CNN Classification of Basic American Sign Language

Jack Miller, Jerry Yang, Aidan Shea,

Shane Fitzpatrick, Joseph Fallait, and Sean Curley

CS3345.01

12/9/2020

Abstract

This program accepts images of American Sign Language fingerspelling and classifies the input into one of 29 symbols: the 26 letters of the alphabet, space, delete, and nothing. We hoped to create an application that could aid people who are deaf or hard of hearing and expand the accessibility of online products that target those communities. Beyond the project's practical application, we also hoped to write an in-depth explanation of our selected computer vision approach for our classmates who might be curious about the field.

We researched several different options for computer vision before settling on a convolutional neural network. The rest of our work involved considerations for data preprocessing, neural network architecture, and model regularization, all in service of reducing overfitting to variance in the dataset. Finally, we considered a body of evaluation metrics that would be useful in the real-world application of the program. Although we initially hoped to integrate the program with a video input feed and a predictive language model, we abandoned those plans to focus on the details of CNN implementation and regularization.

Introduction

Introduction to ASL and Computer Vision ASL

American Sign Language, or ASL, is a language with the same properties as typical spoken languages with grammatical differences from English.[1] The language is used predominantly by North Americans who are deaf or hard of hearing, and different countries or regions typically have their own very distinct forms of the language. ASL has its own distinct

[1] American Sign Language. (2020, November 05). Retrieved December 03, 2020, from https://www.nidcd.nih.gov/health/american-sign-language

ways of expressing ideas or emotions. For example, an ASL user might ask a question by "raising their eyebrows, widening their eyes, and tilting their bodies forward," while an English speaker would simply indicate a question via a change in tone or pitch.[2] A key aspect of ASL is the use of fingerspelling. Fingerspelling is the process of using distinct handshapes (one for each letter of the English alphabet) to spell out words. This makes it particularly helpful for individuals learning ASL, but some people prefer to fingerspell even for words with ASL signs, making it a highly versatile way of communicating.[3]

There are many uses of an application that could translate American Sign Language into English, especially as workplaces and offices across the world begin to rightfully make a shift towards accessibility. Groups like SignAll have devoted time and resources to develop technology capable of translating ASL to English using AI and computer vision.[4] Their stated uses of this technology are to "assist employers to integrate deaf employees into hearing-centric workplaces", help hearing colleagues learn basic ASL vocabulary, make products more accessible for deaf employees, and compliment sign language classes for students learning ASL.[5] Likewise, these uses motivate our own exploration of computer vision translation of ASL fingerspelling.

Introduction to Neural Networks and Convolution Neural Networks

We decided on using a convolutional neural network (or CNN) for our computer vision task, which falls under the umbrella of neural networks and deep learning. CNNs were ultimately decided on as our model of choice because a "CNN can achieve extreme accuracy in image classification, because this model makes use of convolution and subsampling layers to extract the

[2] Ibid.
[3] Signing Savvy, L. (n.d.). Fingerspelling. Retrieved December 03, 2020, from https://www.signingsavvy.com/fingerspelling
[4] SignAll. (n.d.). Retrieved December 04, 2020, from https://www.signall.us/
[5] Ibid.

most relevant features of the image."[6] In other words, a CNN uses convolution layers in order to help identify set features, such as a diagonal finger or an edge of a hand regardless of where in the image those features may be present. CNNs also include pooling layers which reduce the size of the feature map created by a preceding convolution layer. It is this process of identifying defining features and then reducing the complexity of those features in the image that allows for CNNs to work particularly effectively when it comes to computer vision tasks.

This bodes well for our application, as in theory a CNN should be proficient and the feature extraction and subsequent classification necessary. There are also numerous similar applications created by others which successfully use a CNN for ASL symbol recognition, which is another reason that CNNs were eventually chosen as our final model type.

Methods

Data

Our data came from combining two datasets on Kaggle, a free platform that crowdsources data for public use. The first dataset was called "ASL Alphabet" and contained the 26 letters of the alphabet as well as three other symbols: space, delete, and nothing[7]. Each symbol had exactly 3,000 images of 200x200 pixels, which we resized down to 64x64 as we read them into the program. All in all, there were 87,000 images in this dataset. Unfortunately, however, the images in this dataset were all similar—they were only based on one participant's

[6] C. J. L. Flores, A. E. G. Cutipa and R. L. Enciso, "Application of convolutional neural networks for static hand gestures recognition under different invariant features," 2017 IEEE XXIV International Conference on Electronics, Electrical Engineering and Computing (INTERCON), Cusco, 2017, pp. 1-4, doi: 10.1109/INTERCON.2017.8079727.
[7] Accessed at https://www.kaggle.com/grassknoted/asl-alphabet.

right hand, were taken in the same environment, and had minimal alterations to lighting.

