Image Classification of the American Sign Language Alphabet with Convolutional Neural Networks

Jacob Hohn | A11964556
Silvina Rodriguez | A14072699

Abstract

This report attempts to shed a light on the importance of inclusion and communication as technology advances without addressing needs of all communities. This is achieved through the modifying and improving the LeNet convolutional neural network architecture in order to perform image classification on an American Sign Language MNIST data set. This data set contains image data for 34,627 ASL alphabet signs. Through the addition of two dropout layers, image augmentation, and optimization of hyper parameters, we were able to improve testing accuracy from 86.89% to 99.07% and reduce incorrectly predicted samples from 940/7172 to only 67/7172. While this model is limited to classifying static images of the ASL alphabet signs, it represents a first step in bridging communication gaps between the hearing and Deaf community.

1 Introduction and Motivation

We created our project because we realize that with all the technology that we have at the tip of our fingers—there is still a limited area that recognizes signed languages. Even something as simple as a program that recognizes the ASL alphabet it could be useful for the Deaf community to communicate with others. As explained in the PBS special 'Through Deaf Eyes,' 'technological "advancements" were often assessed differently by hearing and Deaf people.'[11] Our technological advancements have not always been deaf user friendly. By making this algorithm it can expose people to the endless possibilities to assist the Deaf community and promote inclusion in the world. It is very common to use finger spelling when signing with deaf people to finger spells words that you don't know how to sign yet. One application of this project could be to create a program that can help hearing people learn ASL signs and communicate through finger spelling. In the article "Unemployment in the Deaf Community: Barriers, Recommendations and Benefits of Hiring Deaf Employees," it explains that even with the increase of Deaf people getting their degrees there has been a continuous decline of deaf individuals in the work force. From the year 1970 to 2014, the percentage of the Deaf community in the workforce dropped from 80% to about 48%. [6] It seems that employers believe that it is too expensive or inconvenient to get tools for the Deaf to be able to communicate and understand what their job needs from them. Our project is creative because our technology can be used to inform the hearing community of Deaf culture and struggles.

2 Problem-Solving Approach

2.1 Thought Process

With a topic in mind we set out to find relevant data sets related to signed languages. We found four data sets total, one that compared British and American Sign language with values collected from a leap motion controller, an Australian sign language data set that measured different signs similarly with a glove that has trackers on it, a data set with colored photos of the ASL alphabet collected through a Kinect device that includes depth data as well, and finally an ASL MNIST dataset similar to the classic handwritten letters MNIST dataset with images pre-packed into testing and training csv files as a label and 784 individual RGB grayscale values. [3, 14, 17, 7] We decided to continue the project with the ASL MNIST data set. While it would have been interesting to compare two sign languages together, we did not know how to interpret the data and it was collected with a system that we are not familiar with. It seemed logical to choose between the two data sets that have images prepared to analyze. The ASL MNIST data set was more attractive because it was simpler data to work with, already split up into testing and training data, and there was a framework to jump off of in regards to image classification through the large amount of documentation on the MNIST data set.

After having a data set in mind, the next logical step was to decide what to do with the data and how to achieve that goal. It was important to frame the scope and recognize some shortcomings of the data of the project before diving too deep into implementing any models. While the ASL MNIST dataset contains static images of alphabet signs, sign language as a whole is very dynamic and complex where even facial expressions contribute to the grammar of constructed sentences. This dataset is very valuable to investigate, but recognizing some missing pieces is important to understand the maximum impact our work could have. So, taking some inspiration from the classic MNIST data set and an in class demo of deep learning with neural networks, we chose to use the data to perform an image classification task with convolutional neural networks.[10, 4]. There was a brief consideration of using the machine learning algorithm K-nearest neighbors. While this worked on simpler image

classification tasks such as the handwritten letter data set, the images available in the ASL MNIST dataset were not quite as clean - they are grey on grey, with the differences in shading and shape accounting for subtle differences between letters. There are too many alphabet letter signs that look close to each other, 'M' 'N' and 'T', 'A' 'S' and 'E', 'R' 'U' 'V' and 'W' (comparison shown in Fig. 1. This coupled with location/rotation invariance and high dimensional photos would have been a nightmare to deal with. Convolutional neural networks offered an alternate solution that, through some architecture manipulation, could learn to predict and recognize the subtle differences between letters. We expected the convolutional neural network to run into the most problems when it came to classifying similar looking letters initially, but there was much more fine-tuning that could be done to account for these issues.

