Using CNN to Distinguish Sign Language Letters

Group Two: Heather Anderson, John Naquin, Andy Mattick

Collaboration statement: Each of us developed our own CNN model and tried to tweak it to be the best for the group. After we determined John's was the best, we all worked to maximize it. Each of us contributed to the paper (Heather wrote the introduction and dataset sections, John described the model development and training and Andy wrote the results, conclusions and set up the visualization section. Each of us made suggestions about each other's contributions. John set up the front end and back end of the web page so we could put the model there and get people to submit their own images to be tested.

Google drive link (with datasets, code, presentation and paper): Heather, Andy, John Final Project Turn-In

John's Initial Colab link: Johns Code

Andy's Initial Colab Link: Andy Code

Heather's Initial Colab Link: mnistcnn.ipynb

Group Colab Link (version one): Group Code Version 1

Group Colab Link (version two): Group Code Version 2

Web Interface (API): Sign Language Recognition Website

Video Presentation: FinalProjectPresentation_Group2_SignLangCNN.mp4

Dataset Selection & Preprocessing

Selected Datasets (from kaggle.com)

1. Sign Language MNIST

   Sign-Language Classification CNN

2. Interpret Sign Language with Deep Learning Dataset

   Interpret Sign Language with Deep Learning

Introduction

Communication is essential for human interaction, yet for the deaf and/or hard-of-hearing communities, getting access to information can present unique challenges. One promising solution to this communication gap is the use of automated sign language recognition. In recent years, deep learning techniques, such as Convolutional Neural Networks (CNNs), have shown tremendous potential in classifying visual data. This paper focuses on developing an effective CNN model to recognize sign language letters, which is a significant step towards a more comprehensive sign language translation system that could potentially help individuals in these communities.

Sign language recognition can be challenging, and some of these challenges in recognition come from the variability of hand shapes, orientations, and lighting conditions in real-world images. While earlier models, which used simpler methods, often fell short when faced with any of these complexities, deep learning and CNNs have emerged as a powerful tool for extracting some of those intricate patterns that are found in images. This leads to development of models that can achieve high accuracy in classification tasks.

Our project aims to address two main objectives. First, we sought to develop a CNN-based framework that would be capable of accurately classifying sign language letters. Second, we wanted to compare the performance of various CNN configurations, including variations in the number of convolutional layers, training time (number of epochs), and input image resolutions. With this we aimed to determine the optimal architecture that would give us a balance of both accuracy and computational efficiency. With all of that being said, the importance of this project extends beyond academic purposes. An accurate sign language recognition system has the potential to be integrated into real-time applications such as web platforms or mobile apps, which would further improve accessibility for the deaf community.

Our team decided to approach this project in a fully collaborative manner. Initially, we each developed separate models using different datasets. Our next steps were evaluating and comparing each model's performance across multiple metrics, such as accuracy, recall, precision, and F1 score. Then, we addressed the shortcomings of each, such as overfitting and dataset imbalances. And ultimately, we combined the best practices from these initial models in order to create a final, fine-tuned CNN architecture that achieved an impressive 99% accuracy. We also extended our work by developing a web interface that allows users to upload an image of a sign language letter and get an instant result of which sign language letter is in the image (classification). This integration of machine learning with a user-friendly application highlights the practical impact of our research.

Overall, our paper adds to the research on sign language recognition by showing off our powerful CNN model and explaining how dataset characteristics and model configurations affect performance. We hope that our research findings will inspire more research and lead to real-world applications that improve communication access.

Literature Review

Early methods for sign language recognition relied on manually programming features, which is essentially telling the computer what to look for, like hand shapes or motion patterns. Approaches like Hidden Markov Models (HMMs) and Dynamic Time Warping (DTW) worked, but they weren't great at handling variation in lighting, background noise, or slight differences in how people sign (Kumari & Anand, 2024). However, with deep learning, CNNs became a game-changer because they could learn directly from images. This made them much better at recognizing hand signs without needing hand-coded rules (MathWorks, 2025a).

CNNs work by detecting patterns in images, layer by layer. The first layers pick up on basic shapes and edges, while deeper layers start recognizing complex structures like hands and fingers. This is why CNNs have been widely used for classifying static sign images, because they're great at spotting details that matter.

There is a catch here, though. 2D CNNs can only process still images, so they don't work well for signs that involve movement, like ASL letters "J" and "Z" (Kumari & Anand, 2024). To fix this, researchers developed 3D CNNs, which analyze sequences of images as opposed to just single frames. These models are more accurate but require way more computing power, making them tough to run in real-time (MathWorks, 2025a).

Since sign language does involve movement, researchers started combining CNNs with Long Short-Term Memory (LSTM) networks, which are great at understanding sequences. This approach allows the CNN to handle spatial features, such as the shape of a hand, while the LSTM tracks motion over time. One of the most effective models in this space was developed by Kumari & Anand (2024). They built a CNN-LSTM model with an attention mechanism, which helped the model focus on the most important parts of each sign, while ignoring distractions. Their model hit 84.65% accuracy on the WLASL dataset, one of the biggest collections of ASL gestures. The attention mechanism made the system more efficient and accurate, reducing common mistakes in sign recognition.

One big challenge with deep learning models is that we don't always know why they make certain decisions. This is where activation maps come in. They allow us to visualize what different layers of a CNN are focusing on. A study by MathWorks (2025a) showed that the early layers detect simple things like edges and colors, while middle layers focus on textures and basic shapes, and deep layers recognize full hand structures and finger placements. For sign language models, this is extremely useful. By looking at activation maps, researchers can check

if the AI is actually focusing on hands and gestures, or if it's being misled by background noise, lighting, or irrelevant details. This kind of visualization helps to tweak models to make them more accurate and less prone to errors (MathWorks, 2025a).

Another way to understand what a CNN is learning is through feature visualization. This technique generates images that show what the AI "sees" at different layers. MathWorks (2025b) explored this using GoogLeNet and the deepDreamImage function, which produces visual representations of the filters inside a CNN. They found that shallow layers pick up on basic colors and edges, deeper layers recognize complex patterns and textures, and fully connected layers create class-specific representations, which means that they focus on features that help the AI make its final decision.

For sign language recognition, feature visualization can help improve accuracy by ensuring that the AI is focusing on hand shapes, gestures and movement, rather than getting distracted by background elements (MathWorks, 2025b). This technique can also help refine training data and model structures, making deep learning systems more reliable in real world settings.

