Transformer-Based ASL Fingerspelling-to-Text System – Implementation Report

Abstract

This report presents a Transformer-based system for translating American Sign Language (ASL) fingerspelling sequences into text. The implemented solution uses **MediaPipe** to extract 3D hand and upper-body landmarks from video frames, converting raw video into structured time-series data. A sequence-to-sequence **Transformer** model with an encoder-decoder architecture is then trained on a large ASL fingerspelling dataset to translate sequences of hand positions into corresponding English letters. The approach emphasizes data preprocessing and robust model design: landmark coordinates are normalized for consistency across different signers, and the Transformer leverages self-attention to capture contextual dependencies between sequential letters. The implemented system demonstrates accurate recognition of continuous fingerspelled words from landmark input, confirming the effectiveness of a Transformer-based approach to ASL fingerspelling recognition. It achieves this without requiring any explicit per-letter segmentation, instead learning the temporal patterns of letters directly from sequence data. The result is a working proof-of-concept that can form the core of a real-time fingerspelling-to-text translation application.

Introduction

Fingerspelling in ASL involves spelling out words letter-by-letter using specific hand gestures for each letter of the English alphabet. Automating the recognition of fingerspelled words is an important step toward bridging communication between Deaf signers and non-signers. Unlike isolated signs (which have unique gestures for whole words), fingerspelling presents a unique challenge: letters are produced in quick succession, and consecutive hand shapes can influence each other (a phenomenon known as co-articulation). Traditional approaches that classify individual hand shapes in isolation struggle with continuous fingerspelling because they ignore these transitions and context.

In this project, we focus on continuous ASL fingerspelling recognition and address these challenges by leveraging modern machine learning techniques. Our approach treats the problem as a **sequence-to-sequence translation** task, analogous to translating one language to another – here, translating a sequence of visual hand poses into an English word. To make this feasible, the system uses **pose estimation** technology to simplify the visual input. Specifically, we use Google's **MediaPipe** framework to extract a sequence of skeletal landmarks (keypoints of the hands and relevant body parts) from video frames. This converts the high-dimensional video stream into a time-series of coordinate data, which is far easier for a learning model to process. By doing so, we decouple the perception problem (identifying hands and their positions) from the sequence learning problem.

With the input now represented as a series of 3D landmark coordinates, we implement a Transformer-based neural network to learn the mapping from gesture sequences to text. The **Transformer** is well-suited for this task because its attention mechanism can capture long-range

dependencies in the sequence – crucial for understanding which hand movements correspond to which letters in context. The model processes an entire sequence of ASL letters at once, rather than one frame at a time, enabling it to use context (previous and next hand shapes) to disambiguate letters. For example, the hand shape for certain letters might look similar in a single frame, but the Transformer can learn to identify the correct letter by considering adjacent frames and overall sequence patterns.

This project uses a large-scale dataset from the **Google ASL Fingerspelling Recognition** challenge, which contains millions of labeled fingerspelling sequences recorded from over a hundred signers in varied real-world conditions. Importantly, the dataset provides pre-computed MediaPipe landmarks for each frame of each sequence. This wealth of data allows the Transformer model to learn a robust representation of fingerspelling, generalizing across different people's hand sizes, orientations, and speeds. The remainder of this report details the implemented methodology, model architecture, training process, and evaluation of the system's performance.

Methodology

Data Preparation: The first step was constructing a high-throughput data pipeline to handle the ASL fingerspelling dataset. Each training example in the dataset consists of a video (or video segment) of a word being fingerspelled, along with the corresponding text label (the word in English). Instead of processing raw video frames, we utilize the provided **MediaPipe landmark data** for each frame. These landmarks include 3D coordinates for key hand joints (21 points per hand, such as fingertip positions and knuckles) and selected upper-body points (e.g. wrist, elbow, shoulder) that give context to the hand's location relative to the body. By focusing on these salient points, we reduce the input dimensionality and filter out background noise.

