ASL Detection Correction and Completion

Andrew Xavier
Columbia University
New York, United States
ahx2001@columbia.edu

Alban Dietrich
Columbia University
New York, United States
ad4017@columbia.edu

Kenneth Munyuza
Columbia University
New York, United States
km3829@columbia.edu

Sam D. Brosh
Columbia University
New York, United States
sdb2174@columbia.edu

Abstract

We introduce an American Sign Language Detection, Correction, and Completion (ASLDCC) unit that is equipped with multiple novel features. The application has been designed to prioritize usability, providing an interface that allows both experienced and inexperienced signers to communicate in American Sign Language (ASL) and non- ASL speakers to understand. Our application can be used on computers or smartphones. We show promising results and highlight the potential for this application to be made available to the general public.

ACM Reference Format:
Andrew Xavier, Alban Dietrich, Kenneth Munyuza, and Sam D. Brosh. 2023. ASL Detection Correction and Completion. In *Proceedings of (ASL Sign Language Auto Complete '23).* ACM, New York, NY, USA, 7 pages. https://github.com/Kennethm-spec/ASL-Auto-Complete

1 Introduction

ASL is a communication method that is specially meant for people to engage with the deaf community. ASL is a form of communication that utilizes hand gestures instead of spoken words. Due to the simplicity of the signs, ASL detection has become a very popular task in the arena of image processing and classification. These tasks are very important to the deaf community because they allow them to communicate with people unfamiliar with ASL. State-of-the-art algorithms have been shown to produce impressive results, including a 99.73% [1] test rate (and other similar results depending on the dataset), indicating that ASL detection is effectively a solved task. Learning how to sign in ASL can be a tedious process, which is a common complaint among many signers.

This introduces our project, ASLDCC. The ASLDCC application has three main submodules: the detection submodule, the natural language processing (NLP) submodule, and the full-stack submodule. Each of these three play a key part in the project.

The detection submodule is responsible for inferencing what each sign that the user is making. The NLP submodule makes inferences on the incoming letters by suggesting three possible corrections or completions. Finally, the full-stack submodule displays all of the information including current letter detection and the adaptive recommendation system to the signer in a user-friendly way.

The ASLDCC is especially useful when signers only have access to a rudimentary interface and can not type on a keyboard what they want to communicate. These types of interfaces are becoming more ubiquitous as we move to a world where Augmented Reality (AR) is a prevalent part of life. By using this application, signers do not have to worry about making mistakes or taking more time to sign individual letters.

2 Related Work

MediaPipe hands is a feature extractor made by Google that lets developers detect the landmarks of the hands in an image. It is useful for Machine Learning (ML) tasks because it can greatly decrease the complexity of an image involving hands. It has been used for sign language purposes in several projects and papers. Researchers have used it across a number of languages such as [2] which uses MediaPipe for detecting Thai finger spelling gestures using a deep learning model for classification. Module **??** employs a comparable technique, but for American Sign Language and personalized hand gestures to interact with our system. Several groups use both hand and body landmarks to do full ASL word recognition such as [3] which builds upon the existing SPOTER architecture [4] and produces the WLASL100 and LSA64 datasets. Both of which use full skeletal tracking data for ASL sign language. However, these approaches use external APIs for classification, full-body detections, and/or large transformer-based models. In comparison, we aim to build a lightweight, mobile-friendly and computer-friendly system that runs all steps locally. We took inspiration from guides such as [5] which demonstrate creating custom datasets using MediaPipe that are suitable for sign language detection.

ASL Sign Language Auto Complete '23, May 2023, NYC, NY, USA
© 2023 Columbia University.
https://github.com/Kennethm-spec/ASL-Auto-Complete

3 Methods: System Design

We will be building a CV implementation for real-time ASLDCC for mobile and computer devices, shown in Figure 1. The goal will be to implement an efficient method of signing words and sentences using ASL. We will use MediaPipe Hands to generate hand landmarks from a video feed which we will then use to train an ML model that attempts to identify ASL representations of alphabet letters. Here, we will use the probability distribution outputs of this model for the current and previous predictions to help correct incorrectly classified hand symbols as well as provide suggestions for auto-completing the word using beam search. After each hand symbol, the model will suggest three auto-complete options for the word that the user can select and avoid tediously signing out the remaining letters.

