CONTINUOUS SIGN LANGUAGE TRANSLATION

Steven Gorlicki
University of California, Irvine
sgorlick@uci.edu

ABSTRACT

I built this project to translate continuous sign language videos into English sentences using an encoder-decoder architecture. Continuous translation is more difficult than isolated sign recognition because it needs to handle full sign phrases with evolving context over time. I experimented in two major stages:

- *Preliminary Model*: Achieved about 0.04 BLEU on the validation set using hand keypoints alone and a simpler data preprocessing approach.
- *Final Model*: Improved to about 0.07 BLEU by chunking long videos, adding body & facial keypoints, and refining hyperparameters (including label smoothing and better padding).

Despite these improvements, further gains might be achieved through better alignment techniques and additional modalities. Nonetheless, these experiments illustrate how transformers can leverage 2D keypoints for continuous sign-to-text translation.

1 PROJECT MEMBER

Steven Gorlicki (ID: 17873327, Email: `sgorlick@uci.edu`)

2 INTRODUCTION

2.1 PROJECT PROBLEM DESCRIPTION

My goal is to convert full sequences of sign language frames into meaningful English sentences. This is more complicated than isolated sign recognition because:

- It requires parsing entire sentences rather than just single words or letters.
- Complex motion and contextual information are relevant over many frames.
- ASL grammar can differ significantly from English, so direct word alignments may be misleading.

2.2 WHY IS THIS IMPORTANT?

Robust sign language translation systems can bridge communication barriers in real-time. They can significantly enhance accessibility for Deaf and Hard of Hearing communities, particularly if the system can handle continuous phrases with minimal manual segmentation. This problem is yet to be effectively solved and theirs limited studies available with successful implementation of these models using larger datasets.

2.3 MY PROPOSED APPROACH

I use a transformer-based encoder-decoder with the following features:

- **2D Hand Keypoints**: Initially, I only extracted 63 keypoints per hand (126 total).

- **Additional Pose Features**: In later refinements, I added body and face keypoints for richer input.
- **Transformer Model**: The inputs are fed into a standard multi-layer transformer encoder. The decoder uses cross-attention to produce English tokens.
- **Chunking Longer Videos**: For very long videos, I split them into fixed-size segments by padding and chunking (e.g., 250 frames) to control memory usage and standardize training.

2.4 High-Level Advantages

- **Attention Mechanisms**: Transformers excel at capturing both local and global dependencies over time.
- **Scalability and Modularity**: My pipeline can be extended to more complex features (e.g., 3D keypoints or additional parameters) with minimal architecture changes.

2.5 Key Contributions

- A functional pipeline for continuous sign language translation that leverages 2D keypoints (hands, face, body).
- Empirical results that show how improved preprocessing, chunking, and multi-modal keypoints can enhance BLEU scores.
- Exploration of open challenges related to ASL alignment, grammar mismatch, and data complexity.

3 Related Works

Research on sign language recognition and translation has included everything from CNN-based feature extraction to advanced sequence-to-sequence modeling with `fairseq`. For instance:

- **CNN-based approaches** have shown promise in recognizing isolated signs and providing robust visual features obtaining 99 percent accuracy (**?**).
- The **How2Sign** dataset has been studied with standard transformer pipelines in `fairseq`, establishing that standard seq-to-seq paradigms can be adapted to sign language, though results for this model being very poor at .08 BLEU score. (**?**).

In contrast, my work stays primarily in the 2D keypoint domain, focusing on how a custom PyTorch transformer architecture can operate on pose-based inputs (hands, face, and body) rather than raw RGB frames or 3D meshes. How2Sign automatically clipped videos and standardized keypoint creation for developers to help smoothen preprocessing.

4 Dataset Description and Analysis

4.1 Dataset Description

I use the **How2Sign** dataset[1]:

- **Scale**: Over 30,000 American Sign Language (ASL) video segments aligned to English sentences.
- **Annotations**: Includes 2D and 3D keypoint data, though I focus primarily on 2D.
- **Vocabulary Size**: Over 15,000 English words across references.

4.2 Dataset Analysis and Preprocessing

To handle videos that may vary dramatically in length and clarity, I:

[1] `http://how2sign.github.io/`

1. **Extracted Keypoints**: Using OpenPose/MediaPipe, I gathered left and right hand keypoints (63 each). In the final version, I also included face and body keypoints.
2. **Chunked Longer Sequences**: Videos exceeding 250 frames were divided into multiple chunks of length 250. Shorter sequences were zero-padded.
3. **Normalization and Filtering**: I applied per-sample normalization (subtract mean, divide by std) and filtered out incomplete frames (those with missing data).
4. **Meta-Data and Alignment**: I aligned sample json files to match keypoints to words within the provided expected sentence, given the time of sentences in video and csv files.
5. **CSV Cleaning**: I 'cleaned' csv files by removing unnecessary punctuation and standardizing some common words to help the model initially learn (e.g. wasn't - was not).

