50.021 AI Project Report

Gay Shinlee 1007001
Ho Xiaoyang 1007006
Janya Mali 1007149
Pankti Shah 1007130
Ryan Lee 1006652

April 2025

1 Introduction

1.1 Motivation

Approximately 5% of the global population—over 360 million individuals—suffers from disabling hearing loss[1]. These individuals often face substantial challenges, ranging from communication barriers in daily interactions to experiences of social exclusion and discrimination.

Recent advancements in artificial intelligence (AI) offer promising avenues to address accessibility challenges faced by people with hearing impairments. Motivated by the growing acceptance and effectiveness of AI-driven assistive technologies, our goal is to develop a system that facilitates communication by recognizing sign language, the primary mode of communication for many individuals with hearing loss. By leveraging state-of-the-art AI methodologies, we aim to contribute towards reducing communication barriers and promoting inclusivity.

1.2 Task Description

Sign language recognition (SLR) is an important research area situated at the intersection of computer vision and natural language processing. It focuses on translating visual-gestural communication into structured linguistic representations. In particular, isolated sign language recognition targets the identification of individual signs or phrases from discrete video sequences, without assuming grammatical or temporal continuity between consecutive signs.

In this project, we decompose the broader objective of sign language translation into two core components:

[1]According to the World Health Organization (WHO):`https://www.who.int/news-room/fact-sheets/detail/deafness-and-hearing-loss`

Isolated Sign Recognition

We formulate isolated sign recognition as a many-to-one sequence classification task, where a model processes a temporal sequence of landmark frames and maps it to a single sign label corresponding to a word or phrase.

To enable effective learning, it is crucial to translate raw sensory data—such as images or video frames—into structured input representations that capture the temporal dynamics and spatial configurations inherent to sign language gestures. Single frames are often insufficient to disambiguate signs, as many gestures are defined by dynamic motion patterns rather than static poses. Thus, sequence modeling becomes essential for accurate recognition.

Moreover, the motions and structure of sign language vary significantly across linguistic communities and regions. American Sign Language (ASL), British Sign Language (BSL), and other variants each possess unique syntax and gesture sets. Therefore, selecting a consistent and representative dataset is a critical prerequisite to narrow the scope of our task and ensure effective training and evaluation of isolated sign recognition systems.

Predicted Sign Sequence to Natural Language Composition

Building upon isolated sign classification, we propose a second-stage sequence-to-sequence pipeline that interprets sequences of predicted sign labels to generate coherent natural language sentences. This component addresses the structural gap between discrete sign recognition and conversational language understanding.

By applying natural language processing (NLP) models, we aim to compose ordered sequences of classified signs into fluent and grammatically correct sentences. This two-stage architecture enhances scalability from word-level classification to sentence-level translation, enabling more natural, human-like communication. Our long-term vision parallels the development of conversational agents—such as chatbots—capable of facilitating fluid interaction with individuals using sign language as their primary mode of expression.

2 Dataset and Preprocessing

2.1 Dataset Selection

Given the requirements outlined in the task description, we selected the dataset from the Google Isolated Sign Language Recognition (ISLR) Kaggle competition. This dataset is particularly well-suited for our objective of developing an isolated sign recognition system due to its size, structure, and focus on isolated sign gestures.

Each sample in the dataset represents a temporal sequence of 3D keypoint landmarks extracted from video frames using the Mediapipe framework. The landmarks correspond to various parts of the human body, including the face, hands, and body joints. Each sequence is annotated with a ground-truth label indicating the performed sign, representing either a word or a phrase.

Importantly, the dataset is aligned with our project scope in several ways:

- **Focus on Isolated Signs:** Each sequence corresponds to an individual sign, without requiring modeling of continuous signing or grammatical transitions.
- **Structured Temporal Data:** The 3D landmarks from Mediapipe provide a compact yet expressive representation of motion, making the dataset amenable to sequential models such as recurrent networks or transformers.
- **Sufficient Scale:** The dataset contains 94,477 sequences covering 250 distinct sign classes, providing ample diversity for training deep learning models.

Furthermore, the dataset targets American Sign Language (ASL), allowing us to focus on a consistent linguistic variant without the confounding influence of regional differences in sign language syntax or semantics. This alignment between dataset characteristics and project objectives makes it an ideal choice for our study.

2.2 Data Structure

The dataset is organized using CSV and Parquet files:

- The provided CSV file contains metadata, including the relative path to each sample's Parquet file, participant ID, sequence ID, and the corresponding sign label.
- Each Parquet file contains the raw keypoint data for a single sequence. It includes the following columns: `frame`, `row_id`, `type` (denoting landmark group: face, left hand, pose, right hand), `landmark_index`, and the normalized `x`, `y`, and `z` coordinates.

For modeling purposes, we focus on the 3D coordinates (`x`, `y`, `z`) as the primary features, disregarding identifiers such as `row_id` and focusing solely on spatial and temporal patterns of motion.

2.3 Problem Formulation

Using this dataset, we formulate the task as a supervised learning problem over multivariate time-series data. Each input sample is a multivariate temporal sequence of 3D keypoints, and the corresponding output is a single categorical label identifying the performed sign. Since sign language gestures are inherently dynamic, the temporal ordering of frames is crucial; frames are non-exchangeable, and the model must learn to capture the evolution of motion over time.

Overall, the dataset's structure enables direct application of sequence modeling approaches, facilitating end-to-end supervised learning for isolated sign recognition.

2.4 Data Preprocessing

The data preprocessing pipeline aims to reduce input dimensionality, standardize sample distributions, and enrich the feature space for downstream modeling.

Point Landmarks: A subset of informative keypoints, denoted as `point_landmarks`, is selected to focus on the most relevant spatial landmarks.

Remove Depth Estimation: To minimize noise associated with depth estimation, only the x and y-coordinates are retained, discarding the z-axis information.

Normalization: Normalization proceeds in two stages. First, all landmarks are centered around the nose position (landmark 17) by subtracting its mean location across frames, thereby standardizing the spatial origin for each sequence. Second, the entire sequence is scaled by the global standard deviation across all landmarks and frames, ensuring scale invariance across samples.

Truncation/Padding: Sequences are truncated or padded to a fixed length $T_{\max}$ to enable uniform batching. Padding is achieved by replicating the final valid frame, and a binary mask is generated to indicate valid versus padded frames, which is subsequently used to ignore padding during model training and evaluation.

Motion Dynamics: Beyond raw positional information, motion dynamics are encoded via first- and second-order differences. Specifically, velocity is computed as the frame-wise difference $\Delta x_t = x_{t+1} - x_t$, and acceleration as $\Delta^2 x_t = \Delta x_{t+1} - \Delta x_t$. The final feature representation for each frame concatenates position, velocity, and acceleration, yielding an input of the form $[\text{position}, \text{velocity}, \text{acceleration}]$ for both x and y-axes.