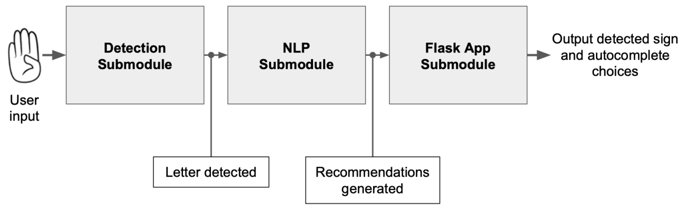


Figure 1. Application high-level schema.

In order to implement all of the features we have proposed, we will include a set of hand signs designated as selection symbols along with the individual letters of the alphabet in our dataset. For example, the user shall sign the number '1' to symbolize that they want to take the first suggested word completion. Finally, we will to scale and adapt the final model for mobile and computer use and deliver a working app. The UI will include both what the model has predicted so far as well as the words it thinks the user is in the process of spelling. We experiment with different custom hand signs for interaction, various numbers of beams in our beam search, and different environments that our model should function in.

3.1 Detection Submodule

An important design consideration for our team was to limit the number of computations as much as possible since we knew that we wanted our application to be compatible with embedded platforms. We experimented with using image-based models such as YOLO but found that these performed terribly as they struggled to generalize to a relatively small set of data. This required much more computational power that can lag devices since inference is done every frame. The biggest decrease in computations came with our implementation of the detection submodule. We chose to use MediaPipe hands and hand localization in order to limit the computational complexity of our application. Both of these options can arguably decrease accuracy, but we deemed the performance increases we got from each step to be more important than any loss in accuracy. A high-level depiction of our detection submodule is shown in Figure 2.

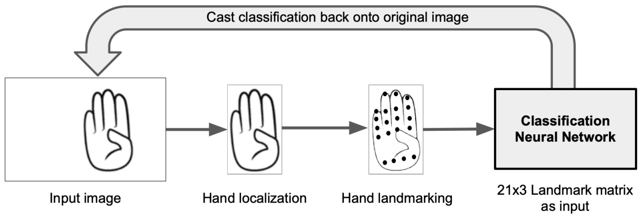

Cast classification back onto original image
Input image
Hand localization
Hand landmarking
Classification Neural Network
21x3 Landmark matrix as input

Figure 2. Detection submodule pipeline.

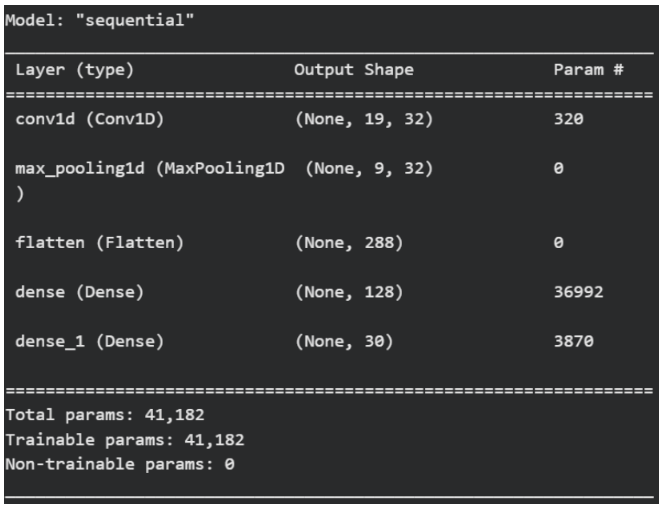
We first use MediaPipe [2] to detect the presence or absence of hand landmarks in an image. When there is a presence, the location of the points is normalized based on the maximum and minimum coordinate values for the x,y, and z values, which form a bounding box, so that the representation is robust to different frame resolutions and dimensions. The values are centralized around the middle of the bounding box so that the feature representations are independent of the hand's location in the frame as well. We then use this data to train a simple convolutional neural network (CNN). We use 900 data points per class for our 30 classes (including the whole alphabet, excluding the letter 'J' and 'Z' as they correspond to a hand motion and we only considered static signs, and some custom gestures for other control options) for training and validation. We confirm our results using a test set that contains 100 data points per class.

The final output of this module is single-frame classification predictions. We can use this to classify over sets of consecutive frames and use a combination of majority vote and confidence scoring to allow detect user inputs in a video live feed. This process is described in detail later in the Flask App Sub module 3.3.