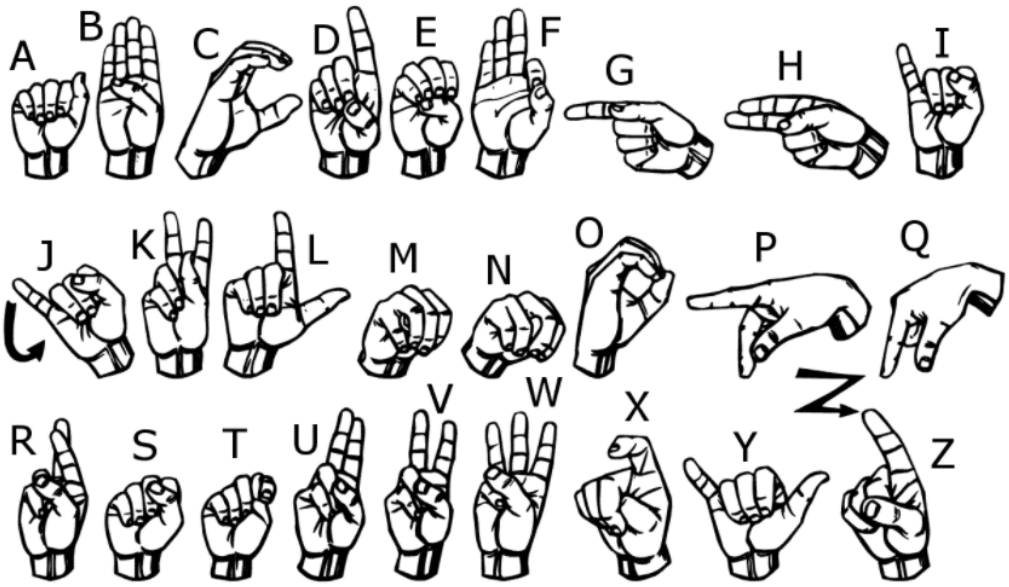

Figure 1: American Sign Language Alphabet Signs

2.2 Theory

Before trying to improve any existing neural network model architecture, we needed to first understand how and why they operated - why were they good for image processing specifically? As shown in Fig. 2 a neural network will take in an input, send it through some 'hidden' layers and predict an output.[8] In the case of an image classification task, a neural network model will train on a set of labeled input images to correctly match the output label to the input label. The model will improve over time through the use of activation and loss functions - activation functions are used to propagate inputs through a neural network and loss functions are used to make adjustments based on how far off a prediction is. When the model is then applied to a new set of data, assuming the model is robust enough, it will be able to correctly identify the images. There are several types of hidden layers that can be used to create model architecture. Some popular architectures are LeNet, shown in Fig. 3, the first state of the art architecture created in 1998 and vgg16, the most recent complex architecture. [2] Convolutional neural networks, like LeNet and vgg16, are useful for image classification tasks because the specific layers implemented reduce the dimensionality of the input data in order to identify and use only the key components.

The first layer used is convolution. The convolution layer, shown in Fig. 4 is used to extract features from an image. The convolutional layer takes five inputs: filters, kernel size, strides, padding,

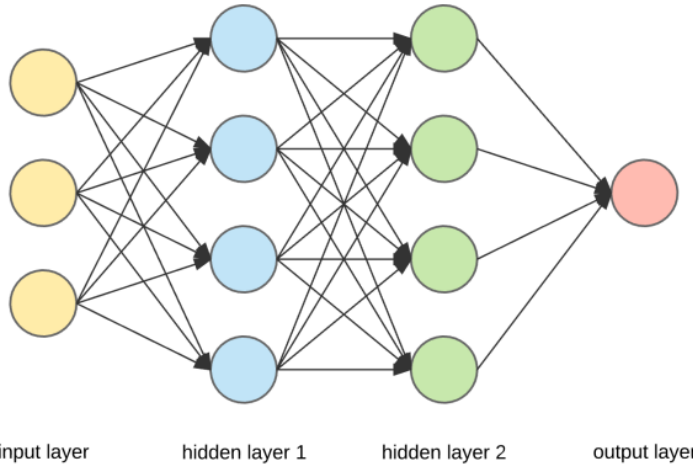

Figure 2: General Setup of a Neural Network. Each input is associated with a different weight that is updated throughout training in order to better predict the class of a sample in the output

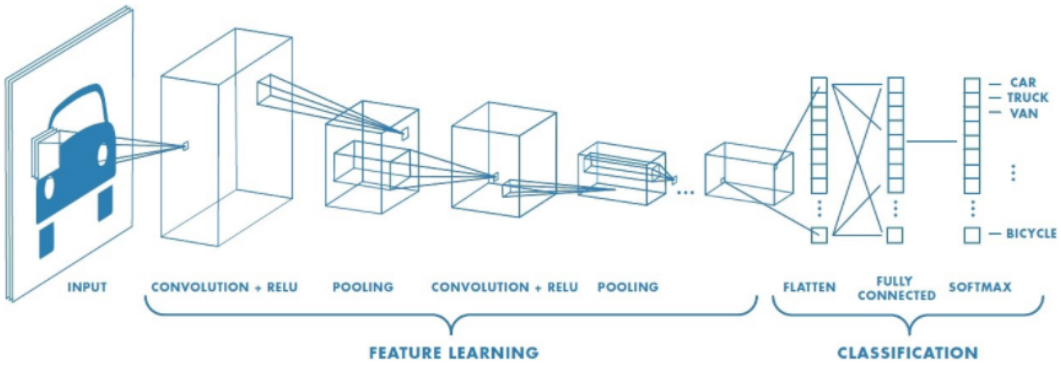

Figure 3: Convolutional Neural Network Base Architecture

and activation. Filters is just a number to determine the number of filters used in this convolution, essentially the number of features to extract. The kernel size argument determines the size of the kernel (or matrix) in the axial directions and is represented as a tuple (x, y). The stride argument determines how much the kernel skips over when moving from input to input. Padding provides padding to reproduce a certain size output so it can be the same or different depending on what you want. The activation argument determines which function is going to be used, either “ReLU”, “tanh”, “sigmoid”, “Softmax”, etc. These introduce a level of nonlinearity into the model which distinguish them greatly from standalone feed forward neural networks. We used the ReLU function for our project, shown in Fig. 5 alongside two other popular activation functions. This function applies the rectified linear unit activation function; with no input this function returns the standard ReLU activation: max(x, 0), the element-wise maximum of 0 and the input tensor.[8, 22, 12, 4]

The pooling layer reduces the number of parameters when the image is too large. It also helps reduce the effects of location and rotation invariance. The two main types of pooling are average and max pooling; average uses the average values as output for a certain set of values, while max pooling assumes the maximum value in the subset should be the output. This project mainly uses average pooling. The Average Pooling layer, shown in Fig. 6, takes in 3 inputs: the pool size, the strides, and