Data Partitioning: Finally, the dataset is randomly shuffled before partitioning into training, validation, and test splits to minimize sampling bias and ensure statistical representativeness. The sequences are mutually independent and share no overlapping frame data, thereby preventing any form of temporal leakage.

2.5 Data Augmentation

To enhance model robustness and generalization to unseen conditions, several augmentation strategies are applied at the sequence level:

Temporal Rescaling: Each sequence is randomly stretched or compressed by a scaling factor $s \sim \mathcal{U}(0.5, 1.5)$ (uniform distribution), followed by linear interpolation to maintain consistent input length. This augmentation simulates variations in signing speed.

Random Temporal Masking: A random subset of frames is replaced with `NaN` values, simulating occlusions or missing frames. This forces the model to rely on broader temporal context rather than individual frames.

Horizontal Flipping: With a fixed probability p_{flip}, x-coordinates are horizontally flipped around the center axis, effectively mirroring signing gestures and augmenting spatial diversity.

Affine Transformations: Random affine transformations—including rotation, translation, scaling, and shearing—are applied independently to each frame. This augmentation introduces plausible spatial perturbations without altering the gesture semantics.

Cutout (Frame Dropout): A contiguous block of frames is randomly zeroed out within each sequence, simulating occlusion or dropped frames. This encourages the model to be resilient to temporal discontinuities and localized frame loss.

These augmentation techniques collectively simulate a wide range of real-world variations in sign language, improving the model's capacity to generalize across speakers, environments, and recording conditions. Since sign language sequences can vary widely in signing speed, gesture orientation, and motion patterns, these augmentations help simulate this natural variability. By exposing the model to a broader distribution of input patterns, augmentation mitigates overfitting to the limited training data and enhances the model's ability to handle unseen real-world variations.

3 Models

3.1 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining a hidden state that evolves over time. This structure allows them to capture temporal patterns, making them particularly suitable for tasks like sign language recognition, where the meaning of a gesture unfolds across a sequence of frames.

Despite their conceptual simplicity, vanilla RNNs are limited by issues such as vanishing and exploding gradients, which make learning long-term dependencies difficult. As a result, they often struggle to retain information from earlier frames that may be critical for recognizing a complete sign. To address these challenges, more sophisticated variants like Long Short-Term Memory (LSTM) networks and Gated Recurrent Unit (GRU) networks introduce gating mechanisms that enable better control over the information flow across time steps.

In our work, after the RNN models processed the entire sequence, the final hidden representations are aggregated and passed through a classification head to predict the corresponding sign.

Long Short-Term Memory (LSTM) Networks

LSTM networks enhance the basic RNN structure with a memory cell and three gates—input, forget, and output—that regulate how information is updated and maintained across time steps. This design helps overcome the vanishing gradient problem and allows the network to model longer-range dependencies.

In the context of sign language recognition, this capability is crucial, as different parts of a gesture may be temporally separated but jointly important for correct classification. Our LSTM models consist of stacked layers, with an option to operate bidirectionally. Bidirectional LSTMs enable the network to incorporate information from both past and future frames, offering a fuller view of the gesture evolution through time. This is done by processing the sequence in both directions and concatenating the hidden states. The final aggregated hidden states are fed into a linear layer to produce the predicted sign label.

Gated Recurrent Unit (GRU) Networks

GRUs simplify the LSTM architecture by combining the input and forget gates into a single update gate and merging the hidden and cell states. This results in fewer parameters and typically faster training, while still retaining the ability to capture important sequence dynamics.

For isolated sign recognition, we hypothesize that GRUs may be more efficient in capturing short and mid-range temporal patterns but may struggle with long range dependencies in comparison to LSTM architectures. However, they serve as an efficient alternative without the additional computational overhead of full LSTM architectures. Like with LSTMs, the GRU can also operate bidirectionally.

3.2 Transformers

Transformers represent a fundamentally different approach to sequence modeling by relying entirely on self-attention rather than step-by-step recurrence. Each position interacts with all others directly through attention mechanisms (attends to other positions), enabling the model to capture dependencies across arbitrary distances in the sequence.

In sign language recognition, not all frames contribute equally to the meaning of a sign. Key frames that define hand shapes, motions, or body positions may occur non-contiguously. Transformers are particularly well-suited for this setting, as self-attention allows the model to dynamically focus on the most informative frames without being constrained to local or recent time steps.

Transformer Encoder

The attention mechanism computes interactions between frames by generating query, key, and value vectors for each input and applying scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V,$$

where d_k is the dimensionality of the key vectors. This mechanism has each position attend to all other positions, enabling each position to be contextualized with respect to the entire input sequence with direct connections.

As our task is formulated as a many-to-one classification problem—mapping an input sequence to a single label (sign)—we only require the encoder component of the Transformer; there is no need for a decoder, which is typically used for sequence generation tasks. Our implementation closely follows the original Transformer encoder architecture proposed by Vaswani et al. [3], with a few modifications. Each time step's feature vector is first projected into a higher-dimensional embedding space through a linear layer. To preserve temporal ordering, we add sinusoidal positional encodings to these embeddings. This step is critical, as the self-attention mechanism is inherently permutation-invariant: it treats the input as an unordered set. For time-dependent tasks like sign recognition, encoding positional information is essential to allow the model to reason about the progression of frames.

The positionally encoded sequence is then processed through the Transformer encoder layers. Each layer consists of a multi-head self-attention module followed by a two-layer feed-forward network, with residual connections applied around both sublayers. Before each sublayer, we apply batch normalization (rather than the original layer normalization). Dropout is applied after the attention and feed-forward modules to promote regularization. The refined sequence representations are then aggregated and passed to a classification head to predict the sign label.

Linformer

We also implemented a variant Transformer encoder layer, `LinTransformer`, based on the Linformer Transformer architecture proposed by Wang et al. [4]. The Linformer introduces an alternative self-attention mechanism that reduces the computational and memory complexity of standard self-attention from quadratic $\mathcal{O}(n^2)$ to linear $\mathcal{O}(n)$ with respect to the sequence length n. This is achieved by projecting the key and value matrices into lower-dimensional representations using learned projection matrices, motivated by the empirical observation that the attention matrix is often low-rank in practice. By approximating the full attention computation, Linformer aims to maintain comparable representational capacity while dramatically improving efficiency, especially for longer sequences.