All landmark coordinates are normalized to ensure consistency across different signers and camera setups. Normalization involves a few steps: - We translate the coordinates to a consistent origin (for example, the center of the signer's torso or a shoulder point) so that the model is invariant to the signer's position in the frame. - We scale the coordinates based on a reference distance (such as shoulder width) to account for different distances from the camera or different body sizes. - We also ensure that for each sequence, the relative motions of the hand are preserved but extraneous variations (like the person moving around in the frame) are minimized.

Each sequence of landmarks is then padded or truncated to fit a fixed length window as needed for batching during training. In practice, fingerspelled words vary in length (number of letters and thus number of frames). We set a maximum sequence length based on the longest word in the dataset and pad shorter sequences with a special mask value so that all sequences in a batch can be processed together. Masks are used later in the model to ignore these padded timesteps. The preprocessing pipeline was implemented in Python, reading the dataset's landmark files (which are stored in an efficient format) and outputting training-ready sequences of normalized coordinates. This pipeline also handled any missing data gracefully – for example, if a particular frame had missing landmark detections, the script could interpolate from neighboring frames or mark that frame to be ignored via masking.

Model Training Strategy: With the processed sequential data in hand, we trained a neural network to perform the translation from sequences of coordinates to sequences of characters. The

Transformer architecture was chosen for its ability to model long sequences and capture global context through attention. During training, each input sequence (landmark time-series) is fed to the model, and the model is tasked with predicting the output sequence (the spelled word). We employ teacher forcing in training the sequence-to-sequence model: the known ground-truth letters are provided to the decoder at each step to guide the learning, especially early in training when the model's own predictions may be unreliable. The loss function is the categorical cross-entropy between the predicted sequence and the true sequence of letters, applied at each output position. This loss effectively measures how well the model's predicted letters match the expected letters for every time step in the output sequence.

Training such a model on a large dataset is computationally intensive, so we monitored performance on a validation subset throughout the process. Batches of sequence data are fed to the model on a GPU, and after each epoch (or even during an epoch), we evaluated the model's translation accuracy on validation examples. Instead of simple accuracy per character, we track an **edit-distance based metric** that reflects sequence prediction quality – essentially counting the number of insertions, deletions, or substitutions needed to turn the model's output into the correct output. This gives a more informative view of how close the model comes to spelling the entire word correctly. A perfect score means the entire word was predicted exactly. Monitoring this metric during training helps ensure the model is truly learning to spell whole words correctly, rather than just getting most letters right but in wrong positions. We continued training for multiple epochs until the error on the validation set stopped improving significantly. Throughout training, techniques like learning rate scheduling were used (for example, starting with a relatively higher learning rate and then reducing it as training progresses) to ensure stable convergence of the Transformer model.

Inference Procedure: Once training was complete, the model can be used to infer text from new sequences of ASL finger movements. For an offline evaluation, we took sequences of landmarks that the model has not seen before (from a test set) and ran them through the model to produce predicted text. The inference works as follows: the encoder processes the entire sequence of input landmarks, and then the decoder generates letters one by one, each time taking into account the previously generated letters. We implemented a greedy decoding approach for simplicity: the model outputs the most probable next character at each step, which is then fed back in as input for the decoder to predict the following character, continuing until an end-of-sequence token is produced or the maximum output length is reached. This yields the spelled-out word the model believes the input sequence represents. We then compare this predicted word to the ground truth. In a deployed real-time scenario (beyond this implementation-focused scope), the same process would be applied continuously on a stream of incoming video frames – extracting landmarks live, feeding them into the Transformer, and updating the displayed text output as new letters are recognized.

In summary, the methodology combines **data-centric preprocessing** and a **Transformer-based learning approach**. By preparing high-quality sequential data (via landmark extraction and normalization) and using a model adept at sequence translation, the implemented system learns to accurately map complex hand motion patterns to written language. The next sections detail the model architecture and the implementation specifics, followed by evaluation results.

