ASL Detection Using Machine Learning

Samantha Darby, Andrew Goldberg, and Justin Klein

May 11, 2024

Abstract

American Sign Language (ASL) is a primary means of communication between and with those who are deaf or hard of hearing, but currently only a small portion of the population knows how to sign. Most previous translation techniques contain a high cost of usage or a low accuracy rate. This paper seeks to improve upon the existing research in this field using the "ASL Alphabet" training dataset from the Unvoiced project to build a Convolutional Neural Network (CNN) model that can better translate for those who do not know ASL. While our final model has a higher accuracy rate than its competitors, the primary issue we encountered and the one most prominent in our review of the existing literature is overfitting due to a lack of diversity in the training dataset. We recommend that future researchers focus on improving the variety of this dataset before attempting to improve upon our model.

1 Introduction

Our group is interested in using a machine learning approach in order to identify photos of sign language users forming one-handed letters and translate them into written text. More than half a million people in the U.S. use ASL to communicate as their native language[1]. That makes ASL the third most commonly used language in the United States, after English and Spanish, and yet only 2.8% of Americans understand sign language[2]. Exacerbating the issue, most schools don't offer sign language as part of their curriculum (1000 out of 23519 public schools in the U.S.)[4][5]. This creates a language barrier that we hope to help mitigate through the use of our model as a translation device.

Up until relatively recently, translating sign language was a task that could only be done with the assistance of difficult-to-sift-through reference images or trained professionals. The earliest technology used to combat this language barrier were finger-spelling robotic hands created in the late twentieth century that took input from a keyboard and translated it into robotically formed signs[6]. This technology had obvious drawbacks, as it only served to translate type into signs, not signs into text or speech. Additionally, the costs of building and maintaining such a machine were prohibitive to say the least, and as a result, the technology never gained widespread acceptance. Later developments in sign language translation came in the form of gloves with motion sensors such as the CyberGlove in the early 2000s and, eventually, cameras[7]. Our project seeks to improve upon this most recent evolution in sign language translation technology.

Unlike the prohibitive and difficult to use machinery of glove or robot hand technology, 97% of Americans own a smartphone as of 2022, any of whom would be able to take a picture of someone signing in order to have it translated by our program[8]. This could be a picture of a sign language user directly, or a picture of signs in a graphic design or logo setting. Thus, our solution is intended to work not only for in person communication, but also long distance or non-kinetic, visual communication.

Our project seeks to use a convolutional neural network (CNN) trained using a batch of 87,000 photos in 29 classes to provide effective translation from sign language to text. The classes include the 26 letters of the English alphabet, the sign meaning space, the sign meaning delete, and a class where the photo contains no sign at all. As the focus of this paper is on the translation model itself and not the interface, this project represents only one step in creating an accessible translation program. However, it is our vision that after an accurate model is developed, an interface would be created that would allow a layman with minimal knowledge of computer programming to upload an image of a sign and receive a text translation in real time. This could be used as a communication aid or a learning tool for aspiring sign language learners.

2 Prior Work

As discussed in the previous section of this paper, computer scientists, linguists, and roboticists have historically approached the issue of sign language translation from a number of different angles. For the rest of this paper, we will set aside mechanical solutions and focus instead on machine learning solutions using image based training data.

There already exist many models with a similar translation purpose to our own. For example, the GitHub programmer Dimitri Govor created a publicly available program that uses a Long Short-Term Memory (LSTM) model to translate signs[9]. However, while this project has similar aims to our own and achieves somewhat similar ends, it requires that the user configure the translator to their personal signs by recording and entering their own training data. While this would likely be more accurate for the individual, it creates a loss of universality and a barrier of use.

Another project with similar intent to our own is Akash Nagaraj's "Unvoiced" project (also on GitHub), which, unlike Govor's, is trained on a database of generalizable images[10]. "Unvoiced" accepts an image as input, translates them using a CNN model, and then uses Google's Text to Speech API in order to read the letters aloud. With the exception of the text-to-speech function, our project is otherwise identical in set up and model choice, but we additionally use K-fold cross validation and multiprocessing to increase the accuracy and efficiency of our model.