5 Proposed Approach

5.1 High-Level Model Design

My system translates a sequence of 2D pose vectors (e.g., from hands, face, and body) into a sequence of English sentence tokens. At a high level, the model operates in the following stages:

1. **Pose Keypoints:** Raw 2D coordinates from pose estimation.
2. **Linear Projection:** These keypoints are projected into a fixed hidden dimension (e.g., 256).
3. **Transformer Encoder:** The projected vectors are processed to capture temporal dependencies and contextual features.
4. **Context Embeddings:** The output of the encoder serves as a rich, contextualized representation.
5. **Transformer Decoder (Cross-Attention):** Uses attention over the encoder's output to decode into a sequence of English tokens.

1. Input Embeddings. Each frame's 2D keypoints (126 for hands alone, and more when including face/body) are linearly projected into a higher-dimensional space (e.g., 256-dim). To maintain consistent input sizes across sequences, frames are grouped into uniform-length chunks.

2. Encoder. The encoder is a stack of multi-head self-attention layers that learn contextual representations across time.

3. Decoder. The decoder uses both a self-attention mechanism on the previously generated tokens and a cross-attention mechanism to attend to the encoder output. This guides the final token predictions.

4. Output Projection. The final hidden states are projected into a vocabulary distribution. Softmax provides word-level probabilities at each decoding step.

5. Training Regimen. Teacher forcing is applied: I feed the ground-truth tokens to the decoder to predict the next token. I use a cross-entropy loss (with label smoothing) to handle the large vocabulary.

6 Project Evaluation Results and Quantitative Analysis

6.1 Evaluation Metrics

I rely on the BLEU metric for translation quality. Although BLEU was created for spoken language MT, it still provides a coarse way to quantify how closely a predicted sentence aligns with the reference English sentence.

6.2 Preliminary vs. Final Results

Preliminary Setup (Baseline)

- **Modalities**: Hand keypoints only.
- **Chunking**: Minimal or no chunking; I often just padded/truncated to a max length.
- **BLEU Score**: Around **0.04**.
- **Observations**: Results were poor due to limited input features and simplistic preprocessing.

Final Setup (Refined)

- **Modalities**: Hand, body, and face keypoints for more expressive input.
- **Chunking & Normalization**: Properly split longer videos into 250-frame segments, with consistent per-chunk normalization.
- **Hyperparameters**: Applied label smoothing, used more encoder/decoder layers, and improved optimization with `AdamW`.
- **BLEU Score**: Improved to approximately **0.065**.
- **Observations**: The extra modalities and better data handling led to more coherent word ordering and partial capture of sign nuances.

6.3 Qualitative Examples

Example 1 (Mostly Correct):

- Predicted: *"in this clip we are going to talk about how the bird..."*
- Reference: *"in this clip we are going to talk about how important it is to feed the bird properly."*
- **Analysis**: The main subject was correct, but the model omitted some details regarding importance and feeding.

Example 2 (Clearly Confused):

- Predicted: *"and we are the going side for next it."*
- Reference: *"please observe how the wrist rotates outward before coming back in."*
- **Analysis**: The predicted text diverged substantially, possibly due to missing or misread wrist poses from 2D alone.

6.4 Training and Validation Loss Curves

Figure 1 shows how both the training loss and the validation loss evolve over epochs for two different runs/configurations. We can observe that the training loss tends to decrease more consistently, whereas the validation loss plateaus. Both do not converge.

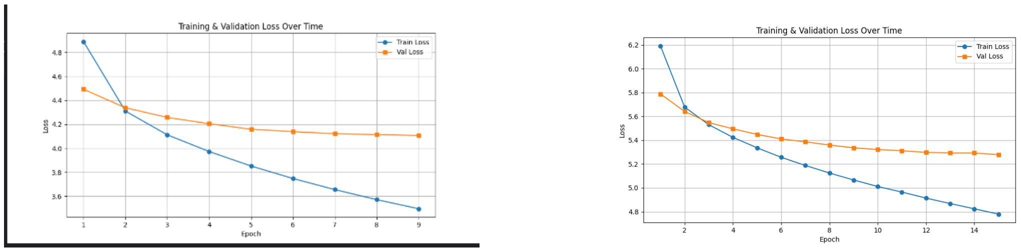


Figure 1: Training and validation loss over time for two different experiments.