Even with these advancements, sign language recognition still faces some big challenges. One of these challenges is that people sign differently. Even within the same language, different people sign words slightly differently than others, which can confuse AI models (Kumari & Anand, 2024). Another challenge is that AI struggles with new environments, so a model that is trained in one setting, such as a well-lit studio, may fail when it's tested in a dimly lit room or with a different camera (MathWorks, 2025a). Computational limits are also a challenge worth considering. While powerful models can boost accuracy, they require a ton of computing power, which makes them hard to run on standard devices (MathWorks, 2025b).

In order to improve sign language AI, future researchers could focus on a few things. One area of focus is using Transformers for better sequence recognition. These models have transformed natural language processing, and they could do the same for sign language understanding (Kumari & Anand, 2024). They could also focus on adding multimodal learning, combining hand tracking, facial expressions and contextual cues, which would also make AI models more accurate (MathWorks, 2025a). And using tools like activation maps and feature visualization could help researchers fine-tune models and reduce errors (MathWorks, 2025b).

Deep learning has completely changed how we approach sign language recognition. CNNs are already incredibly powerful at classifying hand signs, and combining them with LSTMs and attention mechanisms has made them even better at tracking motion over time. At the same time, visualization techniques like activation maps and feature analysis help researchers fine-tune models, which ensures they are focusing on what really matters (hands, gestures, and movement) instead of getting distracted by background noise.

Looking ahead, the next big steps for sign language AI are better generalization, real-time processing, and increased interpretability. If researchers figure out these challenges, we could soon have real-time sign language recognition that works for everyone, everywhere, making communication easier and more inclusive.

Dataset Selection and Preprocessing

Choosing and preparing the data is important for any machine learning project. For our sign language project, we used several datasets before implementing a final CNN model on our chosen dataset.

One dataset that we initially used is called Interpret Sign Language with Deep Learning Dataset. This dataset was more diverse, including a lot of different images, which was good, but it also presented some challenges. One group member, Andy, had trouble with the sheer size of the dataset, while another, John, got high accuracy but ran into overfitting.

Another initial dataset that we used is called Sign Language MNIST. This is a pretty standard dataset for testing sign language classification. The dataset was a csv file that broke the images down into 784 pixels that ranged from 0-255. The labels ranged from 0-25 (A-Z). One of the preprocessing steps we used was to renumber the labels from 0-23 after removing J (label 9) and Z (label 25). The images in this dataset are all the same size and format which was helpful for getting our initial models up and running and for testing basic CNN architectures. This was the dataset Heather originally used, and we found that it had a good balance between accuracy and generalization, which made it a good choice to use as the dataset for our final model. We then worked on improving the model and fine-tuning it, and finally taking it to the next level with the web interface.

Because the datasets were so different, we had to standardize the data to make sure our model performed well and we could compare the results. In order to do this we had to resize the images, while keeping in mind the balance of speed of training the model and keeping the important details. We also turned the images into grayscale, which simplified things without losing too much information. This is due to the fact that sign language letters are mostly distinguished by the shape and not the color. We also normalized the pixel values to be between 0 and 1 which helps the training process. Lastly, we knew that the datasets might be unbalanced, with having too many images of some letters, so we used data augmentation techniques like rotating, scaling, and translating some images which can be helpful depending on the dataset. Through these preprocessing steps we were able to ensure consistency across datasets, which made the models and results easier to compare.

Andy and John both ended up trying models on the same dataset, but there were different issues faced with both. Andy, for example, had to downsample in order to get a better balance and John's was very accurate on the training set but was definitely overfitting. Heather's dataset

approach was okay but needed improvement. Therefore, by carefully preprocessing and standardizing the data, we were able to set ourselves up for success. This made sure our comparisons of different model configurations (number of layers, epochs, image size) were fair and meaningful.

Model Development and Training

For our first model, we all experimented with different values and datasets but settled on John's initial model. It had a num class of 29(space, delete, nothing, and a-z). We used a batch size of 32 and had a total of 6 layers. Two convolutional layers, two pooling layers, and lastly two dense layers. For the function loss, the model used categorical_crossentropy. For the optimizer we used Adam(). Activation functions included reLu and softmax. For the convolutional layer, we started at 32 and then went up to 64 for the second layer. The pooling was (2,2) for both and for the dense layer, we used 128 for the first one and our total class number for the second. We also did not introduce any data augmentation. We used 10 epochs and added early stopping to help with overfitting though this did not do much to stop it. Lastly, the images were RGB and we used raw images.

For the second model, we made several improvements and changes to bring our numbers up. The first change added was removing several of the classes to simplify the model and make it more precise. We trimmed a couple of classes including J and Z. This is because both require motion which we cannot train accurately using just images. We also remove the space, delete, and nothing. The data set we chose to stick with did not include these by default anyway. We then took our previous model and built it on top of it using our new final chosen dataset.

In order to help augment the data, we used a datagen fit. This fit took the images and applied 4 random transformations. The first rotated each image by a value between -10 and +10 degrees. The second zoomed into the images by a value up to 10%. The third shifted the images either left or right up to a value of 10% of the images width. The fourth randomly shifted the images either up or down up to a value of 10% of the images width. The goal of these changes is to add variation to the data set so that even though they all came from the same training set, they would be viewed differently. The other issue we faced was that 10 epochs were then way too much. So the final adjustment was decreasing our epochs from 10 to just 5 and removing the early stop. At this point, our model did not need it and would hit 99 pretty much every time at 5 exactly. For the activation functions and optimizer, we used the same as the first model.

For our group’s model, we used a sequential model which allowed us to add multiple layers. We increased our batch size from 32 to 64 which would result in about 343 batches of 64 images each. We had 3 convolutional layers of size 32, 64 and 128 and used a relu activation on each of those layers. The formula for relu activations is Relu(z) = max(0, z) (Classical ML Equations in LaTeX) which indicates that if the input is negative, it will return 0, if it is positive, it will return whatever that positive value was. The purpose of each of the layers was to pull out a

different number of features using a filter laid over the image (32, 64 and 128 images respectively). In between each convolutional layer, we used a MaxPooling feature. The purpose of the MaxPooling is to break each image down into 2 by 2 blocks where it then extracts the maximum value for each of those blocks. After the 3 convolutional layers and the pooling, we flattened the images which converted the pooled data into a single column vecto9. After flattening, we used another dense layer (256 features), and finally a dense layer with 24 layers, only this time we used a softmax activation function. The sotfmax activation function outputs the results as a probability with each class representing the probability that the input belongs to that class. The softmax formula is:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K$$

(LaTeX Documentation).