Three examples of the letter "C" from the first dataset.

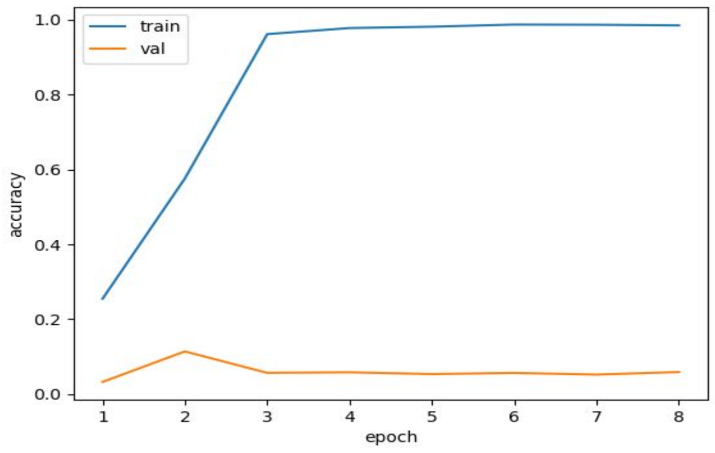
The lack of diversity was a problem in the initial stages of the project because the model found it so easy to overfit to the homogenous input that any further steps, even heavy regularization, would cause just minor shifts in performance. We were especially concerned that if a nonwhite user or someone wearing gloves attempted to use the program, the neural network would expect certain skin tones and struggle to extract features. Alternatively, new environments and lighting levels could confuse the model. Perhaps most importantly, none of the photos induced other body parts, like a chest or head. The model would never learn to ignore these components and become almost unusable in real-world contexts.

We sought out a second data source to expand the universe of training images. We found the "Significant (ASL) Sign Language Alphabet Dataset," which was also on Kaggle[8]. This dataset had all 26 letters and the space character; each symbol had between 2,700 and 3,200 examples. Overall, this second dataset had 77,500 images in it. This data was much more diverse, although the subjects were again entirely white.

[8] Accessed at https://www.kaggle.com/kuzivakwashe/significant-asl-sign-language-alphabet-dataset.

Three examples of the letter “C” from the second dataset.

Training a basic CNN on the first “clean” dataset and testing on the second “messy” yielded clear evidence of overfitting. Consequently, we opted to combine the two into one dataset of 164,500 images; the first dataset was still useful for giving clear examples of each gesture’s features while the second could provide some much-needed variance.



A simple CNN trained on the “clean” data and tested on the “messy” data.

<u>Data Preprocessing</u>

To further improve generalizability, we decided to create a through preprocessing pipeline for all images. Data augmentation tackles overfitting at the root of the problem by increasing the diversity of the training set. In broad terms, data augmentation aids generalizability by limiting the extent to which variance in the training data all “points” in the same direction and incentivizes overfitting to features or elements that are not present in the full population. Our solution was to alter this framing and create more noise in the images via horizontally shifting, vertically shifting, and zooming to a random extent of 0 to 20%. This was done via the keras image preprocessing package, with the shifting and zooming function being applied to a random extent on an image by image basis while loading in our training data.

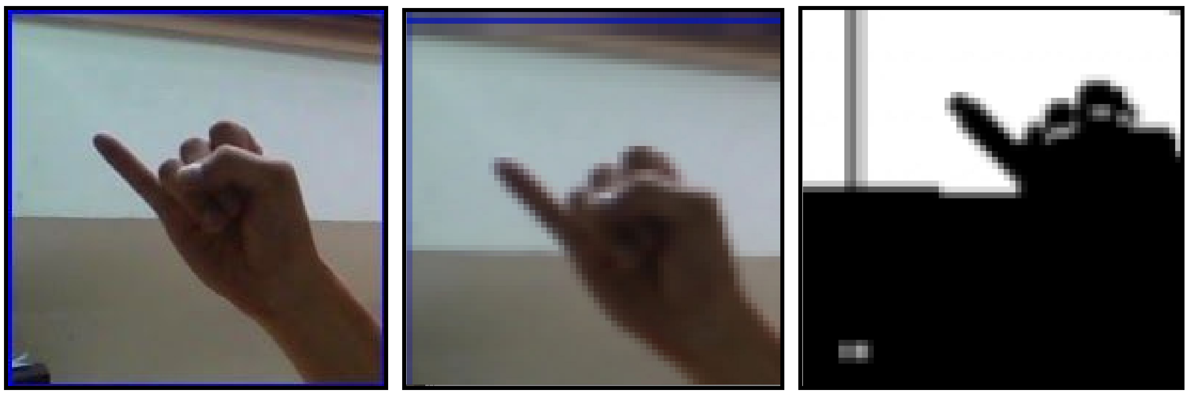
Segmenting the image into the object of focus and other elements is a crucial and widely-used technique in the field of hand gesture recognition.[9] This was especially crucial given the limited diversity in our images—training the model on just white hands will teach it to recognize just certain colors (i.e. skin tones) as the object of focus. Simply transforming an image into black and white based on the RGB values of pixels would not work, however. Variations in lighting and the background could erase portions of the hand.