3.1.1 ASL Image Data. We tried several different ASL image datasets to extract MediaPipe features on. We found that datasets with different angles and types of hands helped our model generalize. The final dataset for our model is a combination of different Kaggle datasets. We use a synthetic ASL data-set [6] from Lexset for the letters of the alphabet because of its variety in hand types and hand orientations, Lexset's synthetic numbers dataset [7] to serve as selection inputs for the autocompletion part, and Kaggle contributor, Akash's "delete" and "space" classes [8] to allow the user to delete letters previously typed and add spaces between words. In total, our training set had 900 images per class, or 28,800 data points and the test set contained 100 images per class, 3,200 data points. Since our dataset is static images, we omit "J" and "Z" as they could only be classified using sequences of images which would require significantly more data or a better static representation. This is a direction for future work.

3.1.2 Detection Model Architecture.

A summary of our model architecture is shown in Figure 3.



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d (Conv1D)             (None, 19, 32)            320

 max_pooling1d (MaxPooling1D  (None, 9, 32)            0
 )

 flatten (Flatten)           (None, 288)               0

 dense (Dense)               (None, 128)               36992

 dense_1 (Dense)             (None, 30)                3870

=================================================================
Total params: 41,182
Trainable params: 41,182
Non-trainable params: 0
_________________________________________________________________
```

Figure 3. Architecture of the classification model.

As previously mentioned, we attempted to implement SOTA detection models for this task, as we figured that they would detect signs the best. We deemed single shot detectors (SSD) as a viable choice, given that we wanted to perform inference in real time, and SSDs tend to be the best option for this type of application. However, after testing, we found that YOLOv5 and similar models failed to consistently and quickly make the correct classifications. It turned out that a custom CNN architecture (combined with a feature extractor for the mediapipe library) that is shown in Figure 3 was the best option for our project. Only a very rudimentary model is needed as mediapipe is an extremely effecient, effictive, and low-cost feature extractor.

3.2 NLP Submodule

A high-level depiction of our NLP submodule is shown in Figure 4.

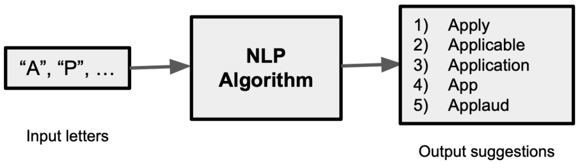


Figure 4. NLP submodule pipeline.

The NLP submodule is responsible for making inferences regarding the current output of the detection module. Depending on the state of the input buffer, the NLP algorithm will change its suggestions. For example, if the letter "A" was in the input buffer, the algorithm may output "Able" as its first suggestion. However, if the letter "P" was followed by the letter "A", then the NLP algorithm will change its first suggestion to "Apply" because it is more likely that the user wants to sign the word "Apply" than the word "Able".