Model Architecture

Figure: The Transformer encoder–decoder architecture used in this project. The encoder (left) processes the sequence of input frames (hand landmarks over time) and converts it into a learned internal representation. The decoder (right) then generates the output sequence of characters one at a time, using both the encoder's output and its own previously generated characters to inform each step. The Transformer's self-attention mechanism (illustrated by the connections within each block) allows the model to focus on relevant parts of the sequence when producing each letter.

The implemented model follows the standard Transformer architecture, consisting of an **encoder** and a **decoder** designed for sequence-to-sequence tasks. Below, we outline the key components and how they are tailored to the fingerspelling recognition problem:

- **Encoder:** The encoder takes as input the sequence of preprocessed landmark frames. Each frame in the sequence is represented by a feature vector (for instance, the concatenated x, y, z coordinates of all selected landmarks for that frame). Since Transformers operate on sets of vectors, we first embed these input features into a higher-dimensional space. In practice, this can be done by a learned linear projection or feed-forward layer that maps the raw coordinate vector into an embedding vector of desired size (e.g., 128 or 256 dimensions). After embedding, a **positional encoding** is added to each frame's vector to inject information about the order of the sequence (so the model knows which landmark vector is early in the sequence and which comes later). We used sinusoidal positional encodings as described in the original Transformer design, which provide a unique set of values for each time step and generalize to sequence lengths not seen during training.

The encoder is composed of multiple layers of self-attention and feed-forward sublayers. In each encoder layer, a **multi-head self-attention** mechanism allows the model to weigh the importance of different frames in the sequence relative to each other. For example, when encoding the hand shape at frame 10, the self-attention might learn that frame 9 and frame 11 (the neighboring frames) are highly relevant (since they show the transition into and out of the shape at frame 10), and perhaps frame 5 is less relevant. Multi-head attention means this process is repeated in parallel with multiple sets of learned attention weights, enabling the model to attend to different aspects of the sequence simultaneously (such as short-term transitions with one head and longer-term dependencies with another). Following the attention, each layer has a point-wise feed-forward network (a simple multi-layer perceptron) that further transforms each position's representation, and residual connections plus normalization to help training. Stacking several such layers (for instance, 4 or 6 layers) gives the encoder considerable depth and capacity to represent complex sequences. After the final encoder layer, we have an encoded representation of the entire input sequence, where each time step's vector now encodes information about that frame *in context* of the whole sequence.

- **Decoder:** The decoder is responsible for generating the output text sequence (the spelled word), one character at a time. It too consists of multiple layers, each containing a multi-head self-attention sublayer, a multi-head *cross-attention* sublayer, and feed-forward sublayer (with similar residual and normalization structure). The decoder operates autoregressively: at the time of predicting, say, the 3rd character, it only has access to the

1st and 2nd characters of the output (not the 4th or later, which are not yet predicted). This is enforced during training by a masking in the decoder's self-attention – the decoder's self-attention looks at the previously generated characters but not future ones. During training, when the full target sequence is known, we apply a mask so that position t in the output can only attend to positions $< t$. This ensures the model learns to predict the next character using only past context.

The decoder's cross-attention sublayer is what connects it to the encoder. After computing a self-attended representation of the output sequence generated so far, the decoder performs attention over the encoder's output. This means for each potential output character, the decoder can focus on those frames in the input sequence that are most relevant to predicting that character. For example, if the model is about to output the letter "C", it might attend strongly to a subset of input frames where the hand was in the "C" shape. Multi-head cross-attention allows it to consider various parts of the input simultaneously for each output step. Finally, the feed-forward sublayer in the decoder further processes the information. At the end of the decoder stack, we add a final linear layer and a softmax activation to produce a probability distribution over the output vocabulary for the next character. The output vocabulary in this case consists of the 26 English letters (A–Z), plus a special end-of-sequence token (and possibly a padding token for internal use). The decoder thus predicts one letter at a time until it outputs the end-of-sequence token, which signals that the word is complete.