Instead, we found useful research on a “k-means” approach to hand segmentation. The paper found that using the YCrCb color space was superior to the RGB color space in segmenting an image. The traditional RGB color scheme is not ideal for sorting pixels into the foreground versus background because the skin color values correlate heavily with luminance (another word for brightness).[10]Additionally, the wide possible variation in human skin tone means that it is possible that the background is a similar color as the hand. The YCrCb space measures, in order, luminance, blue chroma (blueness controlled for luminance), and red chroma (redness controlled for luminance)[11]. It is better for segmentation because it explicitly accounts

[9] Zhagn Qiu-yu, et al. “Hand Gesture Segmentation Method Based on YCbCr Color Space and K-Means Clustering,” in *International Journal of Signal Processing, Image Processing, and Pattern Recognition 8*. pg 105.
[10]Ibid 106.
[11]Ibid 108.

for brightness and can recognize the difference between pixels of the same brightness or find similarities between pixels of different brightnesses.

After transforming the image into the YCrCb space, the program uses a k-means clustering algorithm to sort the pixels into one of two classes: hand or background. The k-means algorithm iteratively groups the pixels (or whatever input) into one of two classes based on similarity to the "average" member of that class. Then, it calculates the new "average" pixel in that class and resorts every pixel again. This repeats until some threshold of similarity within each class has been passed. It doesn't transform anything into black and white but rather into one of these two classes but for our purposes, one class can be visualized by white pixels and the other black. This approach is advantageous because it allows the particulars of each image to speak for itself in the segmentation process. If the whole image is fairly dark, for example, the k-means sorting will bake that in because it can only see each pixel relative to pixels from the same image.[12] Because the k-means process is entirely contained within the image itself, there is no consistency between whether the hand is "white" in one photo and "black" in another.



An image as it is randomized and then sorted into foreground and background pixels.

[12]Ibid 114.

The code uses Python's scikit MiniBatchKMeans model to classify the pixels of each image. This implementation follows the very basic steps of the KMeans algorithm described above but it classifies small random samples at a time to improve the time to convergence. After the randomization step, each image was passed to a method that considered the three values of a pixel (YCrCb) and grouped them into two clusters. Another parameter specified was the maximum number of iterations of the k-means algorithm to conduct before stopping it, which we set at 100. Interestingly, increasing this value did not dramatically change the preprocessing time, suggesting the YCrCb color space helped the algorithm reach convergence pretty quickly. To work around the hefty preprocessing time, we read the images in once, preprocessed them, and then wrote the "class" of each pixel into a massive text document that could be reshaped into input for the model in a separate Python file.

Model Validation

In order to compare different models and eventually select one that would generalize well, we needed to separate our larger data set into three portions for training, validation, and testing. After preprocessing our data, we used scikit-learn's train_test_split method which separates data randomly into two subsets of data in order to define these three smaller datasets. We first designated 60% of our larger data set to use for training, and then split the remaining 40% of data into a validation dataset and testing dataset respectively. Each model architecture we tested was trained on our training dataset and then tested on the validation dataset. Based on each model's performance on the validation dataset, we would experiment with different hyperparameters and different combinations of layers with the goal of improving this performance. Therefore the validation dataset is only indirectly affecting the training of the model, as the models are only ever trained on the training data but assessed on the validation data. After we had selected our model, we used the test dataset to see how different regularization methods would perform on it. As the model had neither been directly or indirectly influenced by this test dataset at any time during training and selection, its performance on this data would offer the best estimate on how the model would actually perform in production when attempting to classify images of which it had never processed.