Lastly, we used Adam compilation with categorical crossentropy. Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. According to Kingma et al., 2014, the method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters". We used the default training rate of .001. We also chose categorical crossentropy. The mathematical formula for categorical cross entropy is as follows:

$$L(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Where:

- $L(y, \hat{y})$ is the categorical cross-entropy loss.
- y_i is the true label (0 or 1 for each class) from the one-hot encoded target vector.
- $\hat{y}_i$ is the predicted probability for class i.
- C is the number of classes.

(GeeksforGeeks)

Categorical cross entropy is used when you have more than 2 target categories and it measures the difference between the predicted probability and the true distribution. It penalizes predictions based on how confident the model is about the correct class.

The initial model also did not include a validation strategy so for this model we used the test split on our training data for better generalization monitoring. This left a training set of 21,964 images, a validation set of 5491 images and a testing set of 7172 images. These changes alone improved the model significantly but still feared overfitting and accuracy problems.

The following table illustrates the setup for the final model.

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 48, 48, 32)	320
max_pooling2d_6 (MaxPooling2D)	(None, 24, 24, 32)	0
conv2d_7 (Conv2D)	(None, 22, 22, 64)	18,496
max_pooling2d_7 (MaxPooling2D)	(None, 11, 11, 64)	0
conv2d_8 (Conv2D)	(None, 9, 9, 128)	73,856
max_pooling2d_8 (MaxPooling2D)	(None, 4, 4, 128)	0
flatten_2 (Flatten)	(None, 2048)	0
dense_4 (Dense)	(None, 256)	524,544
dense_5 (Dense)	(None, 24)	6,168

Total params: 1,870,154 (7.13 MB)
Trainable params: 623,384 (2.38 MB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 1,246,770 (4.76 MB)

The following tables illustrate the setup for the other 3 models:

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 49, 49, 32)	416
max_pooling2d (MaxPooling2D)	(None, 24, 24, 32)	0
conv2d_1 (Conv2D)	(None, 23, 23, 64)	8,256
max_pooling2d_1 (MaxPooling2D)	(None, 11, 11, 64)	0
flatten (Flatten)	(None, 7744)	0
dense (Dense)	(None, 128)	991,360
dense_1 (Dense)	(None, 29)	3,741

Total params: 3,011,321 (11.49 MB)
Trainable params: 1,003,773 (3.83 MB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 2,007,548 (7.66 MB)

John's original model used a batch size of 32 and had conv2D layers with 32 and 64 features, then an additional dense layer with 128 features and a final dense layer that used softmax activation with the 29 features (representing the 29 possible classes).

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_7 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_3 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_8 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_4 (Conv2D)	(None, 4, 4, 128)	73,856
max_pooling2d_9 (MaxPooling2D)	(None, 2, 2, 128)	0
flatten_1 (Flatten)	(None, 512)	0
dense_1 (Dense)	(None, 128)	65,664
dropout_1 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 26)	3,354

Total params: 162,266 (633.85 KB)
Trainable params: 162,266 (633.85 KB)
Non-trainable params: 0 (0.00 B)

Andy's model went through some changes as the process developed. Originally, he only had 2 convolution layers with 32 and 64 features, but he was able to improve that to layers with 32, 64, and 128 features. He used a batch size of 64. After flattening the data, he ran a dropout layer before running the softmax activation on a final dense layer of the 26 target classes.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
batch_normalization (BatchNormalization)	(None, 26, 26, 32)	128
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 11, 11, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 128)	73,856
batch_normalization_2 (BatchNormalization)	(None, 3, 3, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 1, 1, 128)	0
flatten (Flatten)	(None, 128)	0
dense (Dense)	(None, 256)	33,024
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 25)	6,425

Total params: 133,017 (519.60 KB)
Trainable params: 132,569 (517.85 KB)
Non-trainable params: 448 (1.75 KB)

Heather's model used a batch size of 64 and had convolutional layers of 32, 64 and 128 features using relu activation, a dense layer with 256 features after flattening then a dropout layer and a final dense layer with the 25 target classes using softmax activation. Heather also included batch normalization before pooling.

The preprocessing and model features we wanted to compare were the number of training images, number of testing images, size of input images, batch size, number and size of layers, and number of epochs run. The following table illustrates those differences:

Features of the different models								
	Training images	Testing images	Image size	Batch size	Layers	Epoch	Accuracy	Validation accuracy
Andy	10,400	26	32*32	64	32-64-128 Dense(26)	10	84%	96%
Heather	27,455	7172	26*26	64	32-64-128 Dense(128) Dense(25)	20	99.75%	88%
John	87,000	29	50*50	32	32-64-128 Dense(128) Dense(29)	10	99%	99%
Final	27,455	7172	50*50	64	32-64-128 Dense(256) Dense(24)	20	99%	99%

Experimental Results and Analysis

To compare our models preprocessing and setup, we used accuracy, recall, F1 score and precision. These formulas rely on the number of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). Accuracy is the ability to correctly identify the target classes and can be found by dividing the correct predictions by the total number of predictions. Recall (or True Positive Rate) measures how many of the true cases the model was able to identify. Precision calculates how many of the predicted positive cases were actually positive. F1 score balances the recall and precision of the model. (Kumari & Anand, 2024)