Due to the fact that we wanted our application to be compatible with low-storage mobile and embedded devices, we had to deviate from the traditional language models that are more common for tasks like these. The NLP submodule is built using a Directed Acyclic Word Graph (DAWG) through the fast-autocomplete Python library [9]. This library only facilitates the creation of the DAWG object and does not have predefined words or frequencies. As such, we use the Brown Corpus from the Natural Language Tool Kit (NLTK) [10]. We remove all instances of punctuation outside of those that would compose a word itself such as an apostrophe ('), or a dash (-) which can form words such as "it's" and "six-pack". We also only count words that are considered legal words in a scrabble dictionary to help filter out odd or impractical words. We represent the words as an array and then create a hashmap that maps each word to the frequency it occurs which we are then saving to a local .json file. This is so we can easily load this data during inference rather than running the whole pipeline. This data is then used to form the DAWG graph which is then traversed during inference time to suggest autocompletion suggestions based on the provided prefix and the frequency of the word in the original corpus. This information is then handed to the front-end modules 4.2 and 4.3 which provides options to the user. An example of DAWG using only the top 20 words (based on frequency) is shown in Figure 5.

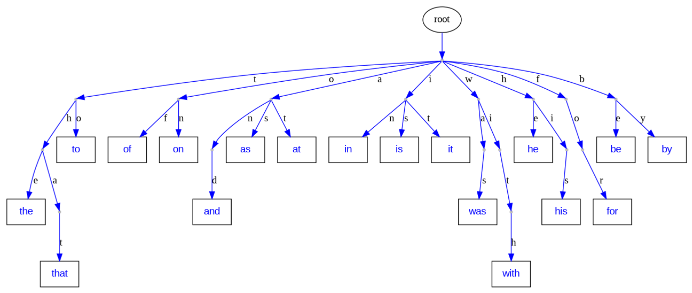


Figure 5. DAWG graph example using only our top 20 words. Actual DAWG during inference uses 23,000 words.

The decision to use a simple graphical model over a large language model (LLM) is to reduce the computation required given our limited-resource mobile devices which already will be running the segmentation and classification modules.

We tested using smaller-sized models, such as a pre-trained tiny-gpt2 [11] and GTP NEO [12], but these models either produced intelligible text or were too large to reasonably implement on mobile devices.

3.3 Flask App Submodule

The Flask app is a 4 steps process that uses the previously mentioned sub-modules in conjunction with its web-side code in order to encapsulate the entire functionality of our project. It begins with HTML5 code for the design of our viewport and CSS for styling and media screen adjustment optimized to flexible viewport sizing. The client's web browser runs JavaScript to access the user's camera and record the video using the HTML5 MediaDevices API. An HTML video element shows the video stream. The JavaScript code then utilizes the canvas API to grab frames from the video stream and delivers them to the Flask application via a Web Socket connection. This allows the Flask app to receive video frames in real-time and then buffer them so that they can be fed to our Model. This behavior allows for higher accuracy from our model by providing sample data for training and inference, further improving the performance of our computer vision-based ASL classifier. The results are then sent back through the web browser API and the viewport is updated through the JavaScript code. Additional duties that the Flask app needs to do include authentication, video encoding, and decoding, and error management.