CNN Architecture

Although there currently exists no set formula for building the ideal CNN for any given dataset, there are general guidelines to abide by that we can infer by examining the architectures of various state of the art CNN models. In general, most CNNs that aim to perform some sort of visual or image classification like CifarNet, AlexNet, and GoogLeNet consist of sets of alternating convolution and pooling layers, which are followed by a single or multiple fully connected layers[13]. Generally successive convolution layers have a greater or equal amount of filters than the layer before it, and successive fully connected layers have less units than the layer before it. For convolutional layers, generally the kernel_size hyperparameter that specifies the size of the convolutional window stays relatively small for the layers, perhaps somewhere in the (2,2) to (5,5) range for many models. The strides hyperparameter, which controls the amount that the window shifts along the image, also seems to be most frequently set to 1 or 2. The amount of filters though in the convolutional layer can differ vastly depending on the dataset and the amount of layers employed by the model, so due to our limited computing power and time that we had to train and validate all of our models we decided to experiment with many variations of this filter hyperparameter for different combinations of layers, while keeping the kernel_size

[13] H. Shin *et al*., "Deep Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning," in *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1285-1298, May 2016

constant at (3,3) and stride being set to 1. Similarly we chose to use the Maxpooling version of the pooling layer and kept its pool_size at (2,2) throughout all of our model testing as these smaller pool_size values seemed common among sophisticated CNNs. For the fully connected layers we choose to experiment with different amounts of units per layer, as well as using different amounts of fully connected layers. In hindsight, these decisions proved to be wise as even with only varying the filter and unit parameters some models would take quite long to train (around 2.5 hours) and we were still able to build quite effective models overall. Additionally we decided that all of our layers should use the ReLu activation function with the exception of our output layer which instead used the probabilistic softmax function since the output layer would actually have to classify each image into one of the 29 categories.[14]

Table 1

Filters	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5	Epoch 6	Epoch 7	Epoch 8	Epoch 9	Epoch 10
16	0.6296	0.7028	0.732	0.7372	0.7441	0.7431	0.7439	0.7434	0.7478	0.7422
32	0.6882	0.7327	0.757	0.7608	0.7653	0.7666	0.764	0.7657	0.7619	0.7662
64	0.6986	0.7492	0.7582	0.7642	0.7703	0.7687	0.7691	0.7686	0.768	0.7701
128	0.7285	0.7658	0.7714	0.7798	0.7749	0.7798	0.7762	0.7758	0.7765	0.7776

Table 2

Filters Per Layer	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5	Epoch 6	Epoch 7	Epoch 8	Epoch 9	Epoch 10
16, 32	0.731	0.7976	0.8284	0.8397	0.8418	0.8438	0.8507	0.8521	0.8447	0.8502
32, 64	0.7664	0.8254	0.8468	0.8542	0.8548	0.857	0.8599	0.8578	0.8582	0.8591
64, 128	0.8115	0.8628	0.8729	0.8805	0.8746	0.8812	0.8867	0.8834	0.8833	0.8837
128, 128	0.8121	0.8584	0.8641	0.8766	0.8719	0.877	0.8774	0.8808	0.8781	0.8751

Table 3

Filters Per Layer	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5	Epoch 6	Epoch 7	Epoch 8	Epoch 9	Epoch 10
16, 32, 64	0.754	0.8274	0.8519	0.8808	0.8949	0.8834	0.8971	0.8963	0.9036	0.9113
32, 64, 128	0.8139	0.8732	0.8998	0.9128	0.9159	0.9196	0.9236	0.9169	0.9225	0.9281
64, 128, 128	0.826	0.8892	0.9156	0.9168	0.9231	0.9309	0.9241	0.9321	0.9374	0.9301

Table 4

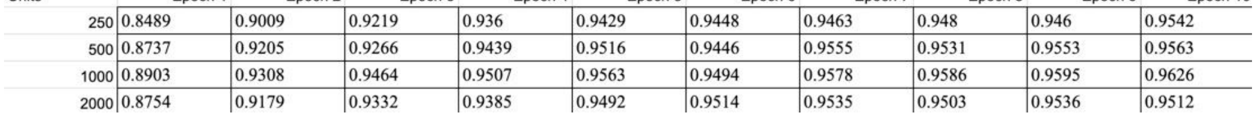
Filters	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5	Epoch 6	Epoch 7	Epoch 8	Epoch 9	Epoch 10
16, 32, 64, 128	0.7431	0.823	0.8639	0.8767	0.8922	0.9013	0.9018	0.9096	0.9131	0.9107

[14] "Application of convolutional neural networks for static hand gestures recognition under different invariant features," pp. 1-4

We first began by experimenting with both the number of filters per convolutional layer and the amount of alternating convolution and pooling layers in the model. We experimented with up to 4 sets of convolutional and pooling layers and up to 4 different combinations of filters per set. Each of the 4 tables shows the model's performance across 10 epochs for the filter variations we tested. As shown in the data, the general performance for each set of alternating convolutional and pooling layers improved as we increased the filter sizes across the layers. Performance also generally increased as we added an additional convolutional and pooling layer to the model. Although this seems to be the general trend, the increase in performance is much more dramatic between tables 1 and 2 compared to tables 3 and 4, suggesting that the increase in performance starts to be extremely minimal if existent at all as the model grows more complex.

