Project Report : CS 7643

Matthew So
Georgia Institute of Technology
mso31@gatech.edu

Bryan Zhou
Georgia Institute of Technology
bzhou83@gatech.edu

Hailey Ho
Georgia Institute of Technology
nho38@gatech.edu

Steven Le
Georgia Institute of Technology
shoang33@gatech.edu

Abstract

The demand for applications that can recognize the American Sign Language (ASL) are high in the today's Deep Learning (DL) community. Previous attempts to optimize sign language recognition have much to be desired. With the introduction of Google's Isolated Sign Language Recognition competition dataset that utilizes landmarks generated via MediaPipe, an opportunity arose to compare different approaches, including new and existing architectures, to gain insight into the performances of different DL approaches on a new batch of sign language data. To this end, we implemented multiple different approaches involving Long Short-Term Memory (LSTM) Networks, changing the structure, data pre-processing, and hyperparameters. We found that an encoder/decoder approach trained on a combed dataset that ignored facial landmarks, as well as a dense layers with lower amounts of encoder and LSTM nodes provided the best performance on our dataset.

1. Introduction

1.1. Background and Motivation

Sign language, the American Sign Language (ASL) in particular, is the primary tool of communication for the deaf community, as well as those who wish to communicate with family or peers who cannot communicate traditionally. For example, many deaf children are born to parents who are of normal hearing and do not understand sign language. Given that the task of learning sign language, like many other languages, is no trivial task, the need for applications that can either assist with or automate this task is high. Given the current popularity of Deep Learning (DL) in recognition tasks, we decided to investigate the use of different DL architectures and their performance in detecting sign language, using a new dataset that offers a new approach

The state-of-the-art in isolated sign language recognition uses pose estimators, handmade features, and transformers [2]. Due to data scarcity, performance on existing datasets is poor, with a recently released state-of-the-art model performing at below 70% accuracy on WLASL, a large word-level dataset collected from publicly available ASL educational and reference videos [5]. However, in CVPR 2021's isolated sign language competition, the top 3 models were able to achieve over 97.5% accuracy on AUTSL, a large isolated Turkish Sign Language dataset, using a combination of pre-trained models, regularization, pose estimation, multi-modal fusion, ensemble models, and spatio-temporal feature extraction [6]. In comparison, state-of-the-art sign language translation tends to adopt an end-to-end approach by using generalized kinetic feature extraction models such as I3D [3] to produce features used by standard transformer networks [4]. However, accuracy remains extremely poor, with the most accurate systems receiving a human rating of just 4 out of 100 in the most recent sign language translation competition at WMT 2022.

Given the aforementioned poor accuracy of previous models, there is much work that can be done to improve recognition of sign language. With the introduction of a new dataset (described in section 1.2), there arises an opportunity to test existing DL approaches with new and more detailed data. By implementing and tuning different configurations and models, we may be able to dissect and analyze aspects of certain architectures that have positive contributions to accuracy for this task and why they have the effects they do. The potential success of our project may bring about new understandings of this recognition problem as well as points of interest for future projects to focus on in order to optimize their approach for sign language recognition.

1.2. Dataset

The dataset that we use originates from Google's Isolated Sign Language Recognition Kaggle challenge, found here. The dataset consists of extracted MediaPipe features from 100,000 sign recordings with a vocabulary size of 250 across 20 users. Because of a team member's involvement with the collection of the dataset, we will also use the source videos used to generate the Kaggle challenge's data. The main detail that separates this dataset from other similar sign language datasets is the use of Google's MediaPipe in collecting videos. Mediapipe's Holistic detection utilizes a pose detection model that can split a body into key points such as hands, wrists, face, and visualize them. Using MediaPipe, the creators of this dataset at Google were able to process raw videos of hand signs and convert them into frames with landmarks (face, left hand, pose, right hand). In short, instead of having a dataset of images, the dataset includes file that contains a set of parquet files that, rather than containing intensities for each pixel in a frame, contains 3-dimensional coordinates for each landmark found in a frame. This file is accompanied by another file that consists of the filepath to each parquet as well as the corresponding sign as well as an id to credit the participant that provided their hands as data. In contrast to the commonly used MNIST sign language set that contains images of hands performing signs, we will be training our models to recognize patterns between a number of landmarks as opposed to full images. Each landmark contains (x, y, z) coordinates, although Google recommends to ignore the z value as MediaPipe is not trained to perform well with depth. The data is labelled with 250 unique words in the ASL, ranging from ”blow” and ”wet”, to longer words like ”alligator” and ”glasswindow”. After cleaning through the data (detailed in later sections) We decide to use an 80:20 split for our training and testing data.