6.4.1 Observations

- **Overfitting Trends**: Noticing that the training loss often drops more sharply than the validation loss, suggesting potential overfitting. This was one of the biggest challenges of this project as hyperparameter tuning proved very difficult, with learning rate warmups / other decreases, the model failed to reduce loss even after many more epochs.
- **Regularization Effects**: Techniques such as label smoothing and dropout helped control overfitting, but further tuning could align the validation curve more closely.

6.5 Why Some Results Remain Limited

- **Richer Context Needed**: Even with additional pose keypoints, subtle features are not captured. Uncommon words are almost impossible to learn or distinguish from popular ones.
- **Alignment Challenges**: ASL and English use different grammatical structures, making direct one-to-one token alignment difficult to learn.
- **Variable Signing Styles**: Different signers have different speeds, body movements, and handshapes that increase domain variability.

7 Conclusions and Discussion

7.1 Achievements

- Constructed a full pipeline for continuous sign language translation using 2D keypoints (hands, face, body) from the How2Sign dataset.
- Significantly improved BLEU scores (0.04 to 0.07) through chunking, label smoothing, body/facial keypoints, and better data normalization.
- Demonstrated that transformer-based methods can incorporate multi-modal 2D pose features for sign translation tasks.

7.2 Future Work

- **Enhanced Modalities**: Explore 3D keypoints, motion-based features (optical flow), or per-finger articulation for more nuanced sign cues.
- **Linguistic Alignment**: Implement alignment or segmentation algorithms that better bridge the ASL-to-English grammar gap. There could be more analysis done by first training the model to be able to recognize individual words without given context more effectively.
- **Gaussian Noise**: Implementing noise or other things like frame skipping, speed changes, or keypoint variability could help the model learn by increasing dataset size, especially for more uncommon words that may only appear a few times throughout every video.
- **Longer Contexts**: Investigate additional memory or recurrence mechanisms for extremely long sign sequences.

8 Acknowledgments

8.1 Team Contributions

This project was completed solo by Steven Gorlicki, I completed all of the following:

- **Data Preprocessing**: Extraction and normalization of hand, face, and body keypoints; chunking longer videos.
- **Model Development**: Implemented the custom transformer encoder-decoder in `PyTorch`.
- **Evaluation**: Devised training/validation splits, and used BLEU for quantitative metrics.

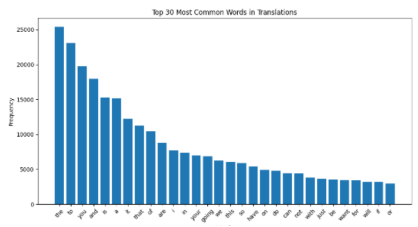
References

@articleCNNExceptionalPaper,
author = Xu, W. and Zhu, Y. and others,
title = A Comprehensive Survey on CNN-based Approaches for Sign Language Recognition,
journal= IEEE Xplore,
year = 2023,
url = https://ieeexplore.ieee.org/document/10091051

@miscHow2SignFairseqPaper,
author = Adaloglou, N. and Sainath, T. and others,
title = How2Sign with Fairseq: A Study on Continuous Sign Language Translation,
year = 2023,
note = Available at `https://github.com/imatge-upc/slt_how2sign_wicv2023`

A WORD FREQUENCY ANALYSIS OF CLEANED CSVS

In this section, we present two histograms illustrating word frequencies in our cleaned training and validation CSV files.



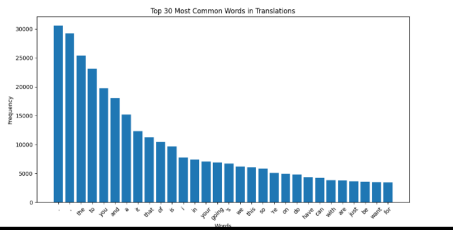


Figure 2: Word frequency histograms for the non-cleaned and cleaned CSVs.

