationalgeographic.org/resource/sign-language/`.

[8] Raghav Goyal, Samira Ebrahimi Kahou, Vincent Michalski, Joanna Materzynska, Susanne Westphal, Heuna Kim, Valentin Haenel, Ingo Fründ, Peter Yianilos, Moritz Mueller-Freitag, Florian Hoppe, Christian Thurau, Ingo Bax, and Roland Memisevic. The "something something" video database for learning and evaluating visual common sense. *CoRR*, abs/1706.04261, 2017.

[9] Alex Graves, Santiago Fernández, Faustino J. Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. *Proceedings of the 23rd international conference on Machine learning*, 2006.

[10] Thomas Hanke, Marc Schulder, Reiner Konrad, and Elena Jahn. Extending the Public DGS Corpus in size and depth. In *Proceedings of the LREC2020 9th Workshop on the Representation and Processing of Sign Languages: Sign Language Resources in the Service of the Language Community, Technological Challenges and Application Perspectives*, pages 75–82, Marseille, France, May 2020. European Language Resources Association (ELRA).

[11] Hamid Reza Vaezi Joze and Oscar Koller. Ms-asl: A large-scale data set and benchmark for understanding american sign language, 2019.

[12] Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, Mustafa Suleyman, and Andrew Zisserman. The kinetics human action video dataset, 2017.

[13] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. Big transfer (bit): General visual representation learning, 2020.

[14] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019.

[15] Dongxu Li, Cristian Rodriguez-Opazo, Xin Yu, and Hongdong Li. Word-level deep sign language recognition from video: A new large-scale dataset and methods comparison. *2020 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1448–1458, 2019.

[16] Dongxu Li, Chenchen Xu, Xin Yu, Kaihao Zhang, Ben Swift, Hanna Suominen, and Hongdong Li. Tspnet: Hierarchical feature learning via temporal semantic pyramid for sign language translation, 2020.

[17] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Annual Meeting of the Association for Computational Linguistics*, 2004.

[18] Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. Multilingual denoising pre-training for neural machine translation. *Transactions of the Association for Computational Linguistics*, 8:726–742, 2020.

[19] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens van der Maaten. Exploring the limits of weakly supervised pretraining, 2018.

[20] Amit Moryossef, Kayo Yin, Graham Neubig, and Yoav Goldberg. Data augmentation for sign language gloss translation. In *Machine Translation Summit*, 2021.

[21] World Health Organization. Deafness and hearing loss. `https://www.who.int/health-topics/hearing-loss#tab=tab_2`.

[22] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[23] Vineel Pratap, Andros Tjandra, Bowen Shi, Paden Tomasello, Arun Babu, Sayani Kundu, Ali Elkahky, Zhaoheng Ni, Apoorv Vyas, Maryam Fazel-Zarandi, and et al. Scaling speech technology to 1,000+ languages, May 2023.

[24] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. *ArXiv*, abs/2212.04356, 2022.

[25] Bowen Shi, Diane Brentari, Greg Shakhnarovich, and Karen Livescu. Open-domain sign language translation learned from online video. In *EMNLP*, 2022.

[26] Suramya Tomar. Converting video formats with ffmpeg. *Linux Journal*, 2006(146):10, 2006.

[27] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *ArXiv*, abs/1609.03499, 2016.

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[29] Saining Xie, Chen Sun, Jonathan Huang, Zhuowen Tu, and Kevin P. Murphy. Rethinking spatiotemporal feature learning: Speed-accuracy trade-offs in video classification. In *European Conference on Computer Vision*, 2017.

[30] Kayo Yin and Jesse Read. Better sign language translation with STMC-transformer. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 5975–5989, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics.

[31] Hao Zhou, Wengang Zhou, Weizhen Qi, Junfu Pu, and Houqiang Li. Improving sign language translation with monolingual data by sign back-translation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1316–1325, 2021.

[32] Ronglai Zuo, Fangyun Wei, and Brian Mak. Natural language-assisted sign language recognition, 2023.

8 Appendix

8.1 Table of Contributions

Project Subject	**A. Bigom**(%)	**M. Harborg**(%)	**N. R. Holm**(%)
Model			
Sign2Gloss	30	30	40
Gloss2Text	40	30	30
Sign2Text	40	30	30
Ablation Study	25	50	25
Performance Tests	30	30	40
Application			
Frontend	40	30	30
Video Processing Pipeline	30	30	40
Backend Model Server	30	30	40
External API modules	40	30	30
Firebase Authentication	40	30	30
Firestore Cloud Database	40	30	30
Application Build	40	30	30
Performance Tests	30	30	40

Table 18: Table of contribution for the implementations of the project.