Additionally, increasing the amount of filters greatly increases the time it takes for the model to train, as some of the more complex models like the 64, 128, 128 variant of table 3 took about 2.5 hours just to fit while the 16, 32, 64 variant took only 15 minutes to fit. Therefore, when selecting a model we took into consideration not only its performance on the validation dataset but also the time it took to fit, as we still had to experiment with dense layers and regularization and an extra 2% in performance did not seem to justify a 900% increase in the time to fit the model. The model with 4 sets of convolution and pooling layers actually performed worse than the 3 sets model variant, so we opted to use 3 sets of alternating convolution and pooling layers. As mentioned before, due to a minimal increase in performance coupled by a stark increase in fit time, we selected the more simple 16, 32, 64 filter variant for this model.

Table 5



Units	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5	Epoch 6	Epoch 7	Epoch 8	Epoch 9	Epoch 10
250	0.8489	0.9009	0.9219	0.936	0.9429	0.9448	0.9463	0.948	0.946	0.9542
500	0.8737	0.9205	0.9266	0.9439	0.9516	0.9446	0.9555	0.9531	0.9553	0.9563
1000	0.8903	0.9308	0.9464	0.9507	0.9563	0.9494	0.9578	0.9586	0.9595	0.9626
2000	0.8754	0.9179	0.9332	0.9385	0.9492	0.9514	0.9535	0.9503	0.9536	0.9512

Table 6

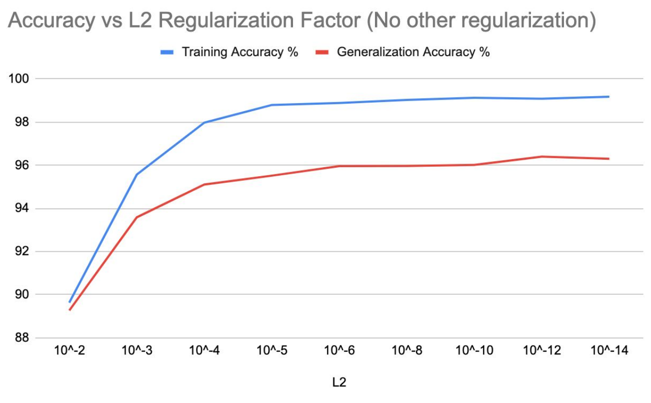
Units Per Layer	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5	Epoch 6	Epoch 7	Epoch 8	Epoch 9	Epoch 10
1000, 1000	0.8767	0.9325	0.9501	0.9545	0.9552	0.9626	0.9629	0.9639	0.9614	0.9644
1000, 500	0.8753	0.9201	0.936	0.9466	0.9488	0.9528	0.9572	0.9609	0.954	0.9606

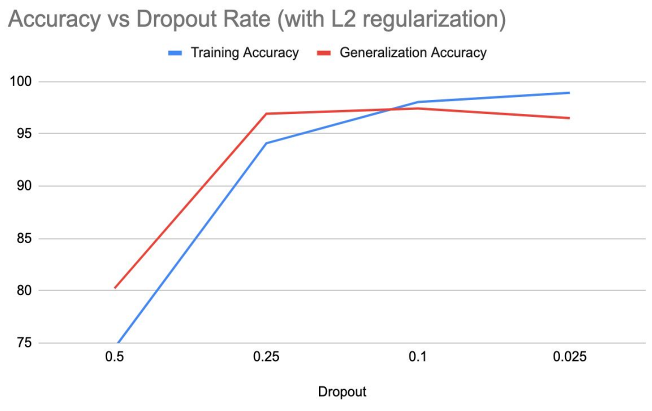
Next we began to experiment with adding fully connected dense layers after the convolutional and pooling layers and before the output layer. As shown in the data, the addition of a dense layer greatly increased the model's performance but the relationship between additional units in a layer and the model's performance is much less transparent. Adding a second dense layer did not seem to greatly affect the model's performance either, so we selected the simpler single dense layer to avoid increasing the model's complexity further. The 1000 unit variant did perform slightly better than its 500 and 250 unit counterparts, so we selected to use a single dense layer with 1000 units. Therefore, before adding any regularization our chosen model consisted of 3 sets of alternating convolutional and pooling layers with 16, 32, and 64 filters respectively, followed by a single dense layer consisting of 1000 units and our output layer.