Given our computational constraints, we were interested in exploring whether this design could deliver faster inference times and reduced resource usage while maintaining competitive classification performance. We implemented this variant to evaluate its practical impact both on the final classification success and on inference speed within our sign sequence recognition pipeline.

3.3 1D Convolutional Neural Network (CNN) Module

Inspired by and borrowing heavily from the first-place solution of Hoyeol Sohn (`hoyso48`) in the ISLR Kaggle competition[2], we incorporate a 1D Convolutional Neural Network (CNN) as an early feature extraction layer. This model design reflects the intuition that sequential input data, such as the frames of hand and body landmarks in sign language recognition, often exhibits strong inter-frame correlations. To leverage this, we pass the input sequence through a 1D CNN before the convolutional outputs are fed into a sequential processing model.

The 1D CNN can be thought of as serving the role of a tokenizer for the sequence, in the sense that it aggregates adjacent frames into tokens, transforming the raw time series into a set of high-level temporal embeddings. These embeddings serve as more informative and compact representations of the sequence, capturing important local temporal patterns and reducing the complexity of the downstream model. The intuition behind using a CNN at this stage is that local temporal dependencies within the sequence—such as the movement of hand landmarks over a short period—are easier to capture with convolutional filters. This is particularly true for sign language, where gestures often consist of distinct, localized motions that are highly correlated from frame to frame. In our implementation, the 1D CNN is applied across the temporal dimension of the sequence, transforming frame-wise landmark features into a sequence of high-level embeddings. This serves as a form of dimensionality reduction, organizing and compressing local temporal information into a compact representation that retains crucial features for the sequential model.

This approach is supported by Hoyeol Sohn's results, where a pure 1D CNN architecture was able to achieve a strong score without the need for any other sequential modeling. However, while convolutional layers excel at capturing local dependencies, they are not well-suited to model long-term temporal dependencies due to their limited receptive field. By combining the 1D CNN with a subsequent sequential model, we aim to maximize capture of both local short-term dependencies (using the CNN) and global long-term dependencies (using sequential models). This hybrid approach allows us to benefit from the strengths of both convolutional layers and sequential modeling architectures, potentially improving performance on tasks like sign language recognition where both short-term and long-term temporal patterns are important. These ideas were implemented as the following module:

[2]Discussion: `https://www.kaggle.com/competitions/asl-signs/discussion/406684`

Conv1DBlock Each block applies a sequence of operations designed for efficient temporal feature extraction:

- **Feature Expansion:** The input feature dimension is first expanded, allowing the model to operate in a higher-dimensional space for richer intermediate representations.
- **Causal Depthwise Convolution (`CausalDWConv1D`):** Depthwise 1D convolution with causal padding is used to ensure that each output at time t depends only on inputs up to and including t. Dilations expand the receptive field without significantly increasing computational cost.
- **Batch Normalization:** Stabilizes training dynamics after convolutional operations.
- **Efficient Channel Attention (`ECA`):** Lightweight channel-wise attention is applied after batch normalization. ECA compresses temporal information with global average pooling and models channel interactions via a 1D convolution followed by a sigmoid gating function, enhancing important features without adding heavy computation.
- **Projection:** The feature dimension is reduced back to the target size.
- **Residual Connection:** A skip connection is added to promote better gradient flow and model stability.
- **Dropout (optional):** Dropout can be optionally applied after projection for regularization.

3.4 Late Dropout Regularization

Across all our model architectures, we apply `LateDropout` as a form of regularization. Unlike standard dropout, which is active throughout training, `LateDropout` delays the application of dropout until a specified number of training steps have been completed.

This strategy allows the model to first stabilize and learn strong feature representations during the early phases of training without the stochastic noise introduced by dropout. Once the threshold is reached, dropout is activated during training, helping to prevent overfitting by randomly zeroing portions of the input features. This phased regularization approach has been shown to improve final generalization performance, especially in large or complex models.

3.5 Model Architecture and Hyperparameters

All of our models share a common structure consisting of an input processing module, followed by three stacked `Conv1DBlocks` (described previously), a sequential modeling layer (e.g., LSTM, GRU, Transformer encoder, Linformer encoder), another three `Conv1DBlocks`, and a second sequential modeling layer of the same type. This hybrid design first extracts local temporal features, globally contextualizes them, and subsequently refines these representations through a second stage of local and sequential processing. The final contextualized sequence is then passed to a classification head to produce the output predictions. Take note that some hyperparameters for our models (such as `dim` are taken from CFG class that specifies configurations, and does not follow the default hyperparameters set in the model definitions.

Input Processing: The input sequence is first processed by a masking layer, which ignores padded frames based on a predefined padding value to prevent models from attending to artificial padding tokens. It is then projected into a higher-dimensional feature space (dimension 192) using a bias-free dense layer, followed by batch normalization with a momentum of 0.95.

Gated Recurrent Unit (GRU): The GRU module uses 192 hidden units per direction. Forward and backward hidden states are concatenated at each time step, producing a 512-dimensional representation. We also found that the bi-directional variant performed better.

Long Short Term Memory (LSTM): The configuration for the LSTM module is similar to the GRU module. Similarly, we found that the bi-directional variant of the LSTM performed better than the uni-directional variant.

Causal Depthwise Convolution Blocks (`Conv1DBlock`): Each `Conv1DBlock` uses a kernel size of 17, a dropout probability of 0.2, and a stride and padding of 1.

Transformer Encoder: Each Transformer encoder layer employs multi-head self-attention with a model dimension of 192, 4 attention heads, and an internal dropout probability of 0.2. The self-attention module is followed by a two-layer feedforward network with an expansion factor of 4. Residual connections and dropout (with probability 0.2) are applied after both the attention and feedforward sublayers.

Linformer Encoder (`LinTransformer`): Each LinTransformer encoder layer employs a linear self-attention mechanism (LinformerSelfAttention) with a model dimension of 192, 4 attention heads, and a key/value projection dimension of 128 to reduce attention complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. A dropout probability of 0.2 is applied to the attention weights. The self-attention module is followed by a two-layer feedforward network with an expansion factor of 4 and Swish activation. Residual connections and dropout (with probability 0.2) are applied after both the attention and feedforward sublayers. Batch normalization is applied before both the attention and feedforward modules to enhance training stability.

Classification Head: The output of the final sequential modeling layer is passed through a dense layer that independently projects each time step's feature vector by doubling it to 384 dimensions. A global average pooling operation aggregates information across the temporal dimension into a fixed-length vector. `LateDropout` with a probability of 0.8 is then applied for additional regularization. Finally, a dense layer projects the pooled features to the 250 output classes, producing logits for classification.