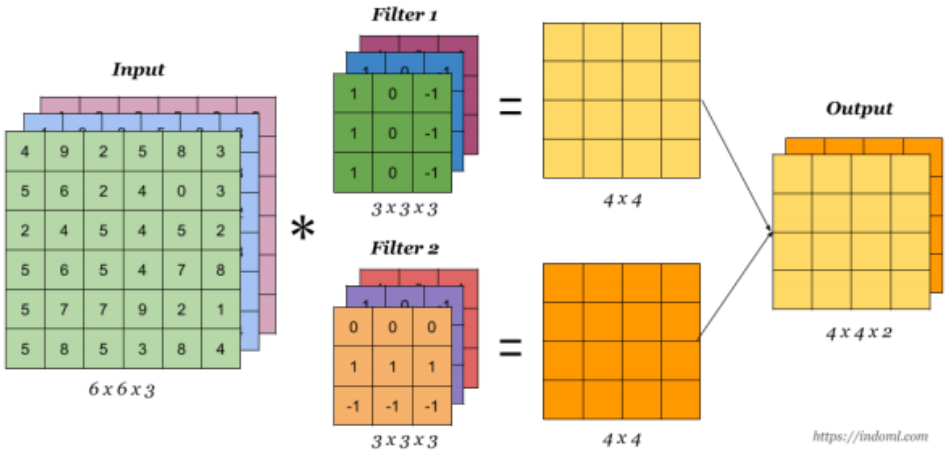

Figure 4: Example of Convolution applying an X x X filter to reduce the dimensionality of an input image[8]

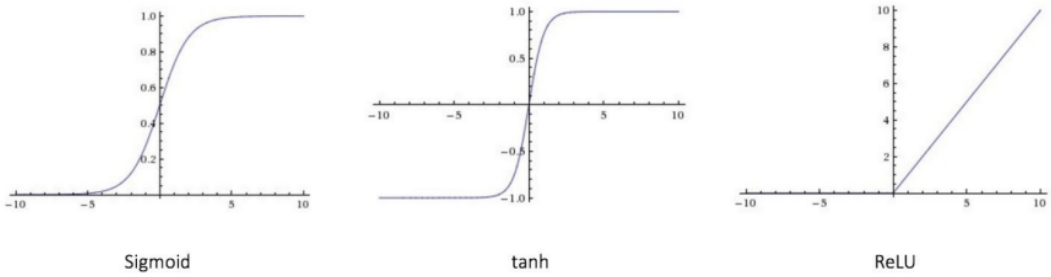

Figure 5: Activation functions which introduce non linearity into neural network models[8]

padding. Pool size is an argument that determines the dimensionality of the pooling kernel in terms of the x and y axis. Strides determines the amount of skipping done on the input layer, since we are pooling and don't want overlap, Striding is usually set to poolsize. The padding argument determines how much to pad before pooling, usually best to leave the argument as "valid". [1, 4, 12]

The Flatten function flattens the dimension of the input without affecting the batch size; in our case it converts a 28 x 28 image of pixel values into one 784 flat array. The Dense function takes two arguments, units and activation, units determines how many hidden units to use in the fully connected layer being created. The activation argument does the same as above, applies different functions such as "ReLU" or "Softmax". The Dense function compiles our layers into one fully connected layer. Once finished with those steps we used the "Softmax" activation function to create a probability distribution from the final layer result. "Softmax" itself converts an input vector into probability vectors that add up to 1.[20, 4]

Neural network models are implemented in Tensorflow as functions detailing the expected input array and series of layers to go through. After setting up an individual model with the necessary layers, an optimizer, in this case Adam is set up. Adam, uses a stochastic gradient descent (SGD)

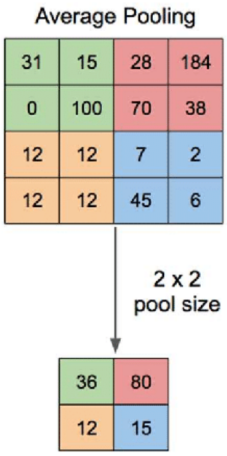

Figure 6: Example of Average Pooling. Another instance of reducing dimensionality through taking the average values of pixels to create a new image[1]

method for optimization. In this project, SGD, shown in Fig. 7, is used together with a cross entropy loss function to calculate the loss and then back propagate through the model and update weights accordingly. Categorical cross entropy is used because there are a total of 24 possible classes that a sample can be labeled/predicted as. While it is updating, parameters will move to certain local and global minima and saddle points; this is one of the reasons that training a model multiple times will not yield the same exact results. Adam is unique because it computes efficiently and has little memory requirements. It is also invariant to diagonal rescaling of gradients and is well suited for problems that are large in terms of data/parameters. Adam works in tandem with the selected loss function whi [23, 18, 8]

The last relevant layer in this project is the dropout layer. This layer randomly sets a percentage of data to 0 and up scales the other data by $1/(1 - rate)$ so that the sum is not changed within the inputs. Incorporation of dropout layers is particularly important in this project as it will help reduce overfitting from the base architecture we start with. Tuning the percentage of how much data is dropped will be crucial to maximizing model efficiency as well. [19, 13]

Hyperparameters determine the structure of our network and how it is trained. Since it determines the training, it must be defined before the training begins. The hyperparameters closely examined in this project are: kernel size, batch size, learning rate, epochs, validation split, and dropout rates. Kernels act as feature detectors in convolution. Batch size is the amount of sub samples to be processed by the network. A higher batch size can decrease training time, but may cause the model to not generalize as well. Good batch sizes correlate with bit sizes: 32, 64, etc. Learning rate, shown in Fig. 8, is associated with SGD optimization. A high learning rate speeds up optimization by reaching local minima quicker, but may overshoot and cause the model to not converge. A small learning rate is the opposite, slower speeds but the learning is guaranteed to converge. The number of epochs is how many times the model will propagate through the network and update weights before completing. Epochs should be increased until validation accuracy is negatively affected (an indication of overfitting); certain measures like the EarlyStopping method can also be used to pre-maturely stop training if error is not changing. Validation split is the percentage of training data reserved for testing during the training of the model. It is recommended to have a smaller validation set when a model has fewer hyperparameters because it is easier to validate, and a higher validation set with more parameters. The last hyperparameter optimized in this project is dropout rate, the percentage of data set to 0 at