Regularization

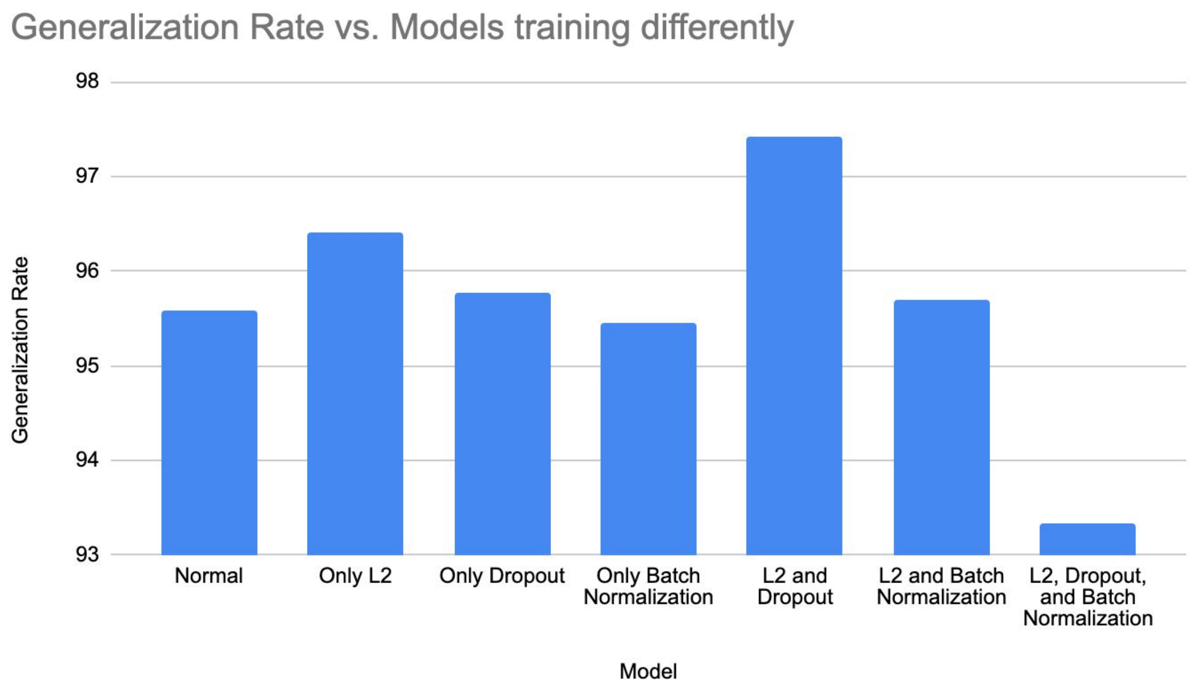
Regularization and normalization are methods by which to improve the performance of a neural network by simply rescaling the features or manipulating the loss function in order to reduce overfitting. For our neural net, we tested 3 different methods of regularization and normalization: Batch Normalization, Dropout, and L2 regularization. These methods were selected because they are known for their improved accuracy of image classification neural networks. Batch normalization is a method of normalization that normalizes the inputs to the succeeding layer. Normalization is the process of rescaling features to values in between [0,1]. We anticipated this would be ineffective because the only input to the first level was already normalized—the binary output of the k-means preprocessing. Dropout is a type of regularization method that approximates training a large number of neural networks with different architectures in parallel. This means that during training, some number of a layer’s outputs are randomly ignored. This has the effect of giving the model a different outlook of every layer's architecture. L2 regularization, also known as ridge regression, has the goal penalizing complex models in order to decrease overfitting. This method accomplishes its goals by adding a term to the loss function to penalize for large weights. L2 adds the square magnitude of the coefficient as the penalty term of the loss function.

Many of the regularization and normalization methods did in fact show increased generalization performance, but others had the inverse effect. Inserting batch normalization after every convolutional layer was by far the least effective method when applied both individually and in combination with L2 and Dropout. Both of these scenarios produced models with worse generalization performance than the original model with no methods applied. Inserting Dropout layers after every MaxPooling, Convolutional, and Dense layer (excluding the final dense layer), improved the generalization performance to 95.77%. This performance was achieved when using a Dropout factor of .1 for each dropout layer. The optimal dropout factor was determined while empirically testing values between .5 and .025. Applying L2 regularization to dense layers, excluding the final layer, improved generalization performance to 96.4%. This level of performance was achieved with an L2 regularization factor of 10^-12. The optimal regularization factor was determined by testing values between 10^-2 and 10^-14.





Some regularization and normalization methods worked better on their own, while others worked better when combined. It is clear from the results that batch normalization was an ineffective method of improving our model's performance. When applied to the model by itself, it actually brought down the models generalization rate by 0.12%. Even more interesting, when batch normalization was applied with both L2 regularization and dropout layers, it dropped the models generalization rate by 2.25%. L2 regularization was the method that improved the model the most by itself, with dropout coming in second place. On their own, L2 regularization and Dropout improved the model's generalization rate by 0.82% and 0.19% respectively. Unsurprisingly, when these two methods were applied together they improved the model's generalization rate by 1.85%. By using both L2 regularization in tandem with dropout layers, we achieved a final generalization rate of 97.43% for our model.