Our model architectures and hyperparameters were selected based on empirical performance observed during preliminary experiments. Due to constraints in time and computational resources, particularly given the size of the dataset, we were only able to conduct limited hyperparameter tuning. For parameters with less obvious optimal values, such as kernel size, padding, and certain projection dimensions, we explored a small range of options. For other parameters, we generally adopted standard or reasonable default values based on prior conventions, given the constraints. A more exhaustive search over the hyperparameter space is left for future work.

Importantly, hyperparameter choices were guided by performance only on the validation set; the test set was reserved exclusively for final evaluation to ensure an unbiased estimate of generalization performance.

3.5.1 Examples of inital experiments:

Before developing the final CNN-based hybrid architectures, we explored baseline implementations using Gated Recurrent Units (GRU) and Bidirectional GRUs (Bi-GRU). These models were chosen for their ability to handle temporal dependencies in sequential data such as gesture sequences. Both models used pre-processed landmark data derived from MediaPipe's holistic model, with sequences of frame-wise features representing hand, pose, and facial motion.

In the GRU model, each input sequence was fed into a unidirectional GRU layer followed by fully connected layers for classification. The model was trained for 10 epochs using the Adam optimizer and cross-entropy loss. While training progressed without technical issues, the resulting accuracy plateaued early and failed to generalize well. The highest accuracy reached was approximately 23.1%, indicating that the model was insufficiently expressive to capture fine-grained motion patterns.

The same results were seen in the LSTM model configuration. We suspect that there was more downsampling needed by adding more CNN layers, and that the model needed to be more expressive. This led us to abandon this version in favour of the current CNN + BiGRU and CNN + BiLSTM models that incorporated normalized features, frame differencing (velocity and acceleration), data augmentation, and a more expressive architecture.

Model Architecture: The model architecture for our final six models are shown here. We refer to each in this work by their caption names. The latter four architectures follow the architecture described above:

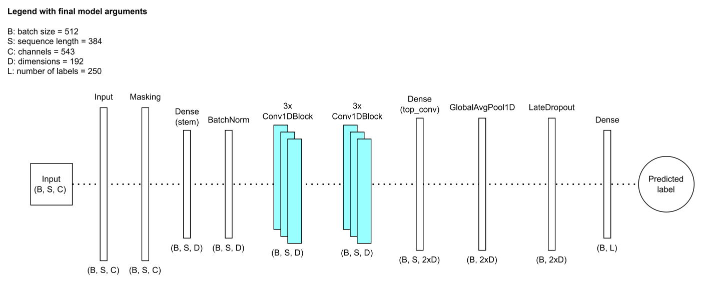


Figure 1: CNN-only

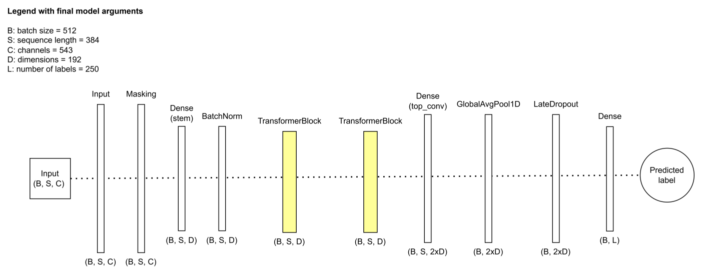


Figure 2: Transformer-only

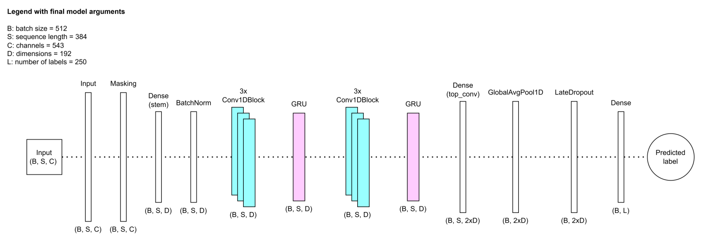


Figure 3: CNN + BiGRU

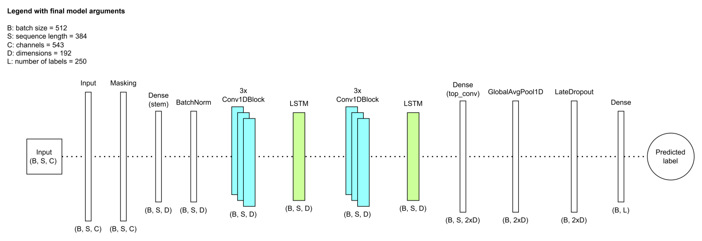


Figure 4: CNN + BiLSTM

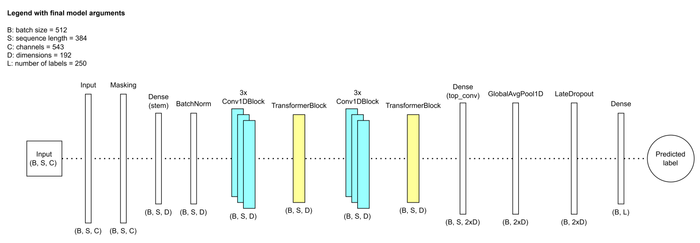


Figure 5: CNN + Transformer

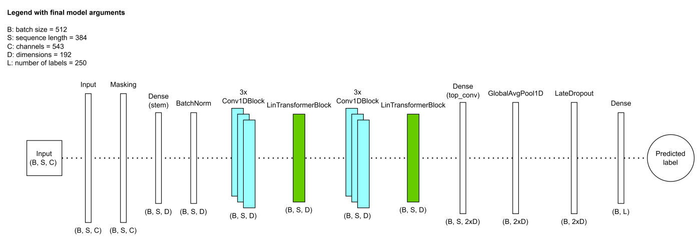


Figure 6: CNN + Linformer

4 Training

We designed the training pipeline to prioritize performance, reproducibility, and robustness under computational constraints. We leverage TensorFlow's `tf.distribute.Strategy` for efficient multi-GPU training. Random seeds were fixed, memory was cleared before training, and mixed precision was used when supported by hardware to maximize efficiency.

The dataset was split into 4-folds. Our pipeline supports multi-fold training, in which one fold is used for evaluation and the remaining folds are used to train the model. We further split the assigned fold equally in half into a new validation set and a test set. Across the splits, there are 140 `tfrecords` files for training, 23 for validation, and 24 for testing, totaling 94,477 sequences. However, due to computational limitations, we only trained on a single iteration (i.e. only one fold was split into validation and test sets).

Similar to our model tuning process, our hyperparameter tuning for training was limited in scope. We focused finding good values for impactful hyperparameters such as learning rate and weight decay. Other settings (e.g., optimizer choice) were fixed based on standard best practices. Like before, determination of optimal hyperparameter values were based on validation set performance; the test set was strictly held back for final evaluation to prevent data leakage, ensuring unbiased assessment.