2. Approach

2.1. Data Preprocessing

2.1.1 Landmark Selection

MediaPipe Holistic Pipeline gives us a total of 543 landmarks (468 face landmarks, 33 pose landmarks, and 21 hand landmarks per hand) per frame. A majority of these are face landmarks and irrelevant to our problem, creating a high-dimensional input that is very CPU intensive. Therefore, one of our aims is to reduce this number while not compromising the model's performance. As shown in Figure 1, some noticeable combinations that we tested include:

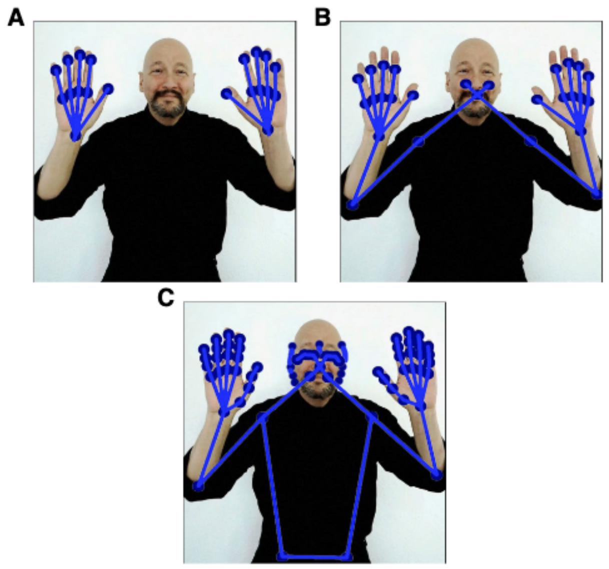

Figure 1. Different Landmark Combinations. A. `HandOnly`, B. `UpperBodyPose`, and C. `PoseNFace`

1. `HandOnly`: only the wrist, metacarpal joints, and fingertips (18 landmarks)
2. `UpperBodyPose`: `HandOnly` and some upper-body pose landmarks of the elbow, shoulder, and mouth corners (27 landmarks)
3. `PoseNFace`: include full hand, pose, and face outline (88 landmarks)

2.1.2 Data Cleaning

In order to further speed up training, we utilized a public notebook from Kaggle. It cleans the data by filling NaN values with 0 and applies batching before converts it into a Tensorflow dataset. Batching especially was helpful to circumvent issues with training time. The notebook not only makes the data more accessible and easy to code with, but also quicker to train on.

2.2. Model

We implemented an LSTM network Figure 2 using the Tensorflow Keras library. Our base model consists of an input layer combined with two dense blocks that consist of a dense layer along with a normalization layer, an activation layer, and a dropout layer. This part of our model contained our learned parameters that we trained and optimized. We decided on an encoder-decoder structure with an inference model to accompany our first network (Figure 3), which instead of having learned parameters, processes data using our trained model to infer a result/sign. Since our problem is in the realm of image captioning, we thought that a encoder-decoder structure would fit best. We used a sparse categorical cross entropy loss for this model since each sample in our datasets belongs to exactly one class, which is the optimal situation in which to use this loss function. We chose sparse categorical cross entropy loss over normal categorical cross entropy loss since our samples do not have soft probabilities, meaning there is no need to sum over all the classes so we can save time and memory with our loss choice. Additionally, we chose to use an Adam optimizer since an Adam optimizer tends to converge faster than other optimizers (like SGD) while requiring fewer parameters to tune and a faster computation time, though we also experimented with other optimizers for comparison's sake.