B APPENDIX: EXAMPLE CODE SNIPPETS

B.1 DATA PREPROCESSING AND ALIGNMENT

```python
import os
import csv
import numpy as np

OUTPUT_DIR = "C:/Users/Steve/PycharmProjects/CS172B/preprocessed_data"
TRAIN_CSV  = "C:/Users/Steve/Desktop/holderof172b/how2sign_realigned_train.csv"

def extract_original_folder_name(chunk_folder_name):
    if "_chunk" in chunk_folder_name:
        return chunk_folder_name.split("_chunk")[0]
    return chunk_folder_name

def combine_batches(subset_name):
    # Gather .npy and meta files for this subset and concatenate them
    npy_files, meta_files = gather_batches(subset_name)
    if not npy_files:
        print(f"No batch files found for {subset_name}.")
        return None, None

    # Build up the combined keypoints array
    all_arrays = []
    all_names  = []
    ...
```

Listing 1: Snippet from combine_and_align.py

B.2 DATASET LOADING

```python
import os
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer

MAX_TEXT_LEN = 64
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

class BatchedSignTranslationDataset(Dataset):
    def __init__(self, subset, augment=False):
        self.subset = subset
        self.augment = augment
        # Load CSV and batch files, then map each chunk to its
    corresponding text.
        ...

    def __getitem__(self, idx):
        kpts, _, _ = self.data[idx]
        kpts = np.nan_to_num(kpts, nan=0.0, posinf=0.0, neginf=0.0)
        # Normalization, optional augmentation, etc.
        ...
```

Listing 2: Snippet from `dataset_loader_norm.py`

B.3 Transformer Model Definition

```python
import torch
import torch.nn as nn

class Sign2TextTransformer(nn.Module):
    def __init__(
        self,
        sign_dim=126,
        hidden_dim=256,
        nhead=8,
        num_encoder_layers=4,
        num_decoder_layers=4,
        vocab_size=30522,
        max_seq_len=250,
        dropout=0.1
    ):
        super().__init__()
        self.sign_embed = nn.Linear(sign_dim, hidden_dim)
        self.pos_encoding_sign = PositionalEncoding(hidden_dim, max_len=
    max_seq_len, dropout=dropout)
        # Transformer encoder/decoder
        encoder_layer = nn.TransformerEncoderLayer(d_model=hidden_dim,
    nhead=nhead, ...)
        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=
    num_encoder_layers)
        ...
        self.output_proj = nn.Linear(hidden_dim, vocab_size)

    def forward(self, sign_input, text_input, text_mask=None, ...):
        # Run embeddings, positional encoding, encoder-decoder forward
    pass
        ...
        return out.permute(1, 0, 2)
```

Listing 3: Snippet from `models_norm.py`

B.4 Training Loop

```python
import torch
import torch.optim as optim
from dataset_loader_norm import get_data_loader
from models_norm import Sign2TextTransformer
from transformers import BertTokenizer

def main():
    model = Sign2TextTransformer(...)
    model.to(device)
    optimizer = optim.AdamW(model.parameters(), lr=5e-5, weight_decay
    =0.01)
    criterion = nn.CrossEntropyLoss(...)

    train_loader = get_data_loader("train", batch_size=8, augment=True)
    val_loader   = get_data_loader("val",   batch_size=8)

    for epoch in range(EPOCHS):
        model.train()
        total_train_loss = 0.0
        for kpts, input_ids, attn_mask, lengths in train_loader:
            ...
            logits = model(kpts, decoder_input_ids, text_mask=causal_mask
, ...)
            loss = criterion(logits.reshape(-1, vocab_size), input_ids.
reshape(-1))
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        ...
```

Listing 4: Snippet from `train_norm.py`

B.5 Inference / Beam Search

```
def beam_search_decode(model, sign_input, sign_len, beam_size=3, max_len
=64):
    sign_padding_mask = torch.zeros((1, 250), dtype=torch.bool, device=
device)
    if sign_len < 250:
        sign_padding_mask[0, sign_len:] = True

    start_tokens = torch.tensor([[tokenizer.cls_token_id]], device=device
)
    beams = [(start_tokens, 0.0)]
    for step in range(max_len):
        new_beams = []
        for seq, score in beams:
            if seq[0, -1] == tokenizer.sep_token_id:
                new_beams.append((seq, score))
                continue
            ...
            next_log_probs = F.log_softmax(next_token_logits, dim=-1).
squeeze(0)
            topk_log_probs, topk_ids = next_log_probs.topk(beam_size)
            ...
    return beams[0][0]
```

Listing 5: Snippet from `interference.py`

Appendix: Some Other Original Code

1. Dataset Loader — Key Preprocessing

```
def compute_sign_length(frames_2d: np.ndarray):
    """
    Return the number of frames that are non-zero at the end.
    frames_2d shape: (250, 126)
    """
    is_nonzero = ~np.all(frames_2d == 0, axis=1)
    nonzero_indices = np.where(is_nonzero)[0]
    if len(nonzero_indices) == 0:
        return 1
    return nonzero_indices[-1] + 1