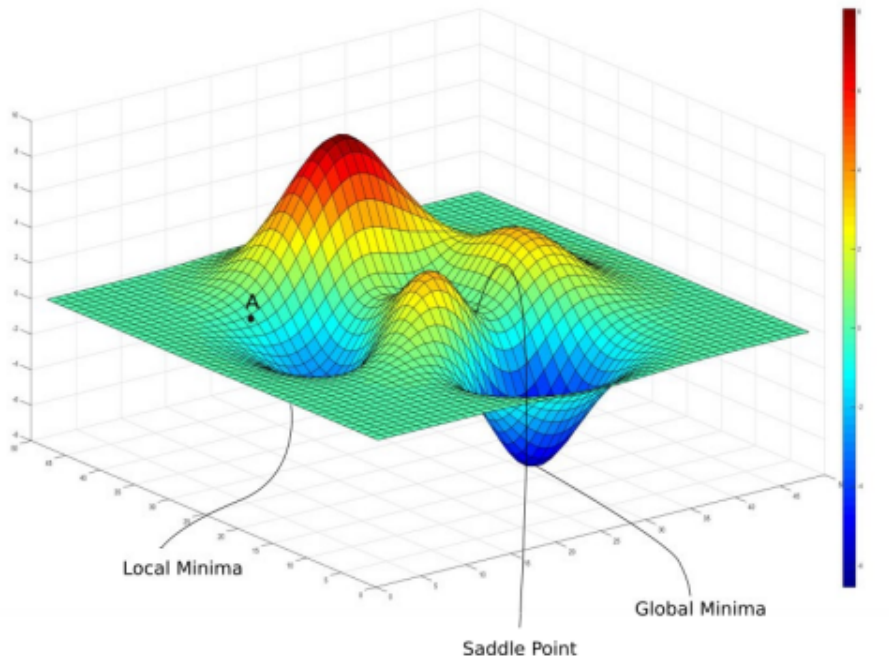

Figure 7: Stochastic Gradient Descent[8]

each dropout layer. If the value is too low, the effect may not be noticeable and a value too high will remove too much data and cause under-learning. A balance is necessary to reduce overfitting and still maintain high accuracy. [8, 13, 16]

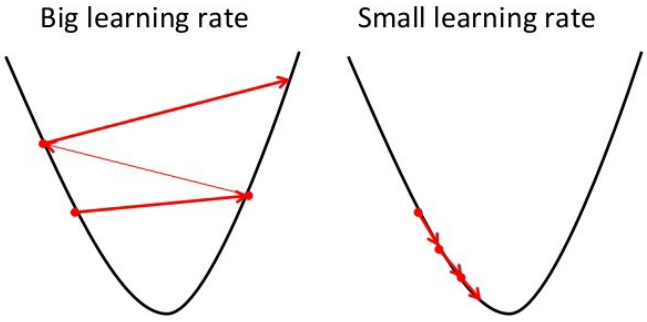

Figure 8: Effects of high and low learning rate[13]

3 Solution

3.1 Preprocessing

We were lucky enough to obtain a data set that was mostly pre-processed. The ASL MNIST data set used in this report resembles the classic handwritten letter MNIST in that there is a number identifying the image as one of 24 alphabet signs, excluding 'J' and 'Z' because they require movement, and 784 values that correspond to greyscale RGB values.[17] When the 784 pixels are reshaped into a 28 x 28 grid, the image can be displayed using matplotlib, as shown in Fig. 9. The data set was pre-split into training and testing data with an approximate 80:20 ratio. The training data has 27455 images and the testing data has 7172 images, giving a total of 34627 images. The only array pre-processing necessary was to reshape each row into a (28,28,1) numpy array, effectively isolating each RGB value for the model to input through each layer. Tensorflow handles the conversion of each numpy array and all necessary mathematical operations as it propagates values forwards and backwards through the neural network to update weights to predict future samples.

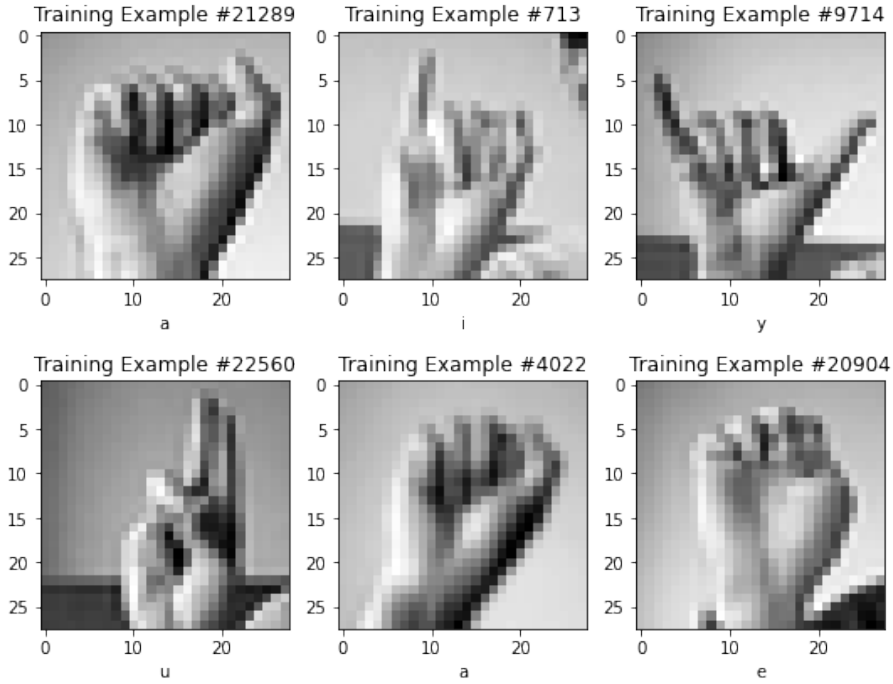

Figure 9: Examples of six signs in the training data set with their corresponding label

Before performing any tests, or applying any model fitting to our data set, we first created simple visualizations to determine if classes were evenly distributed in the data set. Fig. 10 shows that all 24 classes are evenly distributed overall with around 1400 samples per letter. It was difficult to visualize whether all the samples are high quality. For the most part, images seem to be centered and taken against a flat background, but there are several pictures where it appears that the person holding the sign is in the picture, or there is additional harsh contrast between the sign and the wall behind the hand. This sort of variance along with location and rotation invariance may cause neural network models we create to not perform as well by updating weights which reflect these unimportant features. One additional pre-processing step that may help with such invariances in our images and is built into tensorflow is an ImageDataGenerator method which randomly scales, shifts rotates and shears each

image, by manipulating the individual RGB values. This allows the model to better generalize to new data since it will recognize images that have certain location and rotation invariances to still belong to a certain class.