Evaluation Approaches

We decided to use generalization accuracy, which is the number of correct predictions divided by the total number of predictions, as the primary metric for evaluation based on the two ASL datasets used in the training process. There was not a heavy imbalance in letter frequency in the two datasets that would skew the model enough to make generalization accuracy an unreliable metric. In order to supplement the generalization accuracy, we also used top 3 categorical accuracy to evaluate the model. Top 3 categorical accuracy considers whether or not the model's first three predictions were correct which is a considerable increase over the baseline accuracy's single prediction. This metric is often used in language processing as spellcheck can be used to make up for a model's shortcomings. We considered using a metric that weighted the letter predictions based on their frequency in the Oxford dictionary but decided against as the Oxford dictionary is not representative of the lexical changes within the ASL community.

Results and Discussion

As a recap of our results, we achieved a final model with a generalization accuracy of 97.43%. We achieved this level of accuracy by first testing various model architectures and choosing the one with the best generalization accuracy of 95.58%. Furthermore, we tested various regularization and normalization methods on this architecture for an additional increase of 1.85%. Using the top 3 categorical accuracy, this value further increased to 99.18%. A confusion matrix is shown below with the predicted results of our model on 32898 examples. As can be seen, the model generally predicts the correct class (the diagonal line), but occasionally predicts the wrong one. The most common mistake was the model misinterpreting the asl sign for ‘r’ as ‘u.’ This mistake actually makes a lot of sense, because the asl signs for ‘r’ and ‘u’ are very similar.

All of these optimizations were made to improve the model’s performance but they ended up only achieving a very good in-sample performance. In order to stress test the model’s out-of-sample capabilities, we used 4 of our group’s hands with 5 different backgrounds to see what the model struggled with. On a small out-of-sample pool of 50 fingerspelling hand signs, the model only achieved a 24% generalization accuracy which only increased to a 52% accuracy rate when considering the top 3 predicted letters. This is due to a variety of different factors. First, the first dataset we selected was composed of only right handed fingerspelling images while 10 of the 50 images of our own hands were left-handed. Among left-handed test images, only 1 out of all 10 had a correct prediction within the top 3. The model particularly struggled involving any of the images using a window with small glass panes at nighttime as it likely heavily interfered with the model’s YCrCb color space. As a collective, we are also not well-versed in ASL and could have also formed letters incorrectly which would have further decreased the amount of accuracy that could have been achieved.

These errors were mainly caused by our group’s inexperience with the use of ASL. We only came to the understanding that our initial dataset was heavily flawed very late in the model’s development. That clean dataset made up a large percentage of the training, testing, and validation portions which caused the resulting model to be grossly overfitted to the available in-sample images.

Conclusion

Through all these methods, we were able to complete our original goals of an ASL classifier to an extent. In reality, when first mulling over our project we were unaware of some of the domain-specific aspects of ASL as none of us know the language. By the proposal, we knew that computer vision classification of ASL was a worthy goal, and we knew about approaches that other individuals had taken in accomplishing it. But with the resources available we rapidly found ourselves restricted to a focus on classifying fingerspelling-- largely in part due to the computational intensity of convolutional neural networks, which we had decided on using early in the process. Like many groups, we would imagine, the pandemic and BC's social distancing policies complicated the process. Losing in-person access to the GPU computers in the Machine Learning Lab, for example, led to working through a lot of our collaboration by testing out pre-processing and sharing code on Google Collab notebooks. Fingerspelling was particularly appealing as a result, as it is still an extremely helpful part of the language to classify while also being feasible remotely and with static images. Movement, or signation, is an extremely important part of the language, but only two fingerspelling gestures ("j" and "z") require it, making fingerspelling optimal for our model.

In terms of our actual implementation, some of the tools that turned out to be most important in accomplishing our task was data augmentation, pre-processing, and regularization. We found that improving the generalization of our model with our pre-processing, although a computationally costly venture, was worth it as our model could be saved and reused later. However, we found that time spent fitting did not similarly always translate to better model performance. We discovered that adding a few convolutional filters or layers in general could increase the time the GPU computer spent fitting by several hours, but there were clearly diminishing returns in model performance as the amount of time spent training the model increased, sometimes with no improvement in performance at all. We also observed that regularization techniques did not improve accuracy equally for all of our datasets, and that in tandem these techniques might sometimes take away from the model's performance, like our model when Batch Normalization, L2 regularization, and dropout layers were applied in unison. On the other hand, the right combination had even larger positive impacts to our generalization.