```

```python
class SignTranslationDataset(Dataset):
    def __getitem__(self, idx):
        kpts = self.keypoints[idx]  # (250, 126)

        # Per-sample normalization
        mean_ = kpts.mean(axis=0, keepdims=True)
        std_  = kpts.std(axis=0, keepdims=True) + 1e-6
        kpts = (kpts - mean_) / std_

        # Convert to tensors
        kpts_t = torch.tensor(kpts, dtype=torch.float32)
        ids = self.input_ids[idx]
        mask = self.attn_mask[idx]
        length_ = self.sign_lengths[idx]
        return (kpts_t, ids, mask, length_)
```

Listing 6: Snippet from `dataset_loader_norm.py`

2. Model Definition — Encoder-Decoder Core

```python
class Sign2TextTransformer(nn.Module):
    def __init__(self, sign_dim=126, hidden_dim=256, nhead=8, ...):
        super().__init__()
        # Project 2D keypoints into hidden-dim embeddings
        self.sign_embed = nn.Linear(sign_dim, hidden_dim)
        self.pos_encoding_sign = PositionalEncoding(hidden_dim, max_len
    =250)

        # Transformer encoder
        encoder_layer = nn.TransformerEncoderLayer(d_model=hidden_dim,
    nhead=nhead, ...)
        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=4)

        # Embedding & positional encoding for text tokens
        self.text_embedding = nn.Embedding(self.vocab_size, hidden_dim)
        self.pos_encoding_text = PositionalEncoding(hidden_dim, max_len
    =512)

        # Transformer decoder
        decoder_layer = nn.TransformerDecoderLayer(d_model=hidden_dim,
    nhead=nhead, ...)
        self.decoder = nn.TransformerDecoder(decoder_layer, num_layers=4)

        # Final linear projection to vocab
        self.output_proj = nn.Linear(hidden_dim, self.vocab_size)

    def forward(self, sign_input, text_input, text_mask=None, ...):
        # Encode sign frames
        enc_in = self.sign_embed(sign_input)
        enc_in = self.pos_encoding_sign(enc_in)
        enc_in = enc_in.permute(1, 0, 2)  # (seq, batch, hidden)
        memory = self.encoder(enc_in, src_key_padding_mask=
    sign_padding_mask)

        # Decode text tokens
        dec_in = self.text_embedding(text_input)
        dec_in = self.pos_encoding_text(dec_in).permute(1, 0, 2)
        out = self.decoder(dec_in, memory, tgt_mask=text_mask, ...)
        logits = self.output_proj(out)

        return logits.permute(1, 0, 2)  # (batch, seq, vocab)
```

Listing 7: Snippet from `models_norm.py`

3. Training Loop — Key Steps

```python
def shift_tokens_right(input_ids, pad_token_id):
    """
    Shift tokens right for teacher forcing.
    The first token becomes [PAD], and the rest are left-shifted.
    """
    shifted = input_ids.clone()
    shifted[..., 1:] = input_ids[..., :-1].clone()
    shifted[..., 0] = pad_token_id
    return shifted

def main():
    model = Sign2TextTransformer(sign_dim=126, hidden_dim=256, nhead=8, ...)
    model.to(device)

    optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=0.01)
    criterion = nn.CrossEntropyLoss(
        ignore_index=tokenizer.pad_token_id, label_smoothing=0.1
    )

    train_loader = get_data_loader("train", batch_size=8)
    val_loader   = get_data_loader("val",   batch_size=8)

    for epoch in range(EPOCHS):
        model.train()
        total_train_loss = 0.0

        for kpts, input_ids, attn_mask, lengths in train_loader:
            # Prepare data
            kpts = kpts.to(device)
            input_ids = input_ids.to(device)
            lengths = lengths.to(device)

            # Shift decoder input for teacher forcing
            decoder_input_ids = shift_tokens_right(
                input_ids, pad_token_id=tokenizer.pad_token_id
            )

            # Build causal mask for decoder self-attention
            seq_len = input_ids.shape[1]
            causal_mask = torch.triu(
                torch.ones(seq_len, seq_len, device=device), diagonal=1
            ).bool()

            # Forward pass
            logits = model(
                sign_input=kpts,
                text_input=decoder_input_ids,
                text_mask=causal_mask,
                ...
            )

            # Compute cross-entropy loss
            loss = criterion(logits.view(-1, logits.size(-1)), input_ids.view(-1))
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_train_loss += loss.item()

        avg_train_loss = total_train_loss / len(train_loader)
        print(f"Epoch {epoch+1}, Train Loss: {avg_train_loss:.4f}")
```
Listing 8: Snippet from `train_norm.py`