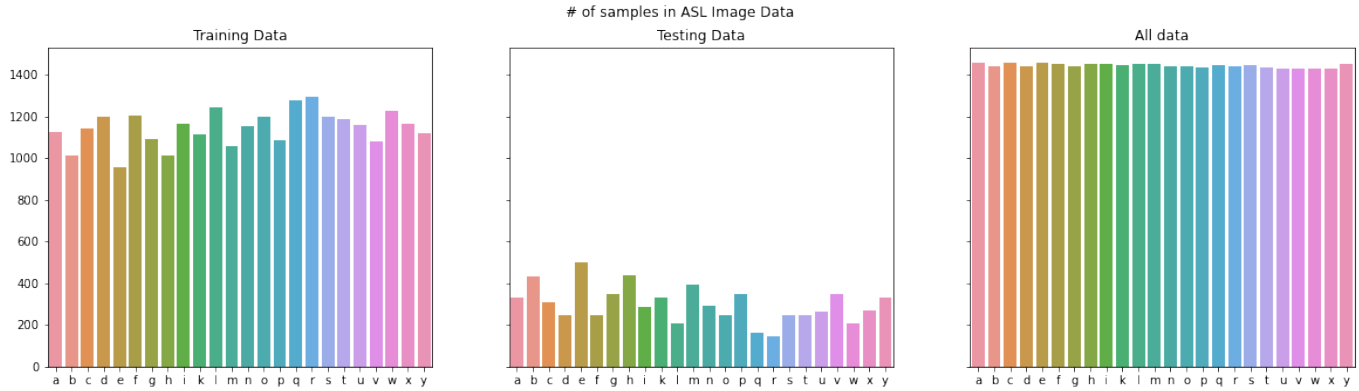

Figure 10: Distribution of 24 American Sign Language Alphabet Signs.
'J' and 'Z' not present because of motion required to perform the sign

One final pre-processing step that became important after several complete run throughs was attempting to limit randomness in an effort to get fully reproducible results, whether that be the same final architecture steps, optimal hyper parameter values or the same final test accuracy. Randomness is a part of almost every part of the process: randomly initialized weights (leading to different local minima during SGD), randomly shuffled data, random dropout data (certain data may be important one run but not the next), and randomness in optimization through SGD to name a few. [5] We attempted to combat this randomness through several methods. First, we tried setting a seed for Tensorflow and Numpy at the start of the notebook, assuming that it would apply to all processes that have randomness factors. A seed is still 'random', but can retrieve the same 'random' each time a process is done. When this did not work on its own, we attempted to shuffle all data before it was input into a model with another specific seed. Normally, not shuffling data is not recommended in case a dataset is not well distributed or too ordered, but previous visualizations showed our data was both evenly distributed and had classes randomized already. Nevertheless, this still produced different results. Next, we tried to add on saving the randomly initialized weights of a model and re-inputting them several times. Finally, we added a specific seed value for the Dropout layer so the same 'random' data would be ignored each time. Even with all of these together, results were still not reproducible; there was some source of randomness we could not track down. This could have also been a consequence of not running for enough epochs, but runtime was another issue we faced, so we decided to compromise and choose one feature to keep constant. In our final models, we believed controlling the dropout seed was the most important as it was a feature being added into existing architecture as a comparison point.

3.2 Workflow

In order to achieve positive results and improve upon existing neural network architecture, we attempted to use a process flow that was as logical and simple as possible. Fig. 16 shows the exact workflow used in the stages to obtain an optimized model with test accuracy of 99.07% and only 67/7172 incorrectly classified signs. This represents a 12% increase in test accuracy from the original LeNet architecture which predicted 940/7172 signs incorrectly.[9] All results including test and validation

loss and accuracy are summarized in Fig. 11. The workflow was split up into three main phases: determining the best combination of additional/removed neural network architecture layers, testing image augmentation, and optimizing the values of a certain set of hyper parameters. In all tests of the first phase, certain hyperparameters were kept constant: a 5x5 kernel size for convolution, batch size of 50, 25% validation split, 0.001 learning rate for Adam optimization, and 20 epochs to fit the model. Before testing any additional layers, we first determined which architecture should be used as the 'default' architecture, Multilayer Perceptron (MLP), a classic feed forward neural network, LeNet architecture, the first state of the art convolutional neural network architecture, or vgg16 architecture, one of the most up to date convolutional neural network architecture. The architecture for the MLP and LeNet architecture were both taken from a CNN demo by a former ECE 196 TA, and the vgg architecture was taken from an article that explained how to "Create vgg from Scratch in Tensorflow" [4, 15]. The architecture for each are shown in Fig. 17, Fig. 18 and Fig. 19 respectively. In all tests, LeNet had the highest testing accuracy. While vgg16 is the most advanced, it is primed to handle data with the shape (224,224,3), which is much higher quality, less pixelated, and has three color channels instead of one. As seen in Fig. 19 there are more than 16 million trainable parameters compared to LeNet's 45,701 trainable parameters, so it is possible that the ASL MNIST data set is just not complex enough.