4.1 Training Details

To reduce the time taken for training due to Kaggle's compute limitations, we will standardize and train our models for 120 epochs each. We have achieved a satisfactory performance for most of models with this amount of training, and training for more epochs will have negligible improvement to our model (diminishing returns).

- **Optimizer**: Rectified Adam (RAdam) combined with Lookahead. RAdam stabilizes training during the early phases, while Lookahead further improves convergence by interpolating between fast and slow weights.
- **Learning rate schedule**: OneCycle learning rate scheduling with cosine decay. OneCycle allows for aggressive learning rate increases early on, improving exploration and convergence, while cosine decay provides a smooth annealing toward the end of training.
- **Weight decay**: Adaptively scheduled using a second OneCycle scheduler, ensuring stronger regularization early on while relaxing constraints as training progresses. We fixed it at a reasonable value of 0.1.
- **AWP** (Adversarial Weight Perturbation) starts after 15 epochs, with $\lambda = 0.2$.
- **Batch size**: 512 (64×8 replicas/TPU cores).
- **Maximum sequence length**: 384 tokens. Shorter sequences in the dataset are padded then masked.
- **Loss**: Categorical Crossentropy with label smoothing (0.1).

We trained the six model architectures described in the previous section, each using our standardized data preprocessing and augmentation pipelines. To further assess the impact of data augmentation on model performance and generalization, we additionally trained a CNN + Transformer model without data augmentation, providing a baseline for comparison. This resulted in a total of seven distinct training runs.

The training curves for all seven runs are included in Table 2 found in the Appendix. Notably, the CNN + Linformer model exhibited a sharp spike in both training and validation loss around the 16th epoch. After this point, the validation loss continued to deteriorate, even producing NaN values in the final epochs (too large). We suspect that exploding gradients occurred, destabilizing the training process irrecoverably. In contrast, the standard Transformer model showed a smaller, temporary loss spike around the same epoch but successfully recovered, indicating that Linformer is more vulnerable to optimization instability.

In hindsight, it would have been beneficial to adjust the learning rate or incorporate gradient clipping specifically for the Linformer-based model. Because Linformer's low-rank attention approximation trades expressiveness for efficiency, it is often more sensitive to hyperparameter tuning compared to the richer representations generated by standard multi-head attention (MHA) in Transformers. Fine-tuning the optimizer settings specifically for Linformer could have helped stabilize training.

Fortunately, we employed periodic checkpointing during training, saving model weights at regular intervals. For evaluation, we used the best-performing model parameters captured before the loss escalation, ensuring that the final reported results are based on stable, high-quality checkpoints.

5 Evaluation

5.1 Evaluation metrics

To comprehensively assess model performance, we employed the following evaluation metrics.

Categorical Accuracy: Categorical accuracy measures the proportion of correct predictions out of the total number of samples. It serves as a straightforward indicator of overall performance, particularly for balanced datasets where all classes are equally represented.

Top-5 Accuracy: Top-5 accuracy relaxes the strictness of exact matches by considering a prediction correct if the true class label appears among the model's five highest-probability outputs. This metric is useful for complex, fine-grained classification tasks like sign language recognition, where multiple similar classes may cause uncertainty in the top prediction but the correct answer remains among the top candidates.

Macro F1-Score: The macro F1-score computes the F1 score independently for each class and then takes the unweighted average across all classes. This treats all classes equally, regardless of their frequency, making it particularly important for detecting performance degradation on rare or underrepresented signs.

Micro F1-Score: In contrast, the micro F1-score aggregates true positives, false positives, and false negatives across all classes before computing the F1 score. It reflects the overall system performance and tends to be influenced more by the performance on frequent classes. Using both macro and micro F1 scores allows us to evaluate models from both a class-agnostic and a class-frequency-sensitive perspective.

5.2 Results

The evaluation results on the unseen test set for the models from each of the seven training runs are summarized in Table 1.

Model Architecture	**Categorical Accuracy**	**Top-5 Accuracy**	**Macro F1**	**Micro F1**
CNN-only	0.7716	0.9170	0.7669	0.7716
Transformer-only	0.7170	0.8955	0.7102	0.7172
CNN + Transformer (no augmentations)	0.7672	0.9134	0.7634	0.7669
CNN + Transformer	**0.7843**	**0.9233**	**0.7806**	**0.7845**
CNN + BiLSTM	0.6410	0.8631	0.6347	0.6410
CNN + BiGRU	0.6557	0.8663	0.6465	0.6557
CNN + Linformer	0.7045	0.8916	0.7011	0.7050

Table 1: Performance Metrics for Various Model Architectures on Test Set

All models were trained with data preprocessing and data augmentations unless otherwise stated.

Our notebook outputs for training of the models are all saved in the zipped files under the "results" folder.

5.3 Discussion

5.3.1 Analysis of Results

We first evaluated the impact of data augmentation on the CNN + Transformer hybrid architecture. As shown in the results, the model trained with augmentations achieved consistently better performance across all evaluation metrics. This highlights the importance of data augmentation in enhancing model generalization, by exposing the model to a broader distribution of input variations and promoting robustness to irrelevant variations that should not affect the classification outcome.

Regarding architectural comparisons, we observe that the CNN + Transformer models outperform Transformer-only models. This result underscores the effectiveness of the implemented `Conv1DBlock` in local feature extraction. Convolutional layers are naturally adept at capturing localized dependencies and inter-frame relationships, which appear to be highly relevant given the structure of our data. This hypothesis is further supported by the performance of the CNN-only model, which surprisingly outperforms the Transformer-only model, although it does not surpass the CNN + Transformer combination. Together, these results support the intuition that combining convolutional layers (for strong local feature modeling) with Transformer layers (for global context modeling via attention) enables the model to effectively capture both short-range and long-range dependencies, leading to improved performance.

The recurrent models (based on LSTM and GRU) exhibit the weakest performance among the tested architectures. This may be attributed to the inherent limitations of recurrent networks in this context: hidden state representations in RNNs compress sequential information into a fixed-size vector at each timestep, leading to potential information bottlenecks, especially over long sequences. Standard RNNs lack an explicit mechanism to dynamically focus on important frames or positions within the sequence, unlike attention-based models, making them less suited for tasks where selectively attending to key frames is crucial. It should be that the GRU performs slightly better than the LSTM, perhaps due to the aforementioned reason that GRU is better at short-term dependency modeling, and we suspect strong inter-frame correlation in our data.