Ultimately, although this problem has many suggested solutions already, it is the goal of this paper to demonstrate a meaningful improvement on the accuracy of existing models in order to further this area of research.

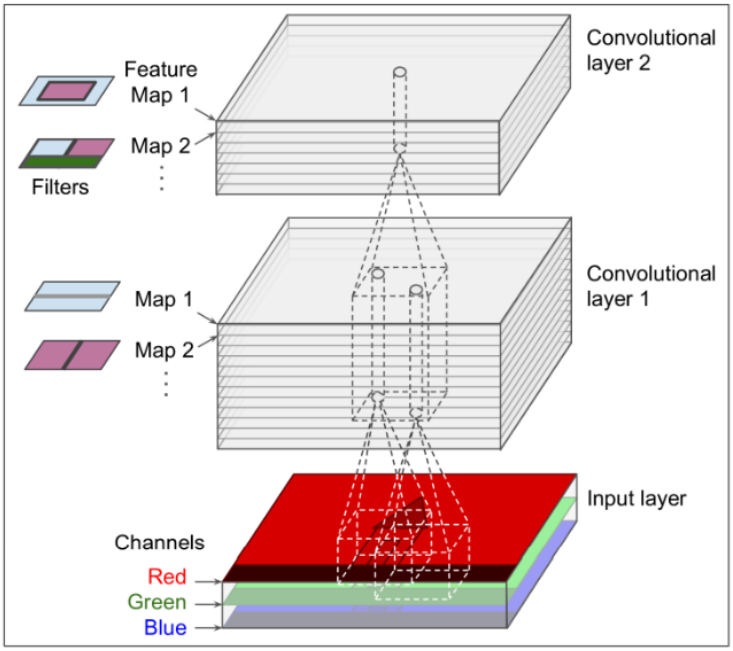

Figure 1: This diagram demonstrates how neural networks can be used to process images.[11]

3 Background

In order to understand the CNN model we intend to use in our project, it's important to understand how a CNN works. At their most basic level, CNNs are a type of deep neural network that use layers of filters to extract features from input images. These filters (see *Figure 1*) are capable of detecting patterns in the input images such as edges, textures, and shapes at different scales. The model learns to combine these features through pooling and fully connected layers to make predictions about the input, such as image classification or object detection. In our case, the dataset we are using comes labeled with the sign represented in each image, so our CNN will use supervised learning in order to achieve image classification.

Another important aspect of CNN functionality we use in our model is pooling. Pooling is a type of layer that is added after a convolutional layer in order to lower the spatial dimensions such as the width and the height while preserving the depth, such as the number of channels. It works by dividing the input features in a set of non-overlapping regions, known as the pooling regions. The regions are then transformed to a single value that represents the presence of a feature in the region. The type of pooling we use is max pooling, where the output value is the highest value of inputs within the region. Some advantages of pooling include reducing the dimensionality, achieving translation invariance, and helping with feature selection. Some disadvantages of pooling are information loss, over-smoothing, and that they introduce another variable into the hyperparameter tuning phase of model building[3].

Our model contains eight layers, starting with a 2D convolution layer with 32 filters and a 3x3 kernel. The 2D convolution layer cycles through each pixel in the image and applies the kernel, which helps detect details such as corners and edges, and then applies the Rectified Linear Unit (ReLU) activation function to introduce non-linearity to better account for real-world data. The second layer is a Max Pooling 2D layer, which has a pool window size of two-by-two. The Max Pooling 2D layer shrinks the size of the input by taking the maximum value from each region in the pool window and transforms the input layer into an output layer with, in the case of our model, the width and height being cut in half. The third layer is another 2D convolution layer, however this time with 64 filters. The fourth layer is another Max Pooling 2D layer with a pool size of two-by-two, and the fifth layer is the same as the third layer. The sixth layer is the Flatten layer, which converts the input from a two-dimensional array into a one-dimensional array to be processed by the fully connected layers. The seventh layer is the first fully connected layer, and we use the Dense layer with 64 neurons and the ReLU activation function. The Dense layer connects each neuron in this layer to each neuron in the previous layer, and the output is computed by a weighted sum of each neuron followed by ReLU. The eighth and final layer is another Dense layer, however the number of neurons used is the number of classes in the datasets, which, in our case, is 29.