	test	training loss	training acc	testing loss	testing acc
0	Multilayer Perceptron	0.001532	1.000000	0.834670	0.827245
1	LeNet original	0.000058	1.000000	0.807563	0.868935
2	vgg16 reshaped	3.177472	0.044728	3.199315	0.034300
3	1a: 10% DO	0.018993	0.994901	0.656520	0.877440
4	1b: 20% DO	0.018041	0.994706	0.541907	0.865588
5	2a: 10% DO	0.020911	0.993055	0.781442	0.855689
6	2b: 20% DO	0.035355	0.988733	0.569484	0.875627
7	3: 2 DO layers	0.025800	0.991695	0.228107	0.939626
8	4a: remove 120 dense	0.000598	1.000000	0.860999	0.848578
9	4b: remove 84 dense	0.000401	1.000000	0.743542	0.879532
10	4c: remove 84 dense, add 2 DO layers	0.025813	0.991841	0.371307	0.917596
11	5a: Add Conv 32 out	0.000065	1.000000	0.842760	0.877858
12	5b: Add Conv 32 out and 2 DO layers	0.056057	0.981157	0.374343	0.900586
13	6a: Add Image Aug	0.084487	0.971226	0.078455	0.976576
14	6b: Add Image Aug and 2 DO layers	0.183354	0.936369	0.059693	0.979922
15	7a: 10x LR	0.000008	1.000000	1.048801	0.857083
16	7b: 3x3 kernel	0.000269	1.000000	0.594173	0.887758
17	7b: 3x3 kernel and 2 DO layers	0.031269	0.989656	0.382568	0.906302
18	Optimized Model without Image Augmentation	0.023535	0.992923	0.186220	0.951757
19	Optimized Model with Image Augmentation	0.168319	0.943325	0.032594	0.990658

Figure 11: Test Results from the best run: LeNet Architecture with 2 additional 30% Dropout layers, image augmentation, and optimized hyper parameters

Although LeNet had the highest testing accuracy of the three default architectures, there were obvious issues of overfitting; results showed 100% training accuracy and 86.9% testing accuracy. The tests that followed aimed to reduce this overfitting and increase overall testing accuracy while maintaining a high training accuracy. Dropout layers specifically aim to reduce over-fitting in models as they ignore a certain percentage of the data. Tests 1-5 consisted of adding/removing different layers and combinations of layers, keeping the combinations that increased testing accuracy the most. It was not feasible to test every single combination of layer, so we relied on a linear approach. Essentially, we would test a few variations of a change to the LeNet architecture on its own then add the best performing change to the best combination and compare the newest combination to the best combination so far. If the new changes did not increase the overall test accuracy then it was discarded. Examining tests 1, 2, and 3 shows an example of this. Two variations of adding a dropout layer were added at different locations in the LeNet architecture. The best performing dropout layer at each location was then combined and all four combinations were compared to see what performed best so far: default, 1 dropout after first pooling, 1 dropout after second pooling, or both dropout layers together. This was actually one of the largest sources of variation in our testing because the best layer would change each run, even after adding a constant dropout layer seed.

The tests continued with removing one of the Dense layers, representing a traditional feed forward neural network, and adding an additional convolution layer, increasing feature extraction. While both of these increased testing accuracy on their own, they did not perform better in conjunction with dropout layers than dropout layers performed alone, in any run. The next test investigated how image augmentation can increase testing accuracy and better generalize to new data. Image augmentation was handled with a tensorflow pre-processing method which randomly rotated data by $\pm$ 5 degrees, and zoomed, horizontally/vertically shifted and sheared data by 10%.[21] As seen in Fig. 11 tests with image augmentation always had higher testing accuracy than training accuracy, proving that it was helping the neural network generalize to new data and prevent overfitting.Image augmentation also caused the sharpest increase in testing accuracy on its own with a 10.77% increase from the base LeNet architecture.

```
#optimize dropout rate after second pooling layer

dropout_rates2 = {}
tracker = 1

for dropout in dropout_rate_range:
    print("Run {} out of {}".format(tracker, len(dropout_rate_range)))
    tempModel = testModel_HyperParamLoop(dropout_rate1 = OPTIMAL_DROPOUT_RATE1, dropout_rate2 = dropout) #create temporary model
    opt = keras.optimizers.Adam(learning_rate= OPTIMAL_LEARNING_RATE)
    tempModel.compile(optimizer = opt, loss = loss, metrics = ['accuracy']) #compile temporary model
    #fit model with optimal batch size
    tempModelHist = tempModel.fit(cnn_x_train, y_train, validation_split = OPTIMAL_VAL_SPLIT,
                                  batch_size= OPTIMAL_BATCH_SIZE, epochs=OPTIMAL_EPOCH_LEN, shuffle=True,
                                  callbacks = [early_stopping, history])

    trainAcc = tempModelHist.history['acc'][-1] #gets final training accuracy
    testLoss, testAcc = tempModel.evaluate(cnn_x_test, y_test) #gets test loss and accuracy
    print("Test accuracy for this model is {}".format(testAcc))
    dropout_rates2[dropout] = [trainAcc, testAcc] #save model accuracy in dictionary
    tracker += 1
```

Figure 12: Sample code of loop used to optimize hyper parameters

With the architecture and image augmentation optimized, we moved onto optimizing individual hyper parameters, specifically, kernel size, batch size, learning rate, epochs, validation split and dropout rate(s). The model used for finding the optimal hyper parameters was the additional optimal architecture without the image augmentation. This was mainly due to the additional time (3-5x) necessary to fit models with image augmentation and the need to loop over 5-12 values per parame-

ter. Optimizing each parameter was achieved through a series of loops, one sample shown in Fig. 12, that would test the same architecture with one varied parameter across a range of values. Whichever value gave the highest testing accuracy was deemed the optimal value. This series of loops followed a similar simplification as tests 1-6 since testing every possible combination of parameters was simply unfeasible due to complexity and time constraints. There is also a strand of ambiguity in the order in which the hyper parameters were optimized. They were optimized in the order listed above, but different results would probably be achieved if, for example, learning rate or dropout rate was optimized before batch size. An increase in batch size correlates with an increase in over fitting, but speeds up process time, so if dropout rate was optimized first, perhaps batch size would have been larger on average to compensate for the dropout rate. While the order is ambiguous, the order in which they were optimized remained constant for all testing runs.