Moreover, another program has been developed for computer users using the open-cv package. This allowed to have a better accuracy on the predictions, as we are not getting the video input from JavaScript but we get it locally via the open-cv package.

4 Results

In order to understand the efficacy of our application, we evaluate each submodule independently. We then conclude with a big-picture outlook as to the performance of the entirety of the application.

4.1 ASL Classification Model

We evaluate our classification and detection model on the Kaggle datasets Synthetic ASL Alphabet and Synthetic ASL Numbers [13] [14]. Our model performed extremely well, reaching **98.3% accuracy and an average f1-score of 0.985 across all classes**. A full confusion matrix of the results on a test set of 50 data points per class is shown below in Figure 6. We additionally display a table of scores for each class in Figure 7.

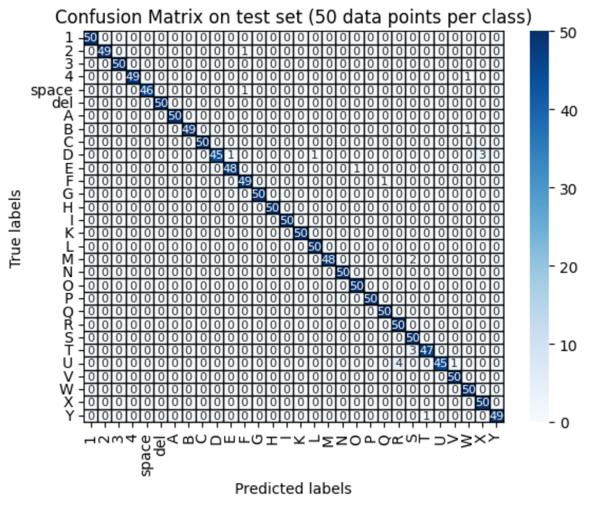
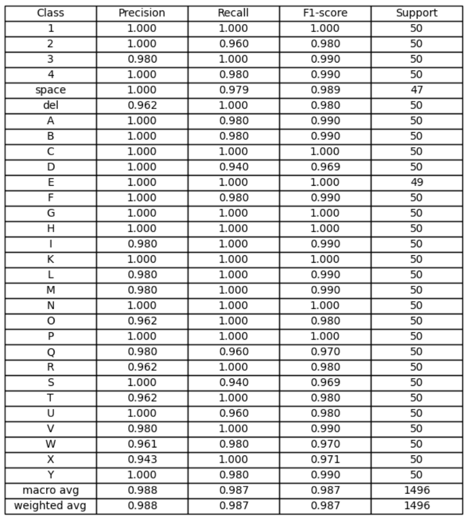


Figure 6. Confusion Matrix of ASL classification model on the test set (50 data-points per class)



Class	Precision	Recall	F1-score	Support
1	1.000	1.000	1.000	50
2	1.000	0.960	0.980	50
3	0.980	1.000	0.990	50
4	1.000	0.980	0.990	50
space	1.000	0.979	0.989	47
del	0.962	1.000	0.980	50
A	1.000	0.980	0.990	50
B	1.000	0.980	0.990	50
C	1.000	1.000	1.000	50
D	1.000	0.940	0.969	50
E	1.000	1.000	1.000	49
F	1.000	0.980	0.990	50
G	1.000	1.000	1.000	50
H	1.000	1.000	1.000	50
I	0.980	1.000	0.990	50
K	1.000	1.000	1.000	50
L	0.980	1.000	0.990	50
M	0.980	1.000	0.990	50
N	1.000	1.000	1.000	50
O	0.962	1.000	0.980	50
P	1.000	1.000	1.000	50
Q	0.980	0.960	0.970	50
R	0.962	1.000	0.980	50
S	1.000	0.940	0.969	50
T	0.962	1.000	0.980	50
U	1.000	0.960	0.980	50
V	0.980	1.000	0.990	50
W	0.961	0.980	0.970	50
X	0.943	1.000	0.971	50
Y	1.000	0.980	0.990	50
macro avg	0.988	0.987	0.987	1496
weighted avg	0.988	0.987	0.987	1496

Figure 7. Scores for each class on test set (50 data-points per class)

This is for single frames only. Since in our implementation, we classify over a set of frames and threshold at a high enough confidence rate, even the slight error in our frame-wise tests becomes negligible in the practical use.

We additionally show in Figure 8 the training accuracy and loss as well as the validation accuracy and loss. We use 10 epochs, but not that only 5 are really needed.

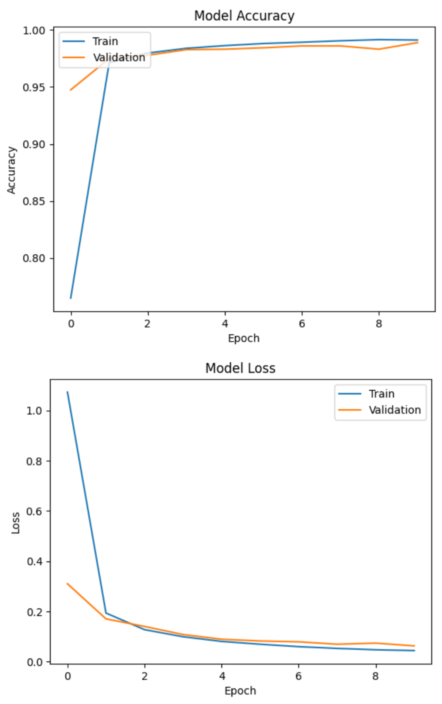


Figure 8. Accuracy and loss plots for training and validation

4.2 Full Stack Mobile App

4.2.1 Letter Recognition. A screenshot of the full mobile application ran on a local server is shown in Appendix A.1.

As seen in the screenshot, the user is signing the letter 'E', and the application is outputting a series of suggestions: 'even', 'each', and 'every'. The user can now sign either '1', '2', or '3', and automatically post the full word to the result field. It should be noted that the number of suggestions can be customized by the user. After the user finishes a word by either signing out the entire word or auto-completing it, they can continue their sentence by signing a new letter.