Project Subject	**A. Bigom**(%)	**M. Harborg**(%)	**N. R. Holm**(%)
Introduction			
Research Questions	Equal.		
Report Outline	30	40	30
Data			
Presentation and Summary Statistics	40	30	30
Ethical Considerations	30	40	30
Methods			
Proposed Model Architecture	40	30	30
Sign2Gloss Task	30	30	40
Gloss2Text Task	30	40	30
Sign2Text Task	40	30	30
Evaluation Metrics	30	30	40
Omitting Glosses - Ablation Study	30	40	30
Software Product			
Problem Formulations	30	40	30
The Real-Time Problem	30	30	40
Backend Architecture	40	30	30
Efficient Pipeline for Video[...]	30	30	40
Model Integrations	30	40	30
Application Architecture	40	30	30
Performance Analysis	30	30	40
Results			
Model Performance	30	40	30
Application Performance	30	30	40
Discussion			
The Sign2Text Model	40	30	30
Assessing the Real-Time Problem	30	30	40
Creating a Real World Application	40	30	30
Accommodating Language Differences	30	40	30
Future Research	30	30	40
Conclusion			
Conclusion	Equal.		

Table 19: Table of contribution for the report of the project.

8.2 Links to Project and Application

Item	**Link**
Sign2Gloss GitHub Repository	GitHub
Gloss2Text GitHub Repository	GitHub
Sign2Text GitHub Repository	GitHub
Application Frontend GitHub Repository	Due to unwanted access to the API keys used, this repo is private. Please contact `andreasbigom@gmail.com` for access.
Application Backend GitHub Repository	GitHub
Model Checkpoints	Google Drive
Application Installation	Google Drive
CTC_decoding.ipynb	Google Colab

Table 20: Link to the project code base, model checkpoints for the results presented in 5 and for the installation of the application.

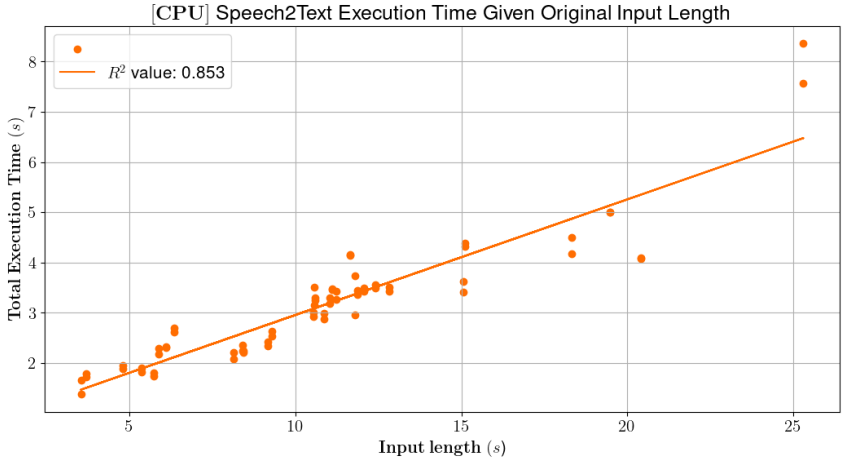
8.3 Linear Models for Execution Time - Sign2Text

(a) Linear approximation of the execution of the Sign2Text application pipeline on CPU. Tests are performed on a 2021 Apple MacBook Pro M1 Pro with 8 CPU cores. Note that k_t is a temporal downsampling factor, and is only displayed here to

(b) Linear approximation of the execution of the Sign2Text application pipeline on GPU. Tests are performed using an NVIDIA RTX 3060 GPU with 8GB of VRAM.

Figure 31: Linear models for execution time comparison of Sign2Text pipeline.

8.4 Linear Models for Execution Time - Speech2Text



(a) Linear approximation of the execution of the Speech2Text application pipeline on CPU. Tests are performed on a 2021 Apple MacBook Pro M1 Pro with 8 CPU cores.

(b) Linear approximation of the execution of the Speech2Text application pipeline on GPU. Tests are performed using an NVIDIA RTX 3060 GPU with 8GB of VRAM.

Figure 32: Linear models for execution time comparison of Speech2Text pipeline.