3.3 Results and Discussion

			Hyper Parameters						
Run #	Testing Acc	Additional Layers	Kernel Size	Batch Size	Learning Rate	Epochs	Validation Split	DO Rate 1	DO Rate 2
1(no seed)	0.9841	1 DO, Conv (accident) Image Aug	3 (accident)	275	0.007	60	0.15	N/A	0.2
2 (no seed)	0.9886	1 DO, Conv (accident) Image Aug	3	150	0.005	40	0.15	N/A	0.1
3 (seed)	0.9907	2 DO, Image Aug	5	50	0.001	50	0.3	0.3	0.3
4 (seed)	0.9891	1 DO, Image Aug	5	25	0.003	50	0.15	0.3	N/A

Figure 13: Summary of all four full runs to determine an optimal model improvement from a default LeNet architecture. Runs 3 and 4 are most accurate as some non-optimal parameters were accidentally kept in later testing steps for Runs 1 and 2

A summary of the four completed runs with their respective architectures and hyper parameters is shown in Fig. 13 The best performing model utilized two 30% dropout layers and image augmentation to achieve a testing accuracy of 99.07%, a total improvement of 12% from the default LeNet architecture. Fig. 15 shows 9 of the signs incorrectly predicted by the optimal model. Since it is difficult to visually compare the 940 signs that the LeNet originally predicted incorrectly to the 67 signs the optimized model predicted incorrectly, confusion matrices were used to quantify which signs were most commonly confused. The confusion matrices for the LeNet model and optimized model from the best run are shown in in Fig. 20 and Fig. 21, respectively. From the optimized model's confusion matrix we can see that the only 3 signs it had a bit of trouble mixing up were 'B's as 'U's, 'R's as 'U's and 'T's as 'X's. An example of each of these is shown in Fig. 15 which shows that the 'T' that is being confused for an 'X' looks very similar in shape to an 'X' with a slight extended index finger, also seen in Fig. 1. The incorrectly predicted 'R's also look very similar to 'U' as seen in Fig. 1. The incorrectly predicted 'B' as 'U' is confusing because it should recognize that the ring and pinky finger should be down, but it's possible that the weights favor more of the index and middle finger showing to predict that a sign is a 'U.' Both of the LeNet's confusion matrix in Fig. 20 and the grid of incorrectly predicted signs in Fig. 14 show the magnitude of mistakes being made, with a highest misclassification of 'R's as 'U's with 36%. To be fair, the signs incorrectly labeled in Fig. 14 do have some similar shapes, like 'L' and 'C' both having a sort of open gap between the index finger and thumb, and 'Y' and 'V' both having a V-shape between two fingers, but these few pictures alone show how the model was over fitting to certain shapes. With some misclassification rates of 36%, it's clear that the LeNet Model recognized

some relevant shapes but not more ambiguous features. These can be further seen in the classification of 'G' and 'H' and 'M' and 'N' in each model. The shapes of these signs are very similar to each other respectively, but in the optimized model there is 0 incorrectly predicted 'G','H','M','N' letters. The optimized model also showed a significant reduction in loss from the original LeNet architecture as seen in Fig. 22 and Fig. 23.

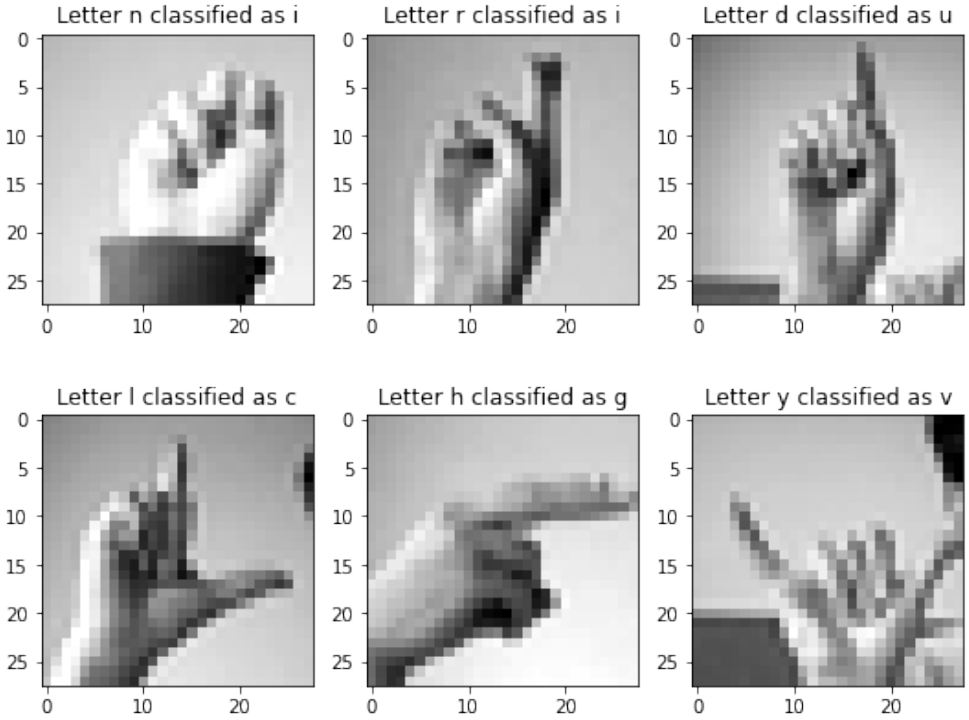

Figure 14: 6/940 of the signs incorrectly predicted by the LeNet model

4 Conclusion

There are several improvements and directions this work could go in the future. This includes improving on the current model and attempting to use the model on new data sets with more signs that the letters of the alphabet. Some improvements and considerations on the model itself, assuming the architecture remains the same, is improving and more effectively testing the best combination of hyper parameters. There are several hyper parameters, like stride, padding, filters, and pool size that were not adjusted at all in this report which could be useful to examine. Another feature that may be more important to optimize are the values used to augment the data. These numbers were chosen mostly arbitrarily based on the the ImageDataGenerator documentation and kept constant throughout all tests. Since image augmentation had the largest singular effect on improving test accuracy, optimizing it is especially important and may lead to 100% testing accuracy. One challenge that came along with image augmentation was additional time required to fit the training data to the model. This cause an increase in time up to about 5x the time it took to fit the model without image augmentation. If time is a necessary resource, which it very may well be if the intended applications in the future are live translation of some sort, then it may be useful to investigate how to either lower the time it takes to fit with image augmentation, or bring the model up to similar accuracy without image augmentation. Another challenge to face to make the model more reliable would be to identify