4.2.2 Mobile Demo: Using Autocomplete. Sill in in Appendix A.1, we show a sequence of screenshots where we demonstrate how autocomplete functions on the mobile version work. The user signs and then the letter "E" is detected and typed. The autocomplete displays 3 autocomplete suggestions below. Next, the user signs "2" to select the second autocomplete option, "each". This is then detected and typed onto the results.

4.3 Full Stack Web App

In addition to the mobile version, we have developed a fully functional web computer application (shown in Figure 9). It should be mentioned that this application does work slightly better than the mobile app as the laptop configuration removes the human error that comes when the user does not hold the phone steadily. An LSTM network could help with this problem of the user being slightly unsteady with their hand, as this would take prior information into account as well as current information. However, the usability is largely the same in both versions.

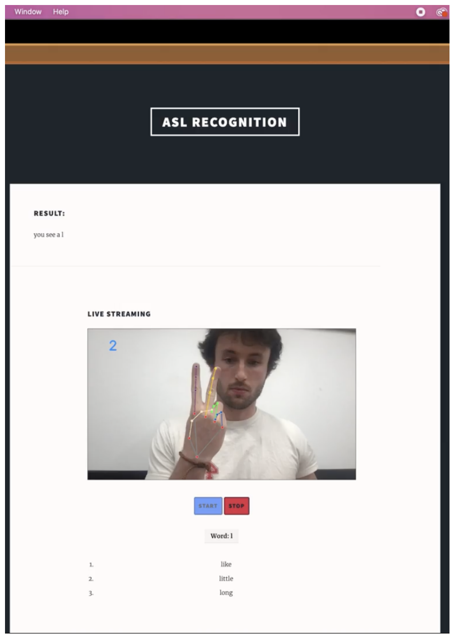


Figure 9. Web application running on a user's laptop computer.

5 Conclusion

An ASLDCC mobile application has been developed, tested, and deployed. ASLDCC has been shown to give ASL users a new tool to improve speed and lessen the difficulty in the process of communication. The detection module has been trained and tested on a diverse dataset, posting impressive results while being compatible with mobile and computer devices with limited computational power. Our NLP model has been adapted to design considerations and constraints. It is thorough, accurate and is robust to even the most uncommon of words. In summary, promising results have been shown in both the detection unit and the web application ease of use, cementing the ASLDCC as a major development in breaking the communication barrier that the deaf community faces.

A video demo of our project has been posted on YouTube. It can be accessed here: https://youtu.be/4rL-yoksNag. The video is also available on Google Drive: https://drive.google.com/file/d/1S6A-2W5vEcP8tb3hSTTVFpVtnhaFsv44/view?usp=sharing.

6 Future Work

In the future, we want to increase the dataset and have it more heavily pruned. Multiple of the three datasets we chose to employ in this project have mislabeled images; we believe that the occasional misclassifications that our program makes stem from these labeling errors. We believe that if we relabel the images, or even make a custom dataset where the angle of the hands are specific to our application use-case, our accuracy and usability will increase. Additionally, using data augmentation methods, such as flipping images across the vertical axis, to improve detection with both hands would be useful as some of the classes are imbalanced between right and left hand symbols.

There are additional areas where our project could be improved. In our testing, often we would try to make a sentence and find that an additional person was walking behind the signer, and mediapipe picked up on their hands. In the future, we want to implement a hand filter mechanism that eliminates all hands that are not the user's. This would greatly improve the application's versatility.