As for the Linformer variant, while we anticipated a modest degradation in performance relative to the full Transformer, the extent of its underperformance was unexpected. Linformer reduces the computational complexity of self-attention from quadratic to linear time by projecting the key and value matrices into a lower-dimensional space before computing attention. Although this yields substantial theoretical gains in memory and computational efficiency, it inevitably introduces information loss due to the low-rank projections. Our empirical findings suggest that this information loss significantly degrades performance in our task. Moreover, while the Linformer exhibited a slightly faster inference time compared to the baseline Transformer, the improvement was not substantial. We hypothesize that this is due to the relatively modest sequence length (maximum 384 tokens) in our setting, which limits the computational bottlenecks that Linformer is designed to alleviate. Additionally, in our TensorFlow implementation, the overhead introduced by reshaping, transposing, and tf.einsum operations in Linformer may not be as optimized as the tf.matmul operations used in standard multi-head self-attention, further diminishing the practical runtime advantage. Overall, given the significant drop in classification metrics, we conclude that Linformer is not well-suited for our task and that its benefits are likely to be more pronounced in domains with much longer sequence lengths, such as natural language processing tasks involving thousands of tokens.

5.4 Comparisons with Existing Models

Our models, while competitive within the scope of our experiments, do not approach the accuracy achieved by top-performing models on the Kaggle Leaderboard, such as the model developed by the aforementioned Hoyeol Sohn, from which we drew inspiration and borrowed heavily from. His model achieved a classification accuracy of 0.8929287 on the Kaggle private leaderboard, setting a high benchmark for this task.

It is important to note that our evaluation results are not directly comparable to the leaderboard scores, as the test set used for the Kaggle competition remains hidden to prevent participants from purposely training their models on the test set. Instead, we carved out a smaller held-out test set from the publicly available dataset, ensuring it remained strictly unseen during model training and hyperparameter tuning. This approach allowed us to better control our evaluation process and examine a wider range of metrics (e.g., micro/macro F1 scores), whereas submitting to the leaderboard would have only provided a single metric (accuracy).

Given our constraints in computational resources, we intentionally designed smaller models with fewer parameters compared to competition-grade architectures. As a result, achieving leaderboard-topping accuracy was not our primary focus. Rather, our goal was to systematically study how different architectural choices — such as the integration of convolutional blocks, recurrent units, and attention mechanisms — influence model performance. Through these controlled experiments, we sought to better understand the trade-offs between complexity, efficiency, and generalization ability in the context of large-scale sign language recognition. Our trained models are adequately performant enough to support the downstream task of sign language-to-sentence translation.

Although our models underperform relative to the state-of-the-art, the insights gained from architectural comparisons and training dynamics provide valuable directions for future improvements. A natural next step would be to scale up model capacity, leverage larger training budgets, and perform more exhaustive hyperparameter searches to bridge the performance gap while validating the architectural and dataset intuitions developed in this work.

5.5 Future Improvements

There are several promising avenues for future research to further improve performance on this task. One direction is exploring ensemble methods that combine different model architectures, such as CNNs, RNNs, and Transformers. An ensemble could leverage the strengths of each model type, potentially achieving better generalization and robustness compared to any individual model.

Another promising idea is to replace static convolutional patching with dynamic patching strategies, as seen in recent byte-level Transformer architectures. For example, in the Byte-Latent Transformer architecture proposed by Pagoni et al. [2], dynamic patching allows the model to adaptively select and aggregate input segments based on the data, potentially capturing richer local structures than fixed kernel strides. Patches are dynamically segmented according to the entropy of the upcoming byte, directing more computational resources and model capacity to regions of higher complexity, while also enabling a more granular understanding of the sequence.

Finally, further exploration into sequence modeling techniques specialized for longer sequences—such as sparse attention mechanisms or low-rank approximations—could help scale the models more effectively to larger inputs without prohibitive computational costs.

More extensive hyperparameter tuning would also likely yield significant improvements. Due to compute and time constraints, only limited tuning was conducted in this study. With greater computational resources, a broader search across learning rates, optimizer configurations, model depths, dimensionalities, and regularization strategies could uncover substantially better-performing models.

6 Predicted Sign Sequence to Natural Language Sentences

This is framed as a sequence-to-sequence task: from a sequence of predicted isolated signs, we aim to generate coherent natural language sentences. To achieve this, we use the `T5Tokenizer` and `T5ForConditionalGeneration` from the open-source Google FLAN-T5 model family. FLAN-T5 [1] is a fine-tuned version of T5 ("Text-To-Text Transfer Transformer"), a large language model (LLM) that casts all NLP tasks into a text-to-text format. It is pretrained on a mixture of unsupervised and supervised tasks and is then further instruction-tuned to better follow task-specific prompts, making it particularly well-suited for general sequence generation tasks like ours. It should be noted that, unlike chatbot models such as ChatGPT, FLAN-T5 uses an encoder-decoder architecture rather than a decoder-only setup. This design allows FLAN-T5 to explicitly encode the entire input sequence into a rich latent representation before generating the output sequence, aligning it closer with our task framing, where a sequence of predicted signs must be mapped to a coherent natural language sentence.

In our pipeline, after predicting the sign tokens, the sign sequence is tokenized and passed into the FLAN-T5 model, which is prompted to output the final natural language sentence prediction. We use a simple prompt: *"Here are a sequence of words obtained from a sign language recognition model. Take this sequence of words and output a plausible sentence in natural English that the user could possibly be trying to communicate: {`keywords`}"*. This leverages the powerful generative capabilities of LLMs to infer semantic meanings from the input (ordered set of sign predictions) and produce fluent, human-readable output.

6.1 Future Improvements

In this project, we concentrated primarily on the isolated sign recognition component, as it represents a major challenge and acts as a performance bottleneck for the entire pipeline. In contrast, the second stage — converting sign sequences into natural language sentences — relies on a large, pretrained language model, which already benefits from extensive pretraining on large-scale text corpora. These LLMs are inherently strong at understanding and generating language, making them a useful and reliable tool for this task.

Nonetheless, there are several promising directions worth exploring to further enhance this stage. For instance, the system could be improved with task-specific post-training the language model with supervised fine-tuning (SFT) on a dataset of predicted sign sequences paired with their ideal output sentences (mirroring the instruction-answer pair of the typical LLM SFT process). This would directly align the model's generation behavior to the task-specific input distribution. Furthermore, Reinforcement Learning with Human Feedback (RLHF) presents an even more scalable method. Instead of requiring humans to manually craft ideal output sentences, we can collect human preferences by ranking multiple generated candidate sentences. This feedback can be used to train a reward model, which then guides policy updates to the language model using reinforcement learning techniques. RLHF exploits what Andrej Kaparthy[3] calls the "generator-discriminator gap" — it is often easier and cheaper for humans to rank outputs than to explicitly construct perfect ones.