- **Positional Encoding:** As mentioned, positional encodings are added to the input of both the encoder and decoder to provide sequence order information (since the self-attention alone has no sense of position, it treats the input as a set). We implemented the standard sinusoidal positional encoding that the original Transformer uses: a set of deterministic values based on sine and cosine functions of different frequencies, for each position index. These values are added to the embeddings at the bottom of the encoder and decoder. The effect is that each position in the sequence has a unique signature, allowing the model to learn relative positions or distances between frames and between output characters. This is important for fingerspelling because the order of hand shapes matters – the sequence "H-A-T" is very different from "T-A-H" even if the same letters are involved. The positional encoding ensures the model can distinguish and make use of the order.

In summary, the model architecture enables the system to take a **variable-length sequence of coordinates** and output a **variable-length sequence of letters**. The self-attention layers let the encoder make sense of the entire gesture sequence globally, and let the decoder consider the entire prefix of output generated so far. The design is fully non-recurrent, meaning the model processes elements in parallel within each attention layer, which is efficient on modern hardware and allows using long sequences (multiple dozens of frames) effectively. For this project, we configured the Transformer with a moderate number of layers and hidden dimensions to balance performance with the limited training time. Even a relatively compact Transformer can learn the fingerspelling translation well given the high-quality data. The next section will discuss how we implemented and trained this model in practice, including tools and techniques used.

Implementation

The system was implemented in Python using well-established machine learning and computer vision libraries. **TensorFlow 2** (with Keras) was chosen for building and training the Transformer model, due to its easy handling of sequence data and powerful high-level API for defining complex architectures. The ASL fingerspelling dataset (provided via Kaggle) was large, so we took advantage of efficient data loading utilities and performed preprocessing in a streaming fashion – reading landmark data batch by batch, transforming it, and feeding it to the model training loop on the fly to avoid storing everything in memory at once.

Data Pipeline Implementation: We wrote a data loader that reads the dataset's landmark files (which are stored in a compact format, Parquet, containing coordinates for each frame). For each training sample (a sequence of frames and corresponding label), the loader applies the normalization steps (centering and scaling as described in Methodology). This was implemented with NumPy for numeric operations. After normalization, we assemble the sequence into a fixed-length tensor, padding with zeros for frames beyond the actual sequence length. Alongside, we prepare the target output sequence: the letters of the word are converted to numerical indices (0–25 for A–Z, for example) and a special end token is appended. This target sequence is also padded to a fixed maximum length (the maximum word length plus one for the end token). We generate an attention mask array to indicate which parts of the input and output are real vs. padding, so the model will ignore pad positions. These prepared inputs (landmark sequence, mask) and outputs are then fed into the model. We implemented this pipeline as a Python generator function so that it could yield training batches to TensorFlow's `model.fit` in a streaming manner, avoiding the need to hold the entire dataset in memory. This generator also performed random shuffling of the sequence order and could be configured to return a validation batch when needed.

Model Implementation: The Transformer model was built using Keras' functional API. We implemented custom layers for certain components: - We created an `EncoderLayer` class for one layer of the encoder (with its multi-head self-attention and feed-forward network) and an analogous `DecoderLayer` class. For the multi-head attention, we leveraged TensorFlow's `MultiHeadAttention` layer, which simplifies the creation of attention heads and queries/keys/values calculation. We configured it with the desired number of heads (e.g., 8 heads) and the size of each head. We also wrote a small function to generate the sinusoidal positional encoding matrix for a given maximum sequence length and embedding dimension, and added that to the input embeddings. - For the output, we used a final `Dense` layer with softmax to project the decoder's output to letter probabilities. The vocabulary size was 27 (letters plus end-of-sequence; since case or other symbols were not needed, we kept it simple). - We combined these pieces: an **Encoder** class that contains a loop to stack multiple `EncoderLayers`, and similarly a **Decoder** class. The overall model was then built by connecting the Encoder's output to the Decoder. Keras' `Model` was defined with three inputs: the source sequence, a boolean mask for source (to mask pad frames), and the target sequence (for teacher forcing during training) along with a target mask. The output is the sequence of predicted character probabilities for each target time step.