4 Data

The dataset we will be using to train our model comes from the open source collaborative data science platform Kaggle[12]. Supplied by user grassknoted, our dataset (called simply "ASL Alphabet: Image data set for alphabet in the American Sign Language") is 1.11 GB large and contains 87,000 photos categorized into 29 classes consisting of the English alphabet from A to Z, space, delete, and a category for empty photos that do not contain any sign[13]. This initial dataset also comes with a testing set, however it is perfunctory, only containing one image for each class. Luckily, another Kaggle user, Dan Rasband, uploaded a test set corresponding to "ASL Alphabet" called "ASL Alphabet Test" which contains 870 images, 30 in each class.

One serious disadvantage of this particular training and test set combination is the low diversity in training images (there are only a handful of different backgrounds across 87,000 images, and they are all fairly monotone) limits the model's performance when run on the test set which has images with much more complex backgrounds. That being said, the training set does provide some variability in lighting and color scheme, which does benefit our model in low visibility conditions.

For some examples of images from our training set, see *Figure 2*.

5 Methods

Once we downloaded our training data, we quickly found that trying to run any code with it took up all the free RAM that was provided within Google Colab, so we took our efforts to Microsoft Visual Studio Code using Live Share. There we were able to work together to program the model, and gather results

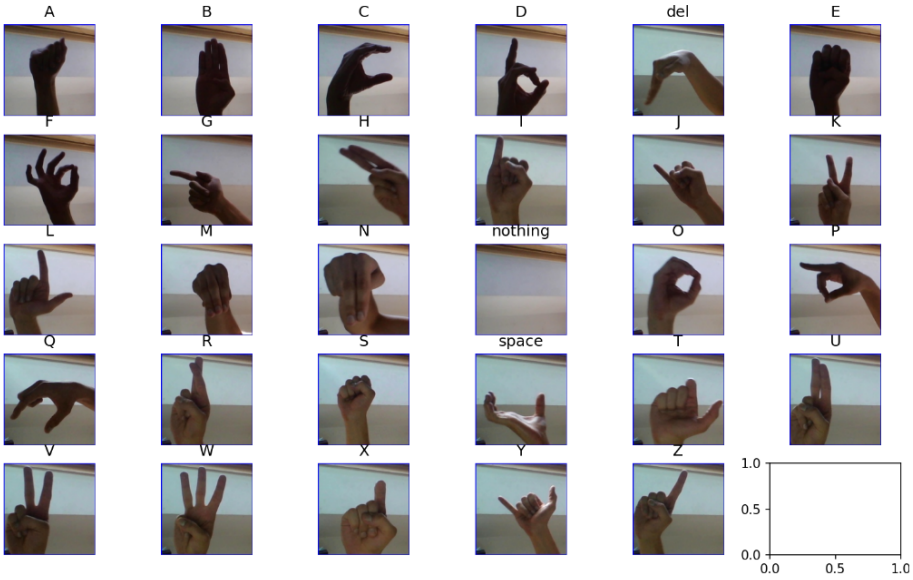

Figure 2: Example images for each class represented in our dataset.

in real time. The crux of our machine learning approach to this problem is a Convolutional Neural Network. In order to implement this model, we elected to use the open source Keras library which is known for constructing neural network algorithms, but before we could get to that, we first had to load our data[14].

Once we loaded our data from Kaggle into a Pandas DataFrame accessible by our code, we had to use the Pillow library to translate the images of the dataset into something viewable in Python. We then sorted the images by label and plotted one image for each label in the training dataset to make sure the data was being loaded into our Python program correctly (see *Figure 2*).