One strategy we used to speed up training was to reduce the learning rate of our model if the validation accuracy has plateaued for 3 epochs. The learning rate is kept the same as long as the performance of our model keeps improving to allow for larger gradients as our model is still far from optimal weights. However, once the performance stagnates, we cut our learning rate by a factor of 0.5 to allow for a smoother and faster convergence of our model to a local minima because otherwise the model will overshoot and bounce around the minimum. In the same vein, we added an early stopping mechanism in order to streamline our training time. If the validation accuracy has not improved in 10 epochs, we stop training our model, and restore the best weights as our final result. The tensorflow keras "callback" utilities make these functionalities simple to implement and save us a lot of time training.

We also attempted to implement a 2D CNN model in order to replicate parts of the 2nd place competition winner in the dataset's source Kaggle competition. However, due to difficulties organizing the data and reducing GPU memory usage, we were unable to test this model in time for the project. The planned model structure is shown below. (Figure 4)

We expected issues in two main areas: cleaning the dataset and tuning our models for optimal solutions.

At first, we encountered issues regarding processing the data as well as training times, due to the massive size of the dataset as well as the more complicated multi-file structure. We eventually circumvented these issues by combing through the data and getting rid of irrelevant data as well utilizing the aforementioned pre-processing notebook.

Once we had our datasets loaded and our models configured, we did not expect that our solutions would immediately be successful, and had to do significant tuning in order to reach a certain amount of improvement that we found successful (see section 3).

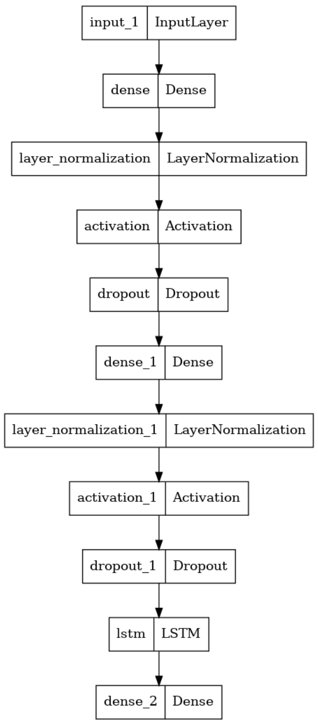

Figure 2. Baseline LSTM model with two dense blocks.

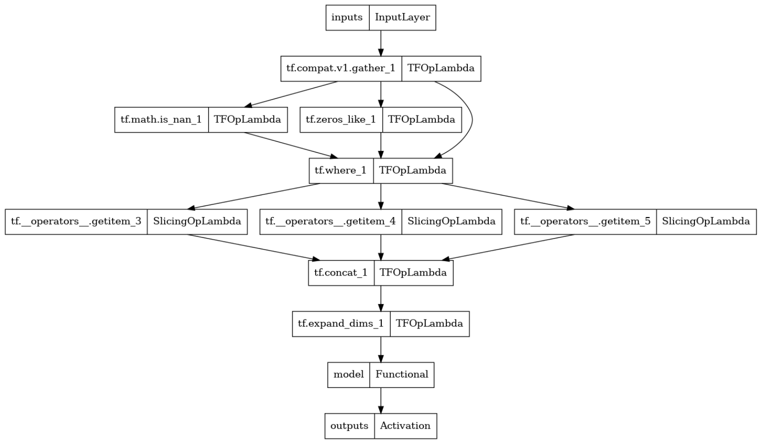
inputs | InputLayer
tf.compat.v1.gather_1 | TFOpLambda
tf.math.is_nan_1 | TFOpLambda
tf.zeros_like_1 | TFOpLambda
tf.where_1 | TFOpLambda
tf.__operators__.getitem_3 | SlicingOpLambda
tf.__operators__.getitem_4 | SlicingOpLambda
tf.__operators__.getitem_5 | SlicingOpLambda
tf.concat_1 | TFOpLambda
tf.expand_dims_1 | TFOpLambda
model | Functional
outputs | Activation

Figure 3. Inference Model.