7 Code Usage Guide

7.1 Training and Evaluation code

Before running our training and evaluation code, download the following datasets from the Kaggle competition page and place them in the input folder:

- **Complete dataset with csv, parquet, and json files:** `https://www.kaggle.com/competitions/asl-signs`
- **Dataset in tfrecord format, split into 4 folds:** `https://www.kaggle.com/datasets/hoyso48/islr-5fold`

To run our main preprocessing and training script, run all the cells in `main_training_script.ipynb` from top to bottom in Kaggle. You can specify the model type that you want to train, in the second-last cell, by changing the `model_type` integer argument in `get_model()`:

- **1:** `cnn_transformers_model`
- **2:** `cnn_lstm_model`
- **3:** `cnn_gru_model`
- **4:** `cnn_model`
- **5:** `transformers_model`
- **6:** `cnn_lintransformers_model`

To evaluate trained models (with `.h5` file weights stored in our results folder), upload it to the Kaggle input folder and specify the path to the trained `.h5` model weights. Make sure that the `model_type` corresponds to the model architecture used to obtain the model weights.

[3]Andrej Kaparthy's post on X: `https://x.com/karpathy/status/1821277264996352246`

7.2 Running of GUI

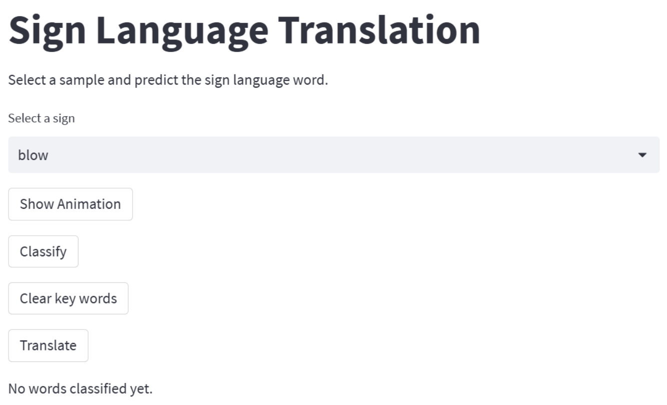
Before running our GUI, download the above mentioned datasets in a Kaggle notebook, placing them in the input folder. Upload the weights for the best-performing CNN + Transformer model. Go to `https://dashboard.ngrok.com`, and navigate to "Your Authtoken" to fetch your personal authtoken. This is used by the ngrok agent to log into your account when a tunnel is started to host the UI online. To run our GUI, follow the instructions in the `ui_for_demo.ipynb` file. Run all the cells in a Kaggle notebook to host the interactive UI on ngrok with streamlit, using your personal ngrok token. Once the last cell is run, click the link that leads to the website UI.

8 GUI

Our GUI is hosted using Streamlit and accessed via ngrok, enabling easy and fast sharing of the interface. We use Kaggle to run the environment setup, which includes pre-installed dependencies like TensorFlow and PyTorch, saving us time on configuration. All our code which includes model loading, preprocessing, and data handling is consolidated into a single `app.py` script. Streamlit makes it simple to deploy this script quickly, and it conveniently retains Kaggle's environment variables, eliminating additional setup steps.

Streamlit is particularly well-suited for rapid prototyping and deployment because it allows developers to transform Python scripts into interactive web apps with minimal code changes. Its compatibility with common data science tools and ability to handle real-time UI updates make it ideal for showcasing machine learning models in action.

This is the first look at our GUI.



Sign Language Translation

Select a sample and predict the sign language word.

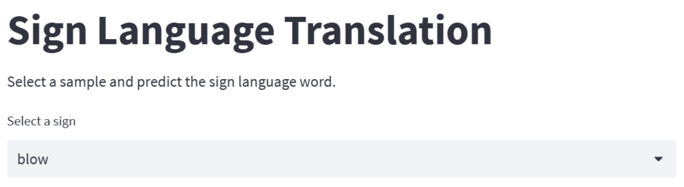
Select a sign

blow

No words classified yet.

Figure 7: Random sample selection

In our demonstration, users can select any of our 250 samples. These are unique sample labels that is equivalent to the number of classes that are present in the dataset.



Sign Language Translation

Select a sample and predict the sign language word.

Select a sign

blow

Figure 8: Random sample selection

Once a sample is selected, the GUI can display an animated visualization of the sign's coordinates (extracted using MediaPipe) along with the ground truth label in the caption. This can be done after the user clicks the "Show Animation" button. We have done this to reduce the loading time to construct and display the animation, only showing it when the user wishes to see it.

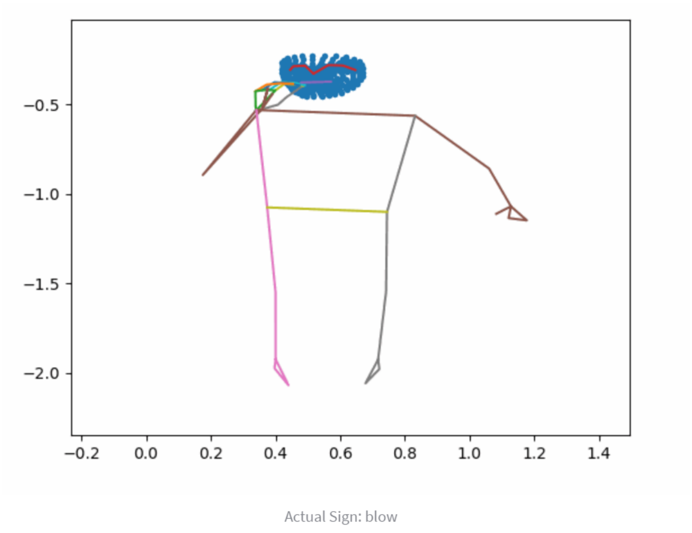
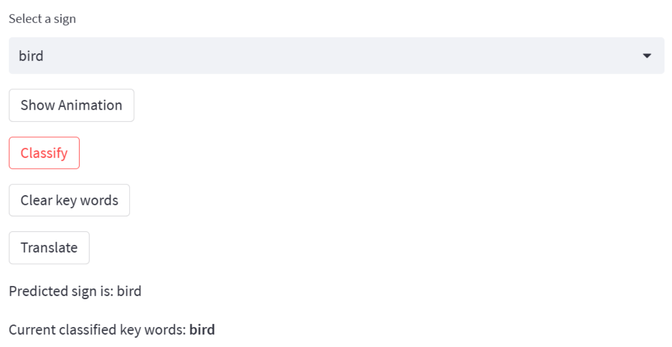


Figure 9: Random sample selection

After clicking the "Classify" button, our trained model predicts the sign, and the result is stored in a session variable.