One challenge in implementation was handling the masking properly so that the model doesn't treat padded frames or future characters as valid inputs. We ensured that: - The padding mask was fed into the encoder's attention layers so they do not pay attention to padded positions. - The

decoder used two masks: one for padding in the target (so it doesn't attend to its own padding positions), and a causal mask to not violate the autoregressive training (prevent looking ahead in the output sequence). TensorFlow's MultiHeadAttention layer allows us to pass these masks in each call.

We configured the **optimizer** as Adam, which is well-suited for training Transformers, with an initial learning rate and used a learning rate schedule that warms up then decays (a common practice for Transformers to stabilize training). In code, this was done by defining a custom learning rate schedule callback or by manually adjusting it per epoch. The **loss function** was categorical cross-entropy applied to the output sequences. Because of padding in outputs, we applied the loss only where the target mask indicates a real character (so the padded positions don't contribute to loss). TensorFlow's sequence modeling utilities helped here, or we manually multiplied the loss by the target mask.

Training Process: We trained the model on a GPU-enabled environment to handle the large dataset. The training script was instrumented with TensorBoard to log training and validation loss over time, as well as the edit-distance metric we computed for validation sequences. To compute the edit-distance (or an approximation of word accuracy), after each training epoch we took a subset of validation data, ran the model in inference mode (greedily generating outputs without teacher forcing), and computed the Levenshtein distance between the predicted and true spelling of each word. We then summarized this as an average Character Error Rate (CER) or simply counted what fraction of words were spelled entirely correct. These metrics were printed for monitoring but not directly used to optimize the model (they were for our assessment only; the model was still optimizing cross-entropy on the training set).

Despite the timeframe constraint for this project, the implementation managed to train the Transformer to a good level of accuracy. By the end of training, the model could reliably translate sequences of hand movements into the correct word for a majority of cases in the test data. We found that the attention mechanism indeed learned to focus on the appropriate frames for each letter. For example, when the model predicted the word "HELLO", analysis of the attention weights showed it concentrated on a distinct set of frames when generating each letter, corresponding to when the signer's hand formed that letter shape in the input sequence.

It's worth noting that we did not implement a full user interface or deploy the model to a live camera feed within this scope. However, we verified the model's functionality on recorded video sequences. The translation output was printed to the console for a sample of test videos to demonstrate end-to-end working: the video's landmark sequence is fed in and the spelled text comes out. This confirms that all components – landmark processing, Transformer prediction, and decoding – work together correctly.

Evaluation

Evaluation of the implemented system was conducted on held-out test sequences from the ASL fingerspelling dataset. The primary goal was to assess how accurately the model translates unseen fingerspelling videos into text. Because the output is a sequence of characters forming a word, we use sequence-based evaluation metrics rather than per-frame accuracy. Specifically, we measured the **edit distance** between the model's predicted sequence and the true sequence for each test sample. The edit distance (Levenshtein distance) counts the minimum number of

single-character edits required to change the prediction into the correct answer. From this, we derived metrics such as the average Character Error Rate (the edit distance divided by the word length, averaged over the test set) to quantify performance. A low error rate indicates that the model's output is very close to the true spelling on average, with few mistakes.