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

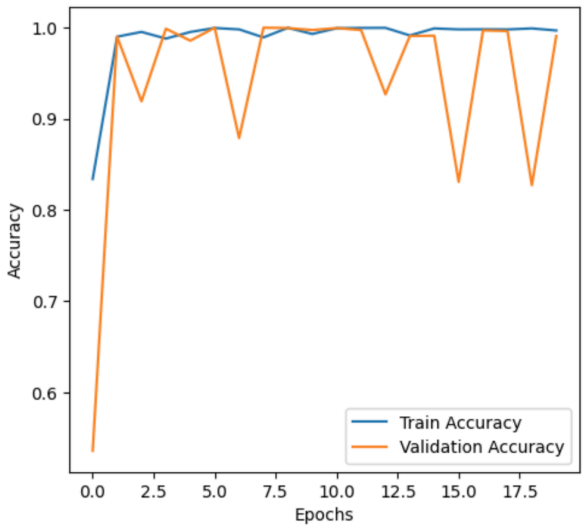
$$Recall = \frac{TP}{TP + FN}$$

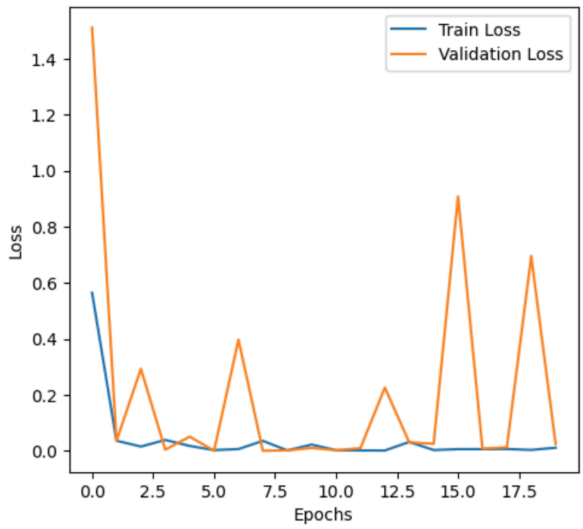
$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN}$$

Visualization of Data

The following descriptions and visualizations represent 1 of the runs of the model for each person.

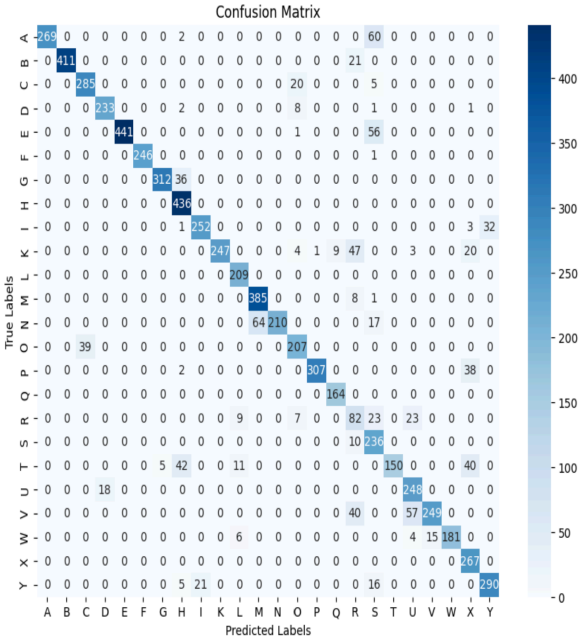
Visualization of Results from Heather's Model





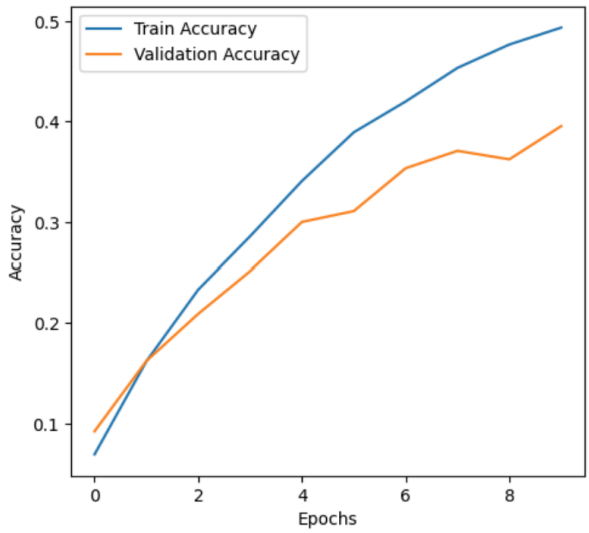
Classification Report:

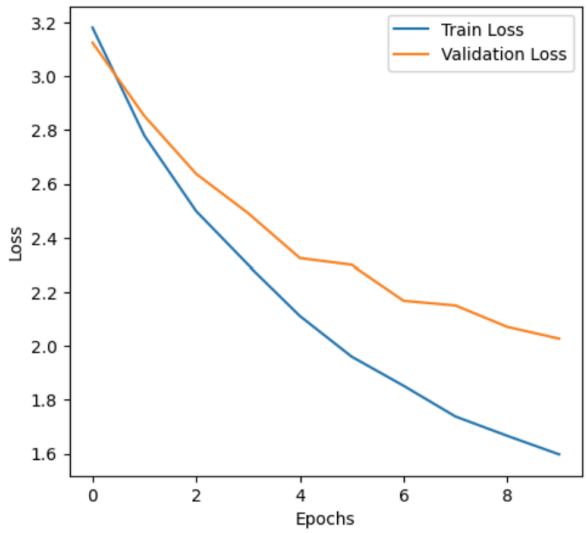
	precision	recall	f1-score	support
A	1.00	0.81	0.90	331
B	1.00	0.95	0.98	432
C	0.88	0.92	0.90	310
D	0.93	0.95	0.94	245
E	1.00	0.89	0.94	498
F	1.00	1.00	1.00	247
G	0.98	0.90	0.94	348
H	0.83	1.00	0.91	436
I	0.92	0.88	0.90	288
K	1.00	0.75	0.85	331
L	0.89	1.00	0.94	209
M	0.86	0.98	0.91	394
N	1.00	0.72	0.84	291
O	0.84	0.84	0.84	246
P	1.00	0.88	0.94	347
Q	0.95	1.00	0.97	164
R	0.39	0.57	0.47	144
S	0.57	0.96	0.71	246
T	1.00	0.60	0.75	248
U	0.74	0.93	0.83	266
V	0.94	0.72	0.82	346
W	1.00	0.88	0.94	206
X	0.72	1.00	0.84	267
Y	0.90	0.87	0.89	332
accuracy			0.88	7172
macro avg	0.89	0.87	0.87	7172
weighted avg	0.91	0.88	0.88	7172



Heather's model resulted in an average precision of 89%, an average recall and F1-score of 87% and an accuracy of 89%. The letters that had the best precision (100%) were A, B, E, F, K, N, P, T, and W. The worst precision rates were R (39%) and S (57%). The letters that had the best recall (100%) were F, H, Q, and X, while the worst were R (57%) and T (60%). The letter F was the only letter with a perfect F1-score (100%) while the worst were R (47%) S (71%) and T (75%). Overall Heather's model was the best at predicting the letter F with a perfect precision, recall and F1 and was the worst with the letter R with it having the worst precision, recall and F1-Score. When considering the accuracy and loss graphs, the validation accuracy and loss was much more inconsistent than the training data from epoch to epoch.