3. Experiments and Results

3.1. Landmark Selection

Using the simplest version of the LSTM model, we compare the performance of 3 landmark combinations (Table 1). `HandOnly` performs worst among the three with accuracy trails behind at around 0.60. On the contrary, `UpperBodyPose` performs much better at 0.6865. We attributed this to the fact that `UpperBodyPose` provides more information about the orientation of the arms as well as mouthing cues. Albanie et al. [1] suggested that this extra context could help with recognizing co-articulated signs, which require two or more sign segments either simultaneously or sequentially. Interestingly, adding more facial data with eyes and face outlines did not improve performance tremendously, as `PoseNFace` had roughly the same accuracy as `UpperBodyPose`. This indicates that these extra details add little significance to sign recognition. Given these results, our model will be using `UpperBodyPose`

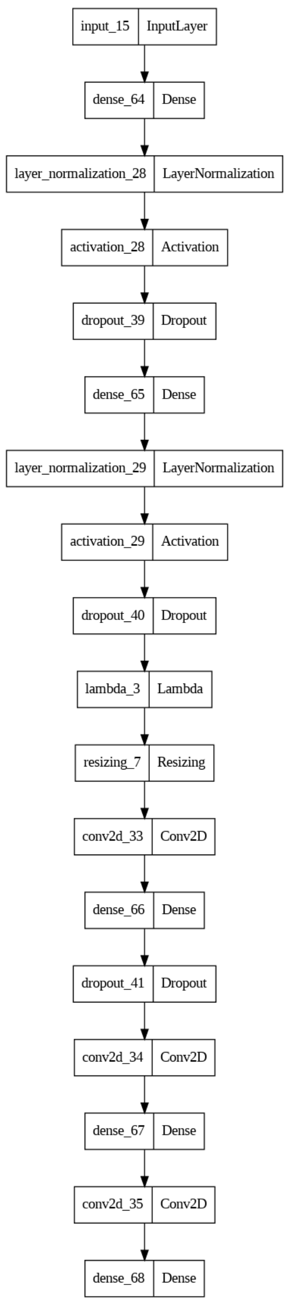

Figure 4. CNN model.

Landmark Combination	Val. Accuracy	Loss
`HandOnly`	0.6050	1.4609
`UpperBodyPose`	0.6865	1.3219
`PoseNFace`	0.6719	1.3403

Table 1. Validation accuracy of different landmark combinations on a simple LSTM model

as our landmark combination from this point onward.

3.2. Hyperparameter Tuning

We also decided to explore the effect different hyperparameters had on our LSTM encoder-decoder model. Our main means of comparison between our models and each different configuration was the accuracy and loss, which each measure the errors our trained model made on our validation set. For our LSTM network, we decided to test our loss and accuracy over 100 epochs on different values for the Encoder Units, LSTM Units, and Optimizer, keeping the decreasing learning rate a constant between all our configurations. Our idea of success was being able to achieve noticeable improvements from our experiments, and to discover patterns in the hyperparameters we chose to tweak.

After running for 100 epochs, our baseline LSTM model reached a validation accuracy of around 0.68 on our cleaned dataset. We first tried changing the number of LSTM units from 175 to 150 but it yielded a slightly lower validation accuracy. Similarly, when we raised the number of LSTM units from 175 to 200 it also yielded a slightly lower validation accuracy. We can attribute these slightly lower validation accuracies to the fact that we initially chose a good value for the number of LSTM units. When we tested a lower value, we ended up not making our model complex enough and slightly underfit on the problem at hand. On the other hand, when we tested a larger value, we ended up overfitting on the training data and so the model wasn't able to generalize as well to the validation set. This is proven through data since the model with 150 LSTM units had a training accuracy of 0.9130 while the model with a 200 LSTM units had a training accuracy of 0.9278 which means the first model underfit more and the second model overfit more.

The second approach we tried was to change the optimizer we used. From our prior research, it was pretty clear that an Adam optimizer would be the best general choice, but we decided to test using a SGD function for our optimizer. Unfortunately, the results matched our expectation as when we switched to a SGD optimizer, the model was able to only reach a validation accuracy of 0.2451. This was to be expected since the Adam optimizer is able to converge quicker to a minimum while the SGD optimizer is better at converging at more optimal solutions but takes much longer. It is evident that 100 epochs of training wasn't sufficient for the SGD model to finish training and even through the last few epochs, the validation score was still going up. Given more time and a longer training period, perhaps a model with a SGD optimizer could have outperformed our baseline model.