While we think that this application has true potential with the ASL crowd that sign letters, we acknowledge that our project does not address the more popular method of signing: words. Full words are signed by making moving gestures with both hands. Our model would fail to generalize to this application for two reasons. First, and the obvious reason, being that we did not train our algorithm on these signs. Secondly, and the more pressing issue, is that our model fails to account for motion. If we wanted to implement signed words, we would need to utilize, for example, an LSTM network (or some other type of residual network) to account for past frames, and accurately detect words and display sentence completion phrases. That being said, our model lays the groundwork for future work relating to ASL auto-complete.

References

[1] Ashok Sahoo, Gouri Mishra, and Kiran Ravulakollu. "Sign language recognition: State of the art". In: *ARPN Journal of Engineering and Applied Sciences* 9 (Feb. 2014), pp. 116–134.

[2] Camillo Lugaresi et al. "MediaPipe: A Framework for Building Perception Pipelines". In: *CoRR* abs/1906.08172 (2019). arXiv: 1906.08172. URL: http://arxiv.org/abs/1906.08172.

[3] Matyáš Boháček, Zhuo Cao, and Marek Hrúz. *Combining Efficient and Precise Sign Language Recognition: Good pose estimation library is all you need.* 2022. arXiv: 2210.00893 [cs.CV].

[4] Matyáš Boháček and Marek Hrúz. "Sign Pose-Based Transformer for Word-Level Sign Language Recognition". In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV) Workshops.* Jan. 2022, pp. 182–191.

[5] June 2021. URL: https://www.youtube.com/watch?v=doDUihpj6ro&amp;ab_channel=NicholasRenotte.

[6] Lexset. *Synthetic ASL Alphabet.* 2022. URL: https://www.kaggle.com/datasets/lexset/synthetic-asl-alphabet.

[7] Lexset. *Synthetic ASL numbers.* 2022. URL: https://www.kaggle.com/datasets/lexset/synthetic-asl-numbers.

[8] https://www.kaggle.com/grassknoted/aslalphabet and Akash Nagaraj. *ASL Alphabet.* 2018. DOI: 10.34740/KAGGLE/DSV/29550. URL: https://www.kaggle.com/dsv/29550.

[9] Sep Dehpour. *Fast-autocomplete.* Sept. 2015. URL: https://pypi.org/project/fast-autocomplete/.

[10] Edward Loper and Steven Bird. *NLTK: The Natural Language Toolkit.* 2002. DOI: 10.48550/ARXIV.CS/0205028. URL: https://arxiv.org/abs/cs/0205028.

[11] Sam Shleifer. *SSHLEIFER/Tiny-GPT2 · hugging face.* 2023. URL: https://huggingface.co/sshleifer/tiny-gpt2.

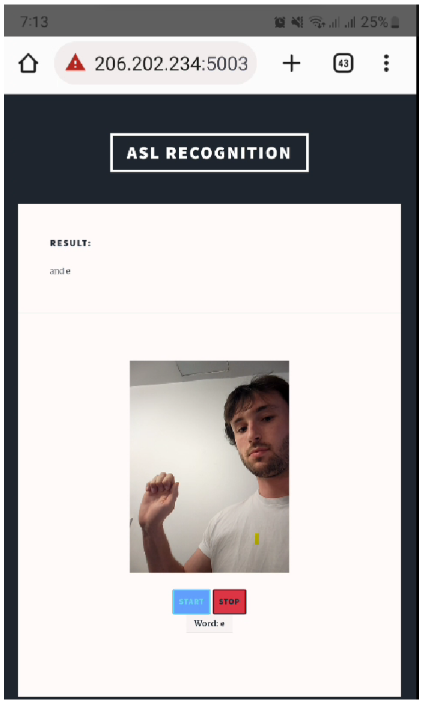
[12] Sid Black et al. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow.* Version 1.0. Mar. 2021. DOI: 10.5281/zenodo.5297715. URL: https://doi.org/10.5281/zenodo.5297715.

[13] Lexset. *Synthetic ASL alphabet.* June 2022. URL: https://www.kaggle.com/datasets/lexset/synthetic-asl-alphabet.

[14] Lexset. *Synthetic ASL numbers.* June 2022. URL: https://www.kaggle.com/datasets/lexset/synthetic-asl-numbers.

A Appendix

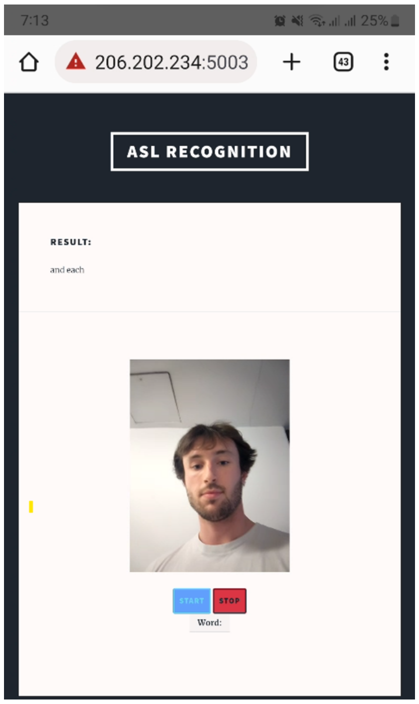
A.1 Frame-by-frame demo of mobile application



RESULT:

and e

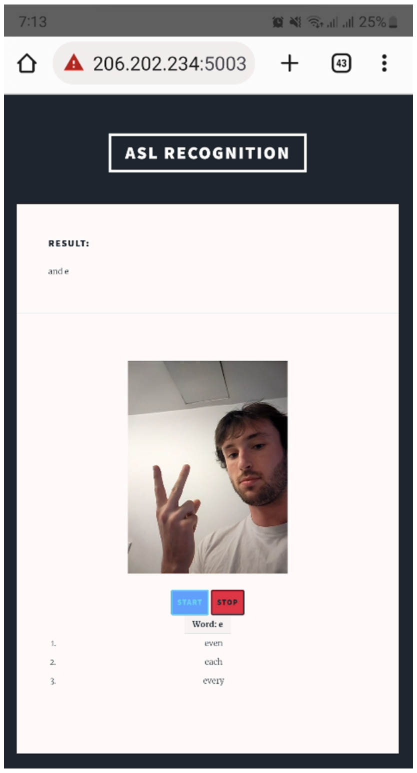
Word: e



RESULT:

and each

Word:



RESULT:

and e

Word: e

1. even
2. each
3. every