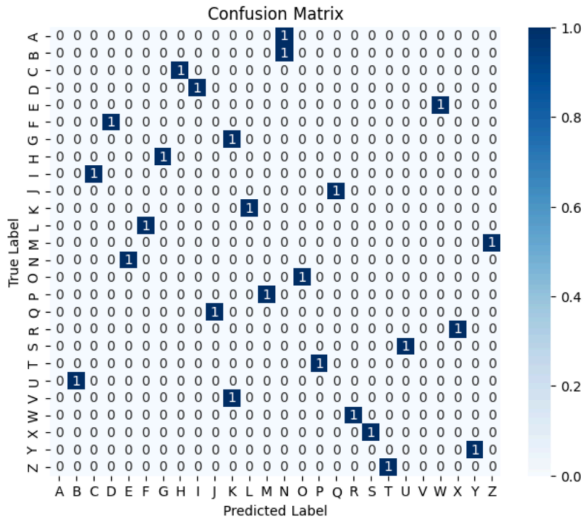
Visualization of Results from Andy's Model





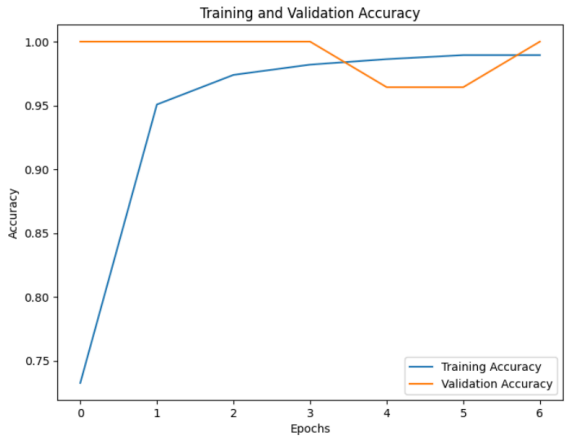
Predicted Label

	precision	recall	f1-score	support
A	0.00	0.00	0.00	1
B	0.00	0.00	0.00	1
C	0.00	0.00	0.00	1
D	0.00	0.00	0.00	1
E	0.00	0.00	0.00	1
F	0.00	0.00	0.00	1
G	0.00	0.00	0.00	1
H	0.00	0.00	0.00	1
I	0.00	0.00	0.00	1
J	0.00	0.00	0.00	1
K	0.00	0.00	0.00	1
L	0.00	0.00	0.00	1
M	0.00	0.00	0.00	1
N	0.00	0.00	0.00	1
O	1.00	1.00	1.00	1
P	0.00	0.00	0.00	1
Q	0.00	0.00	0.00	1
R	0.00	0.00	0.00	1
S	0.00	0.00	0.00	1
T	0.00	0.00	0.00	1
U	0.00	0.00	0.00	1
V	0.00	0.00	0.00	1
W	0.00	0.00	0.00	1
X	0.00	0.00	0.00	1
Y	1.00	1.00	1.00	1
Z	0.00	0.00	0.00	1
accuracy			0.08	26
macro avg	0.08	0.08	0.08	26
weighted avg	0.08	0.08	0.08	26



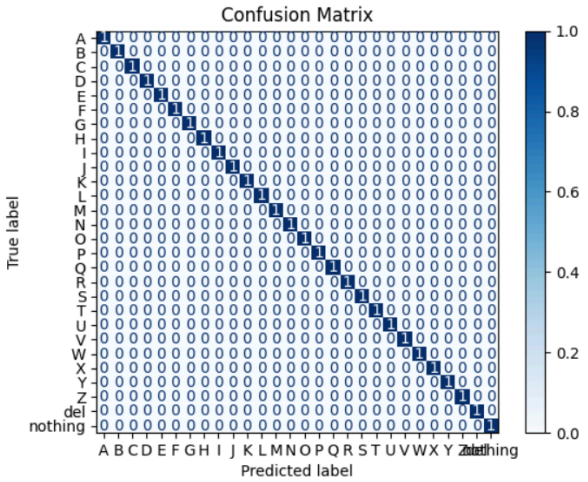
In adapting the code to see if we could reconcile the accuracy with the confusion matrix, we changed the preprocessing for Andy's model to create an 80-20 train/validation split. Andy's predictions on the training set from the sample of 80000 (between 500-1000 per letter) improved from 60% to 88% after adding another layer. Despite being higher, when comparing the validation accuracy (39%) against the accuracy when running a confusion matrix against the test set, the model performed terribly with scores at 8% after running the model multiple times. Overall, we could see that while it might be able to improve on the training data, any data that wasn't part of that dataset would confuse the model and result in false predictions. Overall, this model didn't do well at all. Efforts were made to improve the model such as adding more layers, dropout, batch normalization and some of the pre-processing that John and Heather implemented like zoom, moving up and down, left and right, etc. Ultimately, adding the train-test split reduced the training examples to 10,400 meaning that it couldn't improve to make a useful prediction and only having one example per letter meant that its predicting the test set was extremely inaccurate. Because of the irreconcilable deficiencies of this model, we moved on to other models.

Visualization of Results from John's Model



Visualization of Results from John's Model

	precision	recall	f1-score	support
A	1.00	1.00	1.00	1
B	1.00	1.00	1.00	1
C	1.00	1.00	1.00	1
D	1.00	1.00	1.00	1
E	1.00	1.00	1.00	1
F	1.00	1.00	1.00	1
G	1.00	1.00	1.00	1
H	1.00	1.00	1.00	1
I	1.00	1.00	1.00	1
J	1.00	1.00	1.00	1
K	1.00	1.00	1.00	1
L	1.00	1.00	1.00	1
M	1.00	1.00	1.00	1
N	1.00	1.00	1.00	1
O	1.00	1.00	1.00	1
P	1.00	1.00	1.00	1
Q	1.00	1.00	1.00	1
R	1.00	1.00	1.00	1
S	1.00	1.00	1.00	1
T	1.00	1.00	1.00	1
U	1.00	1.00	1.00	1
V	1.00	1.00	1.00	1
W	1.00	1.00	1.00	1
X	1.00	1.00	1.00	1
Y	1.00	1.00	1.00	1
Z	1.00	1.00	1.00	1
del	1.00	1.00	1.00	1
nothing	1.00	1.00	1.00	1
accuracy			1.00	28
macro avg	1.00	1.00	1.00	28
weighted avg	1.00	1.00	1.00	28