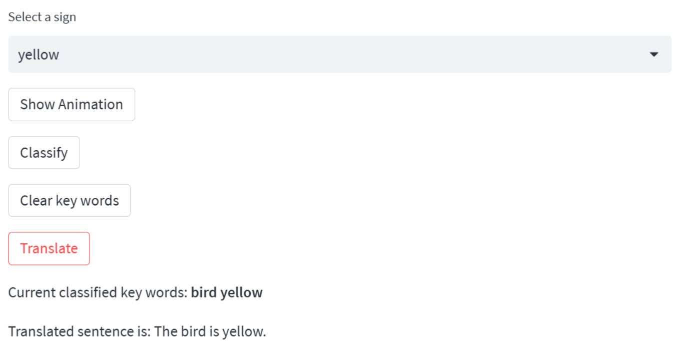


Predicted sign is: bird

Current classified key words: **bird**

Figure 10: Prediction of sign

As users continue classifying samples, all predictions are retained in the session. When the user clicks "Translate", it runs our signs-to-sentence pipeline, generating a complete sentence from the predicted signs. The key words can be cleared from the current session by pressing "Clear key words", allowing users to continue using the GUI to predict and translate other samples.



Current classified key words: **bird yellow**

Translated sentence is: The bird is yellow.

Figure 11: Translation from all predicted signs

Design decisions: We used a persistent saving of session states to maintain the predictions of the individual signs. This is so we can retain a list of key words that can be used to generate a whole sentence. We also cached our classification model, tokenizer and seq-to-seq model to improve loading speeds, making sure that objects are only instantiated once when the user first loads the web UI.

9 Sustainability Goals

This project directly contributes to the United Nations Sustainable Development Goals (SDGs), particularly Goal 10: Reduced Inequalities and Goal 4:

Quality Education. By developing a machine learning system that can recognize and interpret sign language, our work promotes greater accessibility and inclusivity for the Deaf and hard-of-hearing communities. Significantly, over 90% of deaf children are born to hearing parents, many of whom do not know sign language. Our model aims to bridge this communication gap by enabling parents, educators, and peers to understand and learn sign language more easily.

Moreover, we considered sustainability in our design by implementing efficient architectures such as 1D CNNs and GRUs, using mixed-precision training to reduce energy consumption, and leveraging transferable preprocessing pipelines that can generalize across low-resource environments. These approaches lower the computational burden during both training and inference, making the system more adaptable to real-world, edge-device deployment in underserved areas.

Through our GUI deployment on platforms like Streamlit, we also enable accessible digital interfaces that can be easily used by non-technical users, further supporting inclusive technology adoption in line with digital equity goals.

References

[1] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models, 2022.

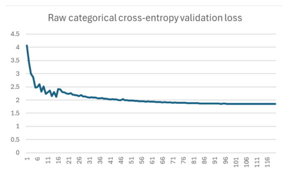
[2] Artidoro Pagnoni, Ram Pasunuru, Pedro Rodriguez, John Nguyen, Benjamin Muller, Margaret Li, Chunting Zhou, Lili Yu, Jason Weston, Luke Zettlemoyer, Gargi Ghosh, Mike Lewis, Ari Holtzman, and Srinivasan Iyer. Byte latent transformer: Patches scale better than tokens, 2024.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

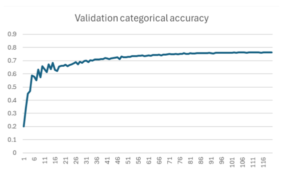
[4] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity, 2020.

Appendix

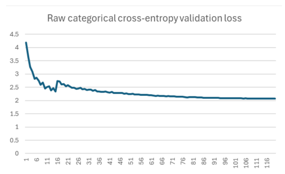
CNN-only model loss



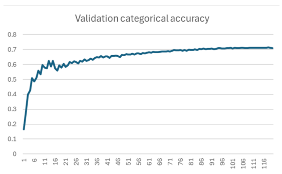
CNN-only model accuracy



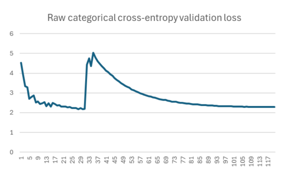
Transformer-only model loss



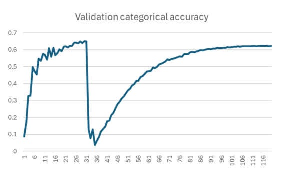
Transformer-only model accuracy



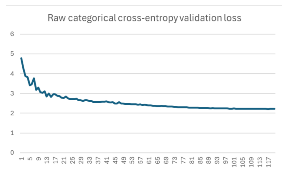
CNN + GRU model loss



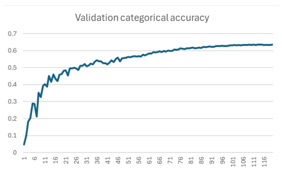
CNN + GRU model accuracy



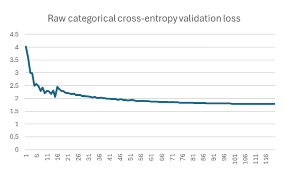
CNN + LSTM model loss



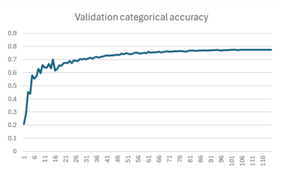
CNN + LSTM model accuracy



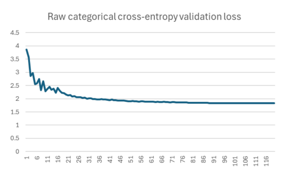
CNN + Transformer model loss



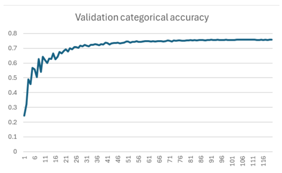
CNN + Transformer model accuracy



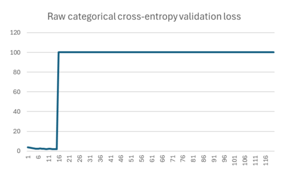
CNN + Transformer model loss (no augmentations)



CNN + Transformer model accuracy (no augmentations)



CNN + LinTransformer model loss



CNN + LinTransformer model accuracy

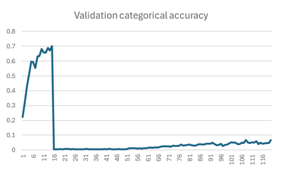


Table 2: Raw categorical cross-entropy validation loss and categorical accuracy curves