Once we had our data properly loaded and accessible, we separated 5% of the training set to form a validation set to be used during hyperparameter tuning. We then used the tutorial provided by TensorFlow in order to figure out the basic parameters necessary to give us a starting output for our model. Using these parameters, we were able to run our model on the training data and get some initial predictions and results, which are more fully explained in Section 6. After our initial results, we decided to retrain and test the model a few different times, varying the number of epochs to plot how this hyperparameter affects the accuracy of the model on the training and validation sets. The results from these experiments are reported in Section 6, *Table 1*. We attempted to speed up our hyperparameter tuning process by using a brute force technique that varied the number of epochs, batch size, and the number of folds used in our k-cross fold validation.

As we trained our model, one way that we have worked on decreasing the computing resources needed for our project is by introducing multiprocessing into our program. This significantly decreased the amount of time that was required to run the program, and thus we were able to get our results significantly faster.

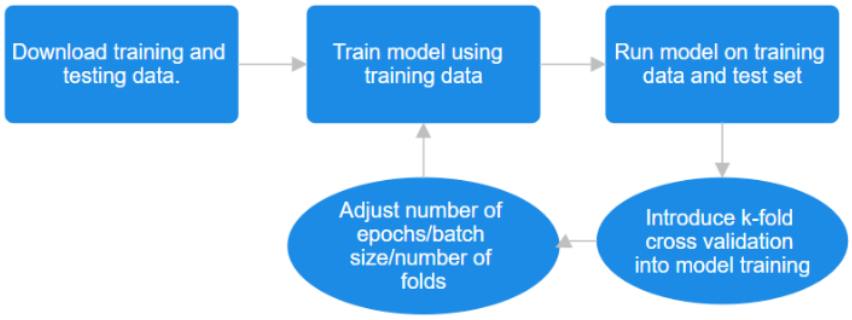

Figure 3: Flow chart representing how we trained our model.

Our unique contribution toward the methodology of this project is using the library Keras with this dataset (Nagaraj's model, our primary point of comparison, uses TensorFlow). Additionally, we used multiprocessing and k-cross fold validation, which neither of the projects discussed in Section 2 of our paper did.

6 Results and Analysis

Our initial CNN model, which we trained using the default settings (batch size of 32; one epoch), was incredibly ineffective. It had an accuracy of 47.42% on the training set, but only 1.73% on the test set.

We did some early testing varying only the number of epochs which showed that the more epochs we used, the better the model performed. Our results (see *Table 1*) make this conclusion immediately obvious.

Epoch	Training Acc.	Training Loss	Val. Acc.	Val. Loss
1	0.4742	1.7147	0.0173	33.8011
2	0.9248	0.2180	0.0331	32.1807
3	0.9654	0.0978	0.0272	41.7845
4	0.9795	0.0736	0.0498	47.3156
5	0.9847	0.0503	0.0366	45.8700

Table 1: Testing results from varying the number of epochs used during training.

However, although the improvement generated by increasing the number of epochs was significant, the overall testing accuracy of the model was still incredibly low. In order to combat this, we implemented two new improvement strategies. First, we started using k-fold cross validation when training our model, which yielded another immediate performance improvement, as evidenced by *Figure 4*.

Second, because we still felt that just increasing the number of epochs and k-folds was an inelegant and ultimately sub-optimal solution, we decided to fine-tune our hyperparameters by implementing a grid search on the number