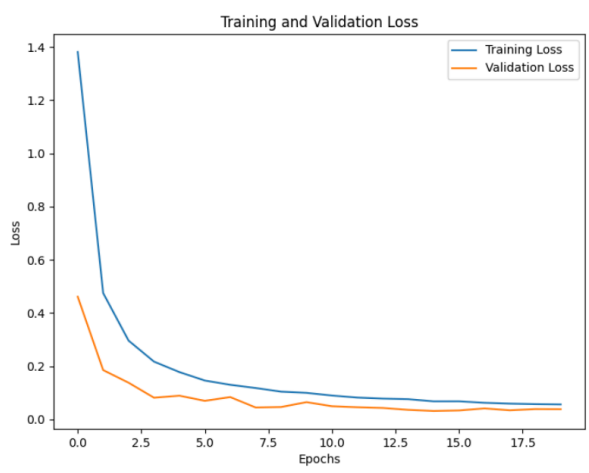
John's predictions from the 87000 images prediction was 100%. As he was the only one with the computing power needed to run that many images, he didn't have to use the sampling methods that Andy did. The model was perfect in all scores for both training and testing and the one image that was present for the testing data was identified correctly (as seen in the confusion matrix). Despite the great scores, we determined that we needed a different model to work with since we feared overfitting.

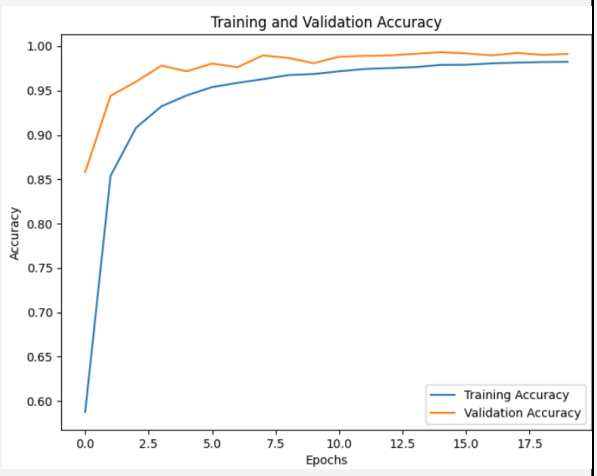
The final model

To improve our final model, we started with Heather's data since it had performed the best. Initially, in version one, we were able to get the model to improve to close to 99% in all categories. The main problem we ran into was that when we introduced it to outside images from the API interface, the quality of the images changed so much that the model was not very good at predicting any of the letters. To attempt to fix that problem, we combined training data from multiple training sets. Because we now had 115,157 training images, we had to set the

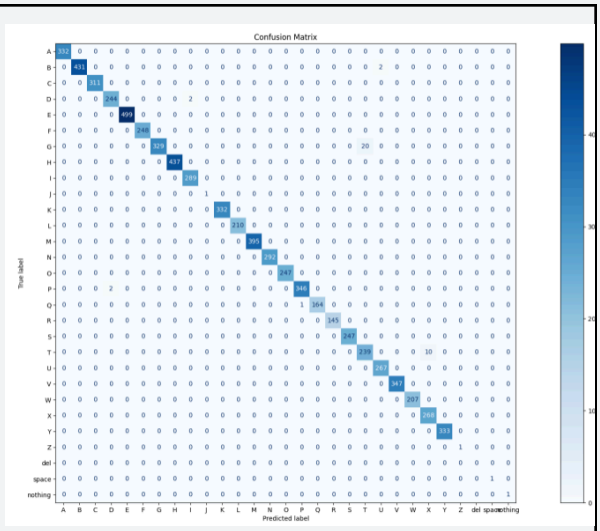
model up differently. Most notably, we increased the epochs from 5 to 20. Also, an additional dense layer of size 256 was added on top of the previous layers of 32, 64,and 128 before the final dense layer of 29 to categorize letters into our 29 classes. We only used flattening and maxPooling and did not include any dropout.

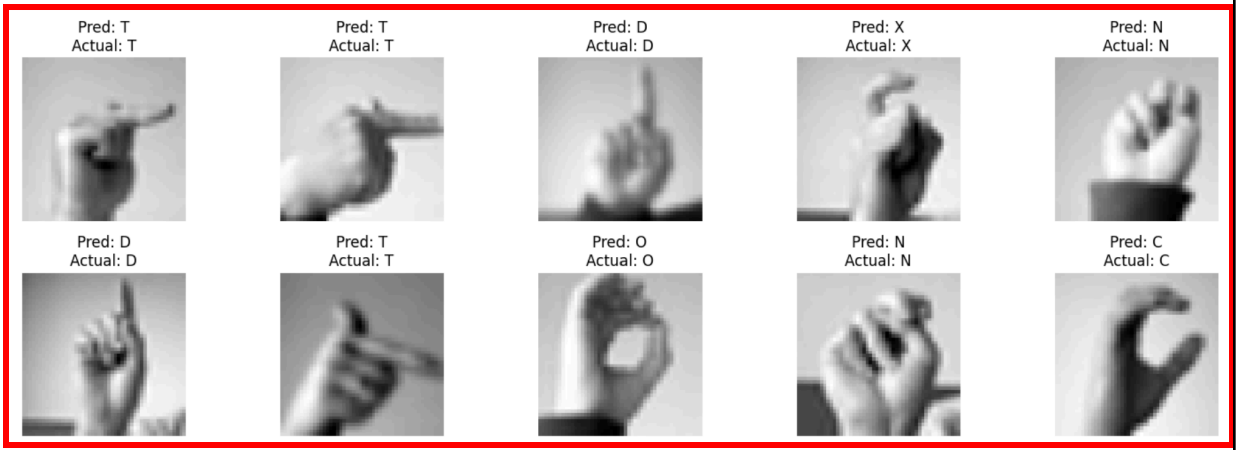
Visualization of Results from New Training Data





	precision	recall	f1-score	support
A	1.00	1.00	1.00	332
B	1.00	1.00	1.00	433
C	1.00	1.00	1.00	311
D	0.99	0.99	0.99	246
E	1.00	1.00	1.00	499
F	1.00	1.00	1.00	248
G	1.00	0.94	0.97	349
H	1.00	1.00	1.00	437
I	0.99	1.00	1.00	289
J	1.00	1.00	1.00	1
K	1.00	1.00	1.00	332
L	1.00	1.00	1.00	210
M	1.00	1.00	1.00	395
N	1.00	1.00	1.00	292
O	1.00	1.00	1.00	247
P	1.00	0.99	1.00	348
Q	1.00	0.99	1.00	165
R	1.00	1.00	1.00	145
S	1.00	1.00	1.00	247
T	0.92	0.96	0.94	249
U	0.99	1.00	1.00	267
V	1.00	1.00	1.00	347
W	1.00	1.00	1.00	207
X	0.96	1.00	0.98	268
Y	1.00	1.00	1.00	333
Z	1.00	1.00	1.00	1
del	0.00	0.00	0.00	0
space	1.00	1.00	1.00	1
nothing	1.00	1.00	1.00	1
accuracy			0.99	7200
macro avg	0.96	0.96	0.96	7200
weighted avg	1.00	0.99	0.99	7200