Thus, possibly our greatest takeaway from this project was that domain-specific knowledge, and understanding of the type of data we have access to is extremely important in training an optimal model, even with algorithms as powerful as a Convolutional Neural Network. Given our experience on this project, there are a few things we might try differently. While much of our early research was rightfully focused on existing applications of ASL classifiers and computer vision via CNN, acquiring more domain-specific information would have set us up with the foundational knowledge we needed for implementation later on in the process. While our generalization performance is great, in hindsight we are curious about how a video input classification model could have performed had we spent the time used to optimize our model on implementing video input. We believe this would be the natural next step in extending the project, and to go a step further it would be possible to classify live video classification. These are, however, very different ball games, with their implementations being considerably more difficult.

Taking a look at the current field of ASL research, SignAll is a company that sells live video feed translation utilizing special gloves with nodes tracked by their computer vision model in order to translate motion. The teams behind Word-Level American Sign Language (WLASL) and DeepASL are both tackling video processing in different ways. DeepASL utilizes Leap

Motion controllers to evaluate a pool of 56 words with more than a 91.8% accuracy. The team behind WLASL created the largest public ASL dataset based on vocabulary size with subsets of the dataset using 100, 300, 1,000, and 2,000 of the most common words in ASL. Owing to the sheer number of words included, the accuracy for their models is much lower with their top word prediction for the 100 word vocabulary pool sitting at 65.89% and only 32.48% for the 2,000 word vocabulary pool. With the increased need for remote learning during the pandemic, a team has created the Signing Avatars & Immersive Learning (SAIL) Virtual Reality program to build educational content for deaf children to consume. The field of hand articulation in VR has also seen a large improvement with the accurate MEgATrack system for real-time hand-tracking. Innovations in ASL based machine learning and tangentially related fields have been being published at a rapid rate and it would be interesting to try and tackle some of those problems. We hope to see further breakthroughs in the field of ASL translation soon.

References

American Sign Language. (2020, November 05). Retrieved December 03, 2020, from https://www.nidcd.nih.gov/health/american-sign-language

C. J. L. Flores, A. E. G. Cutipa and R. L. Enciso, "Application of convolutional neural networks for static hand gestures recognition under different invariant features," 2017 IEEE XXIV International Conference on Electronics, Electrical Engineering and Computing (INTERCON), Cusco, 2017, pp. 1-4, doi: 10.1109/INTERCON.2017.8079727.

Fang, Biyi & Co, Jillian & Zhang, Mi. (2017). DeepASL: Enabling Ubiquitous and Non-Intrusive Word and Sentence-Level Sign Language Translation. 10.1145/3131672.3131693.

Han, Shangchen & Liu, Beibei & Cabezas, Randi & Twigg, Christopher & Zhang, Peizhao & Petkau, Jeff & Yu, Tsz-Ho & Tai, Chun-Jung & Akbay, Muzaffer & Wang, Zheng & Nitzan, Asaf & Dong, Gang & Ye, Yuting & Tao, Lingling & Wan, Chengde & Wang, Robert. (2020). MEgATrack: monochrome egocentric articulated hand-tracking for virtual reality. ACM Transactions on Graphics. 39. 10.1145/3386569.3392452.

H Shin et al., “Deep Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning," in IEEE Transactions on Medical Imaging, vol. 35, no. 5, pp. 1285-1298, May 2016, doi: 10.1109/TMI.2016.2528162.

Li, Dongxu & Rodríguez, Cristian & Yu, Xin & Li, Hongdong. (2019). Word-level Deep Sign Language Recognition from Video: A New Large-scale Dataset and Methods Comparison.

SignAll. (n.d.). Retrieved December 04, 2020, from https://www.signall.us/

Signing Savvy, L. (n.d.). Fingerspelling. Retrieved December 03, 2020, from https://www.signingsavvy.com/fingerspelling

Qiu-yu, Zhang & Lu, Jun-chi & Zhang, Mo-yi & Duan, Hong-xiang & Lv, Lu. (2015). Hand Gesture Segmentation Method Based on YCbCr Color Space and K-Means Clustering. International Journal of Signal Processing, Image Processing and Pattern Recognition. 8. 105-116. 10.14257/ijsip.2015.8.5.11.

Quandt, Lorna & Lamberton, Jason & Willis, Athena & Wang, Jianye & Weeks, Kaitlyn & Kubicek, Emily & Malzkuhn, Melissa. (2020). Teaching ASL Signs using Signing Avatars and Immersive Learning in Virtual Reality. 10.1145/3373625.3418042.