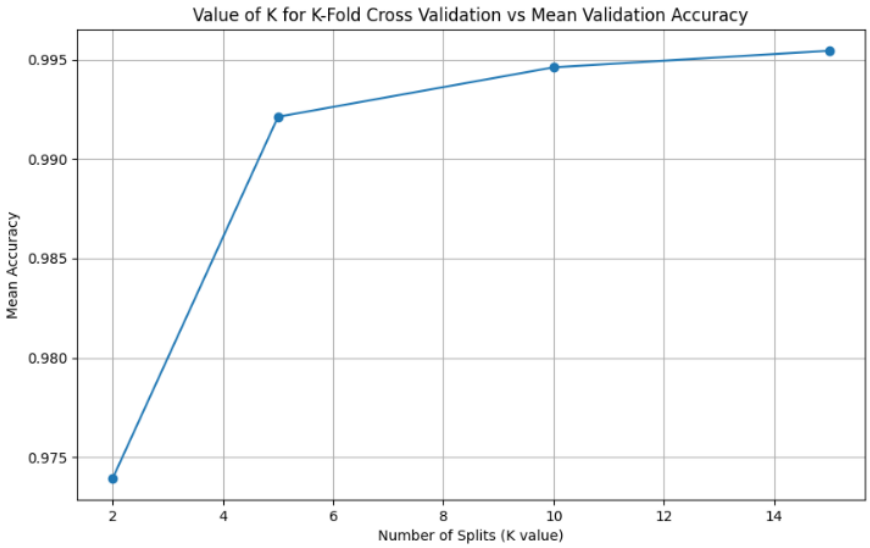

Figure 4: Graph exhibiting the rise in accuracy in correlation with an increase in k-folds.

of epochs, the number of validation sets in our k-fold cross validation, and the batch size. Unfortunately, as soon as the grid search program launched, both the CPU and RAM usage spiked to 100%, and the computer ran extremely slowly. Because of this, we decided to switch from using the library function provided by scikit-learn to a method of our own that ran much more efficiently. We used the following arrays as inputs: [32, 64, 128, 256, 512, 1024] for the batch size, and [2, 5, 10, 15] for both the list of k values for cross validation and the number of epochs. This was incredibly time-consuming as the full brute force search — which included 96 different parameter combinations — took over 34 hours to run. However, it was worth it as we were able to attain several options for training and hyperparameters that created vastly improved results. Some of these were immediately evident to be overfits, as the training and validation accuracy for those models was 100%. Our best model (in terms of time to run - almost all the combinations had the same accuracy and loss values) had the following parameters and results:

- Best Parameters: batch size = 1024, n splits = 2, and epochs = 2
- Best Training Time: 19.37
- Best Training Accuracy: 1.0
- Best Training Loss: $2.74*10^{-12}$
- Best Validation Accuracy: 1.0
- Best Validation Loss: 0.0

These results are exceptional for a machine learning model, but also incredibly suspicious. However, after changing the values in each of the three arrays

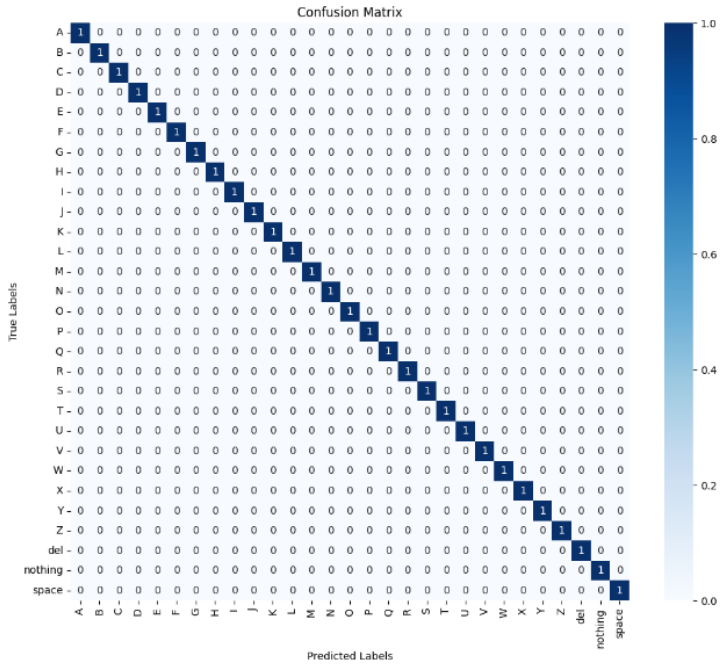

Figure 5: Confusion matrix modelling the results of our model on the "ASL Alphabet" test set.

in order to brute force search with, it was found multiple times that the best k-value and number of epochs was always two, but only when considering time to run (something we did because all of our models were coming out with nearly the same perfect accuracy). Based on the confusion matrix for these results (see *Figure 5*), we also worried about overfitting.

It was about at this point that we realized we were using the test set attached to our training data, which, as described in Section 4, only has one image in it for each letter, making it mostly useless as a metric for our model's performance. To rectify this issue, we returned to Kaggle and downloaded the "ASL Alphabet Test" set, which contains a respectable 870 images.