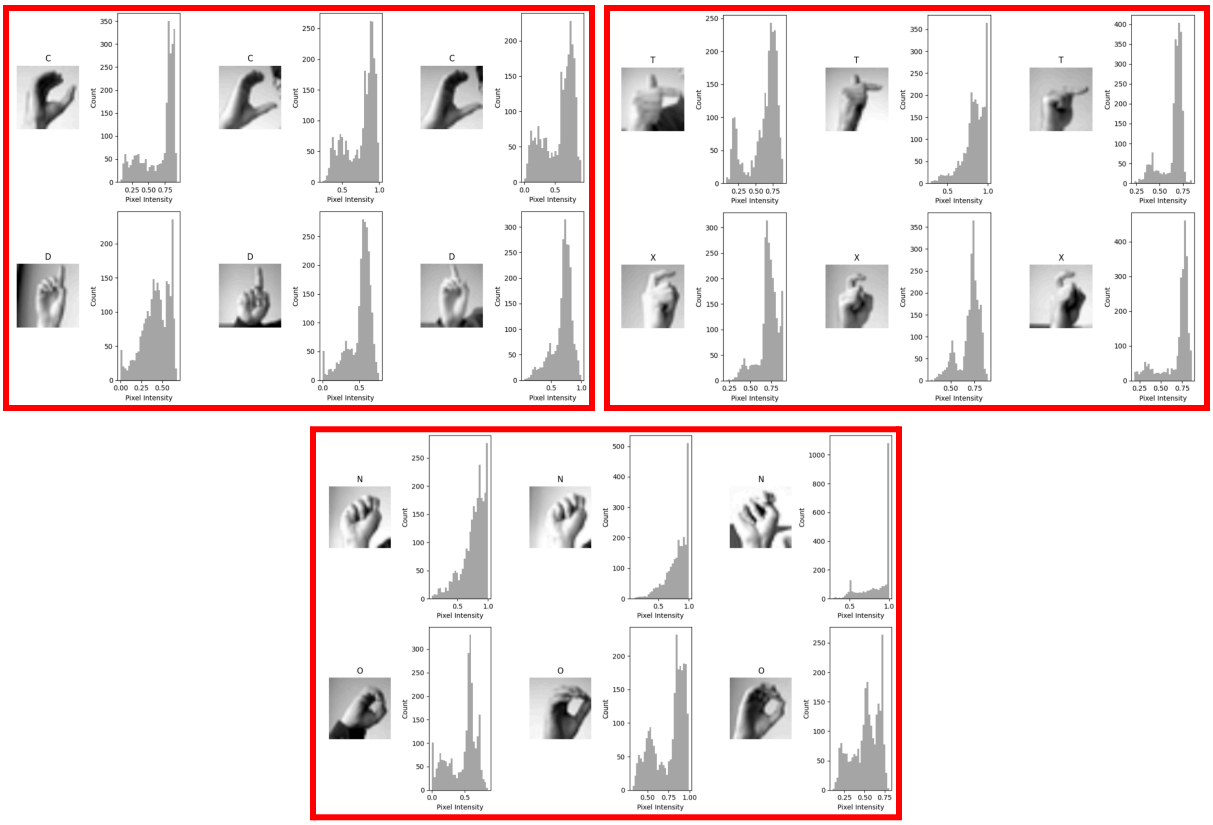




Our group's final model had an average precision, recall, F1-Score and accuracy of 99%. Only 8 did not hit 100% precision (R (92%) X (93%) I (94%) E (96%) G (98%) N (99%) P (99%) U (98%), only 8 did not hit 100% recall (T (88%) S (91%) Y (95%) D (97%) G (99%) K (99%) N (99%) X (99%)), however, there were 12 that didn't hit 100% F1-Score (T (94%) S (95%) X (95%) R (96%) I (97%) Y (97%) D (98%) E (98%) G (98%) N (99%) P (99%) U (99%). The model performed well for all letters except G, N and X, as they were the only ones not perfect in all 3 categories. The model was worst for the letter T as they both had a 88% recall and 94% (both the worst for their respective categories. We felt that this model did a great job of correctly predicting both the training and testing data, yet when we hooked it into a web service and downloaded new images into it, the number of misclassifications slightly increased.

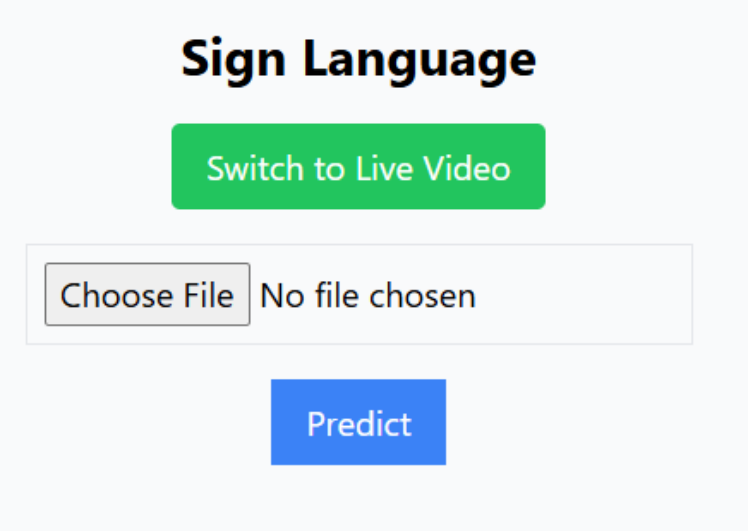
To consider why certain letters were more inaccurate than others, we looked at the confusion matrix and model scores (accuracy, precision, recall and F1-score). While every time the model was run, you would get a different confusion matrix and scores, we chose to consider 6 of the problem letters that showed up in multiple runs of the model fitting (C,D,N,O,T,X). If you just look at the images, it would not seem like it would be easy to confuse one letter for another, but because there is different shape, intensity, distance, clarity of image and so forth, the model's best guess is sometimes going to be a misclassification.