In qualitative terms, the Transformer model demonstrates strong performance in recognizing fingerspelled words. For most test examples, the sequence of letters output by the model exactly matches the ground-truth word. The model is able to correctly identify subtle differences in hand shape and movement that distinguish letters. For instance, in ASL the letters "M" and "N" have very similar hand shapes (both involve placing the fingers on the thumb, differing in the number of fingers used). In continuous spelling, these can be easily confused by a naive classifier. However, our system often disambiguates such cases by looking at context – if an "M" is followed by an "O" in a word, the model learns the transitional motion from "M" to "O", whereas "N" followed by "O" might involve a slightly different transition. As a result, the Transformer produced the correct letter in context where a frame-by-frame approach might falter.

For a concrete example, consider an input sequence representing the word **"TEAM"** being fingerspelled. This is a four-letter word, meaning the input video had a person signing T → E → A → M in sequence. The model's output for this test case was "TEAM" exactly, with no errors. It was observed that the model particularly benefited from the sequence approach in the transition between "E" and "A" (two letters that involve distinct hand orientations). Even if a few frames of the letter "A" were visually ambiguous, the model recognized the overall pattern of going from E to A to M and produced the correct letters. On the other hand, in some failure cases the model output letters that formed a different word or a nearly correct word. For example, one test sequence's true label was "CARD", and the model predicted "CARDS" (with an extra "S" at the end). In this case the model detected an "S" gesture where perhaps the signer's motion at the end was interpreted as an S. Such errors highlight that while the model is largely accurate, it can occasionally insert or miss a letter, which is why using the edit distance metric is appropriate to capture these off-by-one mistakes.

Overall, the evaluation shows that the implemented system successfully learns the mapping from continuous ASL fingerspelling to English text. The error rates were reasonably low, and importantly, nearly all errors were minor (such as an extra or missing letter, as in the example above) rather than completely incorrect words. This indicates that the model has a solid grasp of the fingerspelling alphabet and sequence structure. Another positive sign is that the system generalized across different signers: the test set included people whose signing styles or speeds were not seen during training, yet the model was still able to interpret their fingerspelling correctly in most cases. This can be attributed to the large and diverse training set and the model's use of robust features (landmarks and normalized coordinates), which together ensure good generalization.

It's worth mentioning that we did not compute a traditional word accuracy or word error rate (WER) in this implementation report, since our focus was on getting the system working and demonstrating correct outputs rather than benchmarking against competition-grade metrics. However, the qualitative and character-level evaluations we performed strongly suggest that the model's performance is competitive and that it produces accurate translations for continuous fingerspelling input.

Conclusion

In this project, we implemented a complete pipeline for translating ASL fingerspelling into text using a Transformer-based model. By focusing on what was concretely developed – from data preprocessing through model training and testing – we demonstrated the viability of a modern sequence-to-sequence approach for sign language interpretation. The use of **MediaPipe landmarks** proved to be a crucial design choice, simplifying the input representation and making the learning task tractable within a limited development timeframe. This allowed us to sidestep the complexities of raw image processing and devote our efforts to sequence learning, which the Transformer excels at.

The resulting system can take a sequence of hand movements (captured via a standard camera and converted to keypoint data) and output the corresponding English word that was spelled. This is achieved with high accuracy, as verified by our evaluations on unseen data. The Transformer model effectively learns the implicit "language" of fingerspelled sequences – understanding not only individual letter signs but how they connect in a continuous flow. Technically, this work demonstrates how attention mechanisms and sequence modeling can be applied beyond text or speech, extending into the domain of visual-sign languages.

In conclusion, the implemented ASL fingerspelling-to-text system underscores the power of combining advanced neural architectures with well-chosen intermediate representations (skeletal landmarks) for solving a challenging real-world pattern recognition problem. The project delivered a functional proof-of-concept that focuses solely on the core translation task. It establishes a foundation that could be built upon for a fully interactive application, but even in its current form, it serves as a compelling demonstration of translating visual signed input to written language using a Transformer. The success of this implementation opens the door for further developments in sign language technology, where such sequence models could be expanded to larger vocabularies and integrated into communication tools for the Deaf community.