Now that we had an improved test set, we ran our brute force algorithm again with some improvements to optimize for accuracy rather than time. We also adjusted the levels to our epoch and k value arrays making our final search parameters [32, 128, 256, 512, 1024] for the batch size, and [10, 20, 30, 40, 50] for both the list of k values for cross validation and the number of epochs. We chose to make this change in order to test whether higher k-values and epochs would significantly improve the results. With these improvements, we found a new best model:

- Best Parameters: batch size = 128, n splits = 20, and epochs = 50
- Accuracy: 0.3713
- F1 Score: 0.3552
- Precision: 0.4013

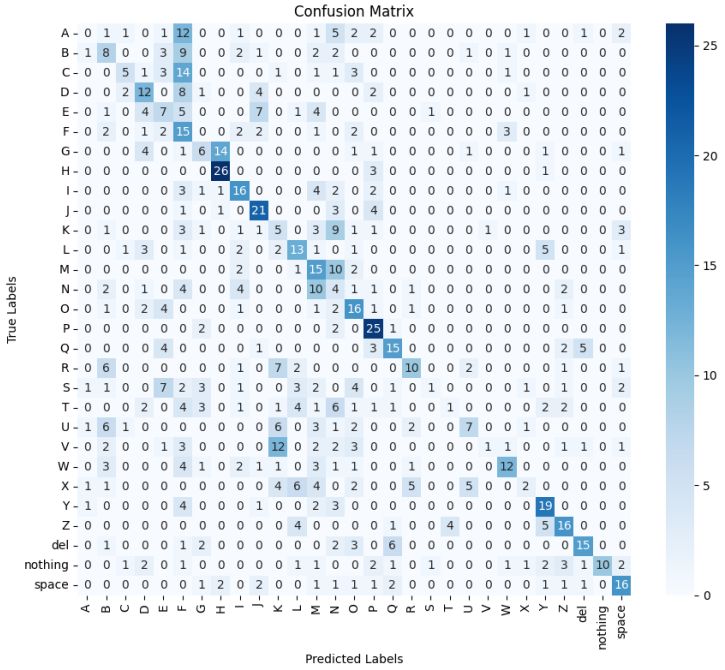

Figure 6: The confusion matrix for our final model.

- Loss: 81.17

Note: best training time has been removed as a factor, and we have added both F1 and precision to our results. The confusion matrix associated with this model (*Figure 6*) is significantly messier than the one generated by the test set that only had 29 images, but it reveals to us a few interesting insights into our model. For example, the letters P, H, Z, and del have the highest frequency of being predicted accurately, while our model has a tendency to erroneously predict that an image is C, F, M, or N when it is not.

Clearly, from its accuracy and confusion matrix, this latest model is performing much worse than our previous models. This difference is even more stark than we would naturally expect just from transitioning to a larger test set.

In order to investigate why this might be, we looked back at our data again and found the problem almost immediately: as described in Section 4, the "ASL Alphabet" set consists of images with very simple backgrounds, generally consisting of only geometric shapes of color. The positioning of the signs and the lighting are variable, but none of the training images achieve the level of background noise present in the "ASL Alphabet Test" set, whose images mostly have things like complex construction sights and beamed ceilings in the background that create a less clearly defined profile for the signs. See *Figure 7* for a comparison between a typical image in our training set vs. a typical image from our testing set.

Figure 7: On the left is an image from our training set. On the right is an image from the test set. The difference in the two backgrounds is apparent.

7 Comparison to Other Projects

As described in Section 2, we looked at two other attempts to create a machine learning model that can recognize ASL before starting our project. In this section, we are going to be comparing our results to those obtained by just one of those other projects. The first project we looked at was that of Dimitri Govor. We found that his model is a Long Short Term Memory (LSTM) model which is an RNN model over a CNN model, which means his approach differed greatly from ours. He did not include the training and testing data used to evaluate his model, so we ran his model on our training and testing datasets. We found that the accuracy of the model was only 4.03%, which is substantially lower than our model. This can be attributed to his model only having five layers, compared to our model's eight, and the fact that he trained it on 100 epochs, but did not use K-Fold Cross Validation. A table of Govor's model's performance on our testing datasets is shown in *Table 2*.