When looking at the pixel intensity, it becomes a little more obvious why this process is so difficult for our model. Despite looking at the same letters, depending on how light or dark the background is, it would have different levels of difficulty in distinguishing the hand from the objects in the background. Even within the same letter, there are slight (and sometimes drastic) differences between the letters. It would be a difficult task for a model to come up with a general rule when there isn't a set pixel intensity pattern for each letter because of different features of a picture.



Pixel intensity graphs for various letters

After finalizing the model, we hooked it into an API that would allow users to select any image of a sign language letter (excluding J and Z) and the input would be sent through the model where it would output the predicted letter. The initial results were not great and were far less than the 99% accuracy that the model had obtained. Because of these results, we determined that our model was overfitting the input data and would not be reliable for anyone using data that it wasn't trained in. As we look to continue to expand on this model, we realize that we would need to spend a lot more time distinguishing the features of the input that we used to train our model so we could normalize them and make the images that are coming in via the camera more similar and hopefully performs as well as our model did. Our attempt at a solution to this problem was to expand the training data and try to get multiple versions of the letters that were taken from different perspectives. Ultimately to make the API as useful as we would like we would need to include closer, farther, multi-colored, black and white, hand-drawn, different lighting, etc. so the model wouldn't be confused by future input that it hadn't seen before.



Challenges in Model Development and Deployment

Building a high-accuracy CNN model for sign language recognition was a rewarding experience, but it definitely wasn't without its challenges. One of the biggest hurdles we faced was overfitting—where the model performed amazingly well on the training data but struggled when tested on completely new images. At first, this was especially noticeable in Andy's model, which had an almost perfect accuracy during training but completely fell apart when predicting real test images. The issue was that the model was too good at memorizing patterns from the training data instead of actually learning general features that could apply to unseen images. To fix this, we used data augmentation, meaning we rotated, zoomed, and shifted images to help the model get used to variations it might see in real-world conditions. This made a big difference in improving generalization.

Another issue we ran into was class imbalance. Some letters had way more training examples than others, which meant the model naturally leaned toward predicting the more common ones. For example, if there were twice as many pictures of "A" as "X," the model would get really confident about guessing "A" just because it saw it more often. We tackled this by using balanced sampling, making sure all letters were represented more evenly in training. We also adjusted the model's loss function to give underrepresented classes more weight, which helped improve performance across the board.

Then there was the challenge of moving from a lab-trained model to real-world use. The images we trained on were clear and well-lit, but real-world conditions are a whole different story. Imagine someone trying to use this model in a dimly lit room or with a shaky webcam—it could completely throw off the predictions. Even small changes in hand positioning, background clutter, or camera angles can impact accuracy.

Another roadblock for us was the issue of computational limits. Deep learning models, especially ones with multiple layers and high-resolution inputs, need a ton of processing power. Running this on a high-end GPU would be no problem, but running it on an average laptop or mobile device is where things usually slow down.

And finally, there's the issue of scalability. Our model worked really well on the dataset we trained it on, but for sign language recognition to be truly useful, it needs to handle different signers, lighting conditions, and even variations in regional dialects. Some sign languages incorporate facial expressions and slight hand movements that aren't easy to capture in still images.

Conclusion and Future Work

There were significant differences among the models. Ignoring Andy's 8% match model, we saw a significant jump in F1-score between Heather's model and the group's final model (88% to 99% respectively). There was also a significant difference in the accuracy and loss graphs. Andy's was consistent, but overall very low. Heather's had wild swings between 90% and 100%. As the final scores approached 99%, we didn't feel that further epochs would improve it drastically and would probably start to lean towards overfitting. Our final model had more consistent growth across all 5 epochs.

We would like to figure out why Andy's model had such a high accuracy, but couldn't correctly identify any of the letters correctly. Since it ran similar layers to the other two, there must be something in the input or the labels where it is not registering the letter correctly. The data used in that model was different from the other dataset as it was just images and they were running pixels, so there may be something there.

We would also like to go through the scores and figure out why certain letters typically did better or worse across all models. We would have to look at the shape of the letter and see if it is easier to distinguish, but we would also have to look at the files themselves to figure out if it was the letter or the condition of the input (lighting, size, RGB value, quality of the photo, etc.) that made it easier for our model to work.

Lastly, we would like to improve our input for our API so that it can not only take in images, but videos. This would allow us to start training the model to interpret signs that are more than just a stationary frame and would also allow us to train the model to interpret J and Z which require motion. In order to do this, we will need to change from a CNN to a RNN that is capable of breaking apart the frames and learning the meaning of signs over time.

Overall, we're proud of the model we built, though there's still a lot of room to grow. With the right improvements, this project could be expanded upon to eventually turn into a real-world tool that makes communication easier for the deaf and hard-of-hearing community.

References

Datamunge. (n.d.). *Sign Language MNIST* [Data set]. Kaggle. https://www.kaggle.com/datasets/datamunge/sign-language-mnist?select=amer_sign2.png

GeeksforGeeks. (n.d.). *Categorical cross-entropy in multi-class classification.* https://www.geeksforgeeks.org/categorical-cross-entropy-in-multi-class-classification/

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning.* MIT Press.

Kumari, D., & Anand, R. S. (2024). Isolated Video-Based Sign Language Recognition Using a Hybrid CNN-LSTM Framework Based on Attention Mechanism. *Electronics, 13(7)*, 1229. https://doi.org/10.3390/electronics13071229

LaTeX Documentation. (n.d.). *Classical ML equations in LaTeX.* https://blmoistawinde.github.io/ml_equations_latex/

Mooney, P. (n.d.). *Interpret Sign Language with Deep Learning* [Data set]. Kaggle. https://www.kaggle.com/code/paultimothymooney/interpret-sign-language-with-deep-learning/notebook

Visualize Activations of a Convolutional Neural Network. (2025). Mathworks.com. www.mathworks.com/help/deeplearning/ug/visualize-activations-of-a-convolutional-neural-network.html

Visualize Features of a Convolutional Neural Network. (2025). Mathworks.com. https://www.mathworks.com/help/deeplearning/ug/visualize-features-of-a-convolutional-neural-network.html