The final hyperparameter we tuned was the number of encoder units. Similarly to our approach to LSTM units, we decided to run one test with a larger number of encoder units and one with a smaller number. These models returned validation accuracies of 0.6589 and 0.6672 respectively which

Learning rate	Optimizer	Encoder Units	LSTM Units	Val. Accuracy	Loss
Variable	Adam	[512, 256]	150	0.6968	1.2944
Variable	Adam	[512, 256]	175	0.6865	1.3375
Variable	Adam	[512, 256]	200	0.6922	1.3278
Variable	SGD	[512, 256]	175	0.2451	3.4234
Variable	Adam	[1024, 512]	175	0.6589	1.3647
Variable	Adam	[256, 128]	175	0.6672	1.3261
Variable	Adam	[512, 256, 128]	150	0.6685	1.3455

Table 2. Validation accuracy after running our LSTM network for 100 epochs

lead us to believe that our baseline model's inital configuration of encoder units was already optimal. Additionally, we tried adding an additional dimension to the encoder units to see if the model would be able to learn a more complex solution; however, the validation accuracy only reached 0.6685 which was still lower than our baseline model so it seems adding another dimension didn't help to improve our model performance.

4. Conclusion

Our exploration of LSTM models with the ASL dataset created by Google led to some interesting findings, especially regarding dataset preprocessing and hyperparameter tuning. We tried a variety of approaches, from changing which MediaPipe Landmarks to train on or filter out, as well as increasing the amount of layers and nodes in the dense layers of our model. Our best approach seemed to be training on landmarks that include the upper body and hands but not the face, as well as our baseline hyperparameters that included 512 nodes and 256 nodes for our encoder units in the dense layers, and 150 nodes for our LSTM units. We found that an Adam optimizer combined with a steadily decreasing learning rate and a sparse categorical crossentropy loss function was optimal for the labelling task at hand. We encountered issues when it came to grappling with this new dataset as well as optimizing training times for our models, and there is still much exploration that could be done in future research regarding the performance of LSTMS on this sign langauge data set, including experimentation on different architectures such as CNNs.

5. Work Division

Refer to Table 3 for work division.

6. References

References

[1] Samuel Albanie, Gül Varol, Liliane Momeni, and Triantafyllos Afouras. Bsl-1k: Scaling up co-articulated sign language recognition using mouthing cues. 2020. 3

[2] Matyáš Boháček and Marek Hrúz. Sign pose-based transformer for word-level sign language recognition. pages 182–191, 2022. 1

[3] Joao Carreira and Andrew Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset. pages 6299–6308, 2017. 1

[4] Subhadeep Dey, Abhilash Pal, Cyrine Chaabani, and Oscar Koller. Clean text and full-body transformer: Microsoft's submission to the wmt22 shared task on sign language translation. 2022. 1

[5] Dongxu Li, Cristian Rodriguez, Xin Yu, and Hongdong Li. Word-level deep sign language recognition from video: A new large-scale dataset and methods comparison. pages 1459–1469, 2020. 1

[6] Ozge Mercanoglu Sincan, Julio C. S. Jacques Junior, Sergio Escalera, and Hacer Yalim Keles. Chalearn lap large scale signer independent isolated sign language recognition challenge: Design, results and future research. pages 3472–3481, June 2021. 1

Student Name	Contributed Aspects	Details
Steven Le	Implementation Background and Analysis	Tuned and trained variations of the LSTM network, recorded/analyzed results, and wrote the Abstract, introduction, and approach sections of the paper
Matthew So	Implementation and Analysis	Implemented the 2D CNN of the encoder and analyzed the LSTM results. Proposed project, wrote background and related work of proposal, and performed literature review.
Hailey Ho	Implementation and Analysis	Tuned and trained a simple LSTM network. Analyzed different landmark combinations for best performance. Wrote the preprocessing section and the first subsection of results.
Bryan Zhou	Implementation, Hyperparameter Tuning, and Analysis	Wrote the code implementing the LSTM model used throughout the paper and performed hyperparameter tuning on the model to attempt to improve the model's performance. Wrote portions of Sections 2 and 3 in the paper and provided support on the rest.

Table 3. Contributions of team members.