Letter	Accuracy	Letter	Accuracy	Letter	Accuracy
A	0.00%	B	3.33%	C	0.00%
D	0.00%	E	0.00%	F	6.67%
G	0.00%	H	3.33%	I	0.00%
J	0.00%	K	0.00%	L	0.00%
M	0.00%	N	0.00%	O	0.00%
P	0.00%	Q	0.00%	R	0.00%
S	0.00%	T	0.00%	U	0.00%
V	0.00%	W	0.00%	X	0.00%
Y	0.00%	Z	3.33%	DEL	0.00%
NOTHING	86.67%	SPACE	0.00%	Average:	16.44%

Table 2: Results from Govor's model when run on the "ASL Alphabet Test" set.

Recall from Section 2 that the second model we reported on was constructed by Akash Nagaraj. This model was made using the same dataset that we based our model on, and Nagaraj reports that his final model achieved a validation accuracy of roughly 95%, with an accuracy of 80% on the original training

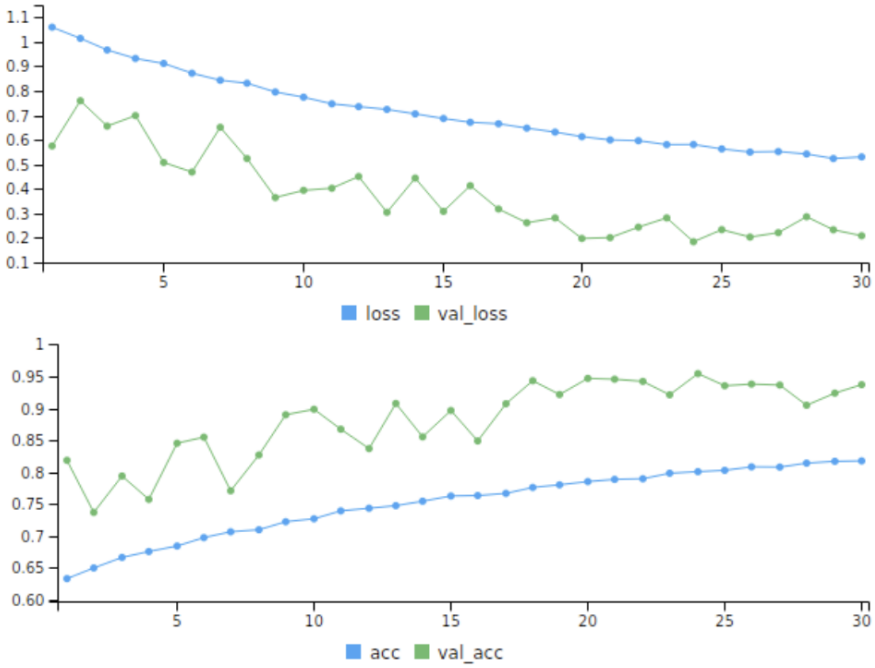

Figure 8: Accuracy and validation results from Nagaraj's final model on our data and testing set.

set[10]. We have included his accuracy and loss diagrams for the training and validation dataset in *Figure 8*.

However, when we downloaded Nagaraj's model and ran it with our training and testing dataset, the accuracy of his model plummeted to 16.44%. The full breakdown of his model's performance is shown in *Table 3*.

Letter	Accuracy	Letter	Accuracy	Letter	Accuracy
A	63.33%	B	0.00%	C	0.00%
D	33.33%	E	3.33%	F	33.33%
G	6.67%	H	50%	I	53.33%
J	3.33%	K	3.33%	L	3.33%
M	13.33%	N	3.33%	O	33.33%
P	3.33%	Q	3.33%	R	0.00%
S	3.33%	T	6.67%	U	3.33%
V	0.00%	W	16.67%	X	3.33%
Y	6.67%	Z	6.67%	DEL	56.67%
NOTHING	16.67%	SPACE	46.67%	Average:	16.44%

Table 3: Results from Nagaraj's model when run on the "ASL Alphabet Test" set.

Therefore, despite our initial fears, it appears that our model's performance

is in fact better than that of both of our competitors', but we can still learn from their methods. The primary difference between Nagaraj's model and our own is that he chose to construct his model using a TensorBoard graph, while we used the Sequential class from Keras. We also saw that he used the softmax function in his model, although most of his code remains hidden, preventing further study. One interesting discovery we made while investigating his project was that he originally planned to use a CNN using the Keras library like we did, but ultimately decided against it, writing, "...it did not give good accuracy...". This likely means that he hit the same dead end that we did in our project and was forced to pivot. However, the disconnect between his reported results and the results of his model when run on our test set cast many of his project decisions into serious doubt.

8 Conclusion

Overall, we believe we've made a significant contribution to the public issue of sign language translation. Our model had a higher accuracy rate than that of its competitors, likely due to the optimizations we made compared to just using a CNN alone. Although it still has a long way to go before it could be classified as a reliable translation device, it is still an improvement on existing ASL translation models. The primary issue that remains in our model is overfitting due to flawed training data, and thus our recommendation for further research would be to focus on diversifying that dataset rather than focusing only on the model itself.

References

[1] "American sign language," American Sign Language, Commission on the Deaf and Hard of Hearing (CDHH), https://cdhh.ri.gov/information-referral/american-sign-language.php. (Accessed Feb. 28, 2024).

[2] R. E. Mitchell and T. A. Young, "How many people use sign language? A national health survey-based estimate," Journal of deaf studies and deaf education, https://pubmed.ncbi.nlm.nih.gov/36423340/ (Accessed Feb. 28, 2024).

[3] "CNN: Introduction to pooling layer," GeeksforGeeks, https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/ (Accessed May 9, 2024).

[4] Paul, "How to learn ASL through video lessons online," ASL Deafined, https://www.asldeafined.com/2019/04/american-sign-language-in-high-school/ (Accessed Feb. 28, 2024).

[5] "The NCES Fast Facts Tool provides quick answers to many education questions (National Center for Education Statistics)," National Center for Education Statistics (NCES) Home Page, a part of the U.S. Department of Education, https://nces.ed.gov/fastfacts/display.asp?id=84 (Accessed Feb. 28, 2024).

[6] Jaffe, DL, "Evolution of mechanical fingerspelling hands for people who are deaf-blind," Journal of rehabilitation research and development, https://pubmed.ncbi.nlm.nih.gov/7965881/ (Accessed Mar. 21, 2024).

[7] B. S. Parton, "Sign language recognition and translation: A multidisciplined approach from the field of Artificial Intelligence," OUP Academic, https://academic.oup.com/jdsde/article/11/1/94/410770?login=true (Accessed Mar. 21, 2024).

[8] "Cell phone statistics 2024," ConsumerAffairs, https://www.consumeraffairs.com/cell_phones/cellphonestatistics.html\#:\~:text=About\%2097\%25\%20of\%20Americans\%20owned,364\%20million\%20people\%20by\%202040 (Accessed Mar. 21, 2024).

[9] D. Govor , "Sign-language-translator: Neural network that is able to translate any sign language into text.," GitHub, https://github.com/dgovor/Sign-Language-Translator (Accessed Feb. 28, 2024).

[10] A. Nagaraj, "UNVOICED: Sign-languge to speech," Akash Nagaraj, https://www.akashnagaraj.me/portfolio/13unvoiced/ (Accessed Feb. 28, 2024).

[11] A. Geron, *Hands-on machine learning with scikit-learn, keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*, 2nd ed. Sebastopol, CA: O'Reilly Media, 2019. (Accessed Feb 28, 2024)

[12] "Your machine learning and Data Science Community," Kaggle, https://www.kaggle.com/ (Accessed Mar. 21, 2024).

[13] Akash, "Asl alphabet," Kaggle, https://www.kaggle.com/datasets/grass_knoted/asl-alphabet (Accessed Feb. 28, 2024).

[14] Team, K. (n.d.). Simple. Flexible. Powerful. https://keras.io/ (Accessed Feb 28, 2024).