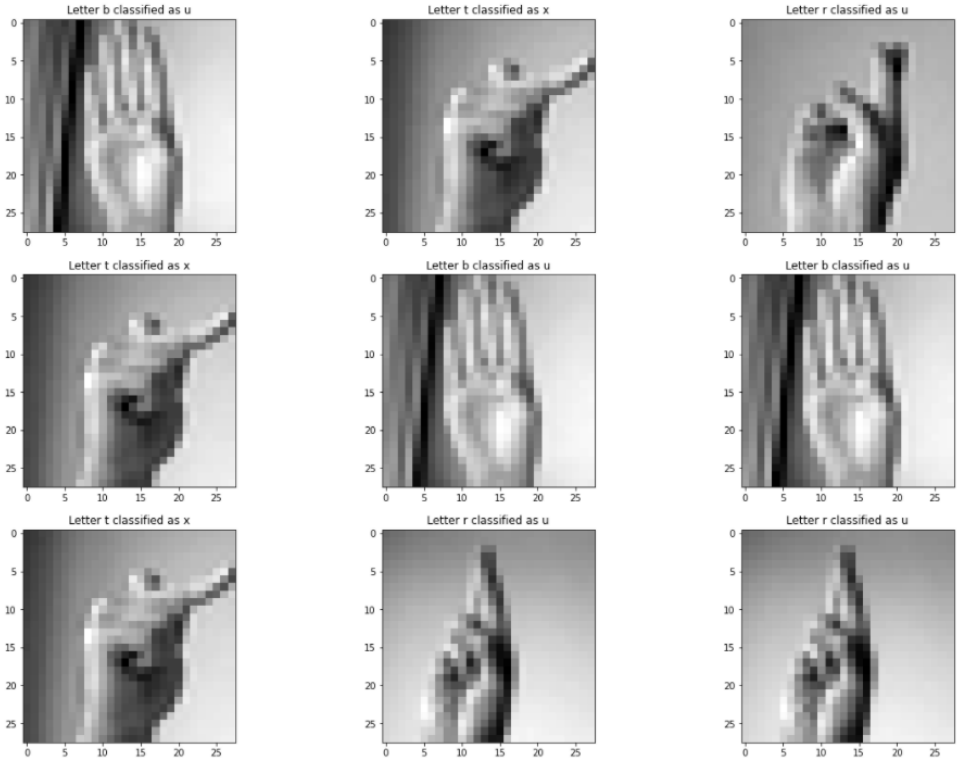

Figure 15: 9/67 of the signs incorrectly predicted by the optimal model

all the sources of randomness that occur in Tensorflow's algorithms in order to set proper seeds and get reproducible results. Although this project is confined to static images of signs, it would be important to explore the incorporation of computer vision to capture signs that require movement and facial expressions which assist in ASL grammar. This was out of scope for the project, but would be integral if live complex ASL to English programs were created.

There is a lot left to be done on this project, but we are proud of the work we accomplished in raising a relatively old neural network's architecture to 99% accuracy on the American Sign Language MNIST dataset. We hope this project can shine a light on the importance of inclusion and communication as technology advances even further.

References

(1) Alake, R. (You Should) Understanding Sub-Sampling Layers Within Deep Learning https://towardsdatascience.com/you-should-understand-sub-sampling-layers-within-deep-learning-b51016acd551.

(2) Alake, R. Understanding and Implementing LeNet-5 CNN Architecture (Deep Learning) https://towardsdatascience.com/understanding-and-implementing-lenet-5-cnn-architecture-deep-learning-a2d531ebc342.

(3) Bird, J. Leap Motion ASL and BSL Sign Language Recognition https://www.kaggle.com/birdy654/sign-language-recognition-leap-motion.

(4) BrianMH ECE 196 CNN Demo https://github.com/BrianMH/ECE196_Demo/blob/master/CNN_Demo.ipynb.

(5) BrownLee, J. How to get Reproducible Results with Keras https://machinelearningmastery.com/reproducible-results-neural-networks-keras/.

(6) DeafJobWizard Unemployment in the Deaf Community: Barriers, Recommendations and Benefits of Hiring Deaf Employees https://www.pbs.org/weta/throughdeafeyes/deaflife/technology.html.

(7) Geislinger, V. ASL Fingerspelling Images (RGB Depth) https://www.kaggle.com/mrgeislinger/asl-rgb-depth-fingerspelling-spelling-it-out.

(8) Huang, P. Intro to Deep Learning https://github.com/PoH0716/Intro_to_DL/blob/main/ECE196_lecture7.pdf.

(9) Jacob Hohn, S. R. ASL CNN Image Recognition https://github.com/jfhohn/ASL_CNN_Image_Recognition.

(10) Kaggle Digit Recognizer - Learn computer vision fundamentals with the famous MNIST data https://www.kaggle.com/c/digit-recognizer.

(11) PBS Through Deaf Eyes https://www.pbs.org/weta/throughdeafeyes/deaflife/technology.html.

(12) Prabhu Understanding of Convolutional Neural Network (CNN) — Deep Learning https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148.

(13) Radhakrishnan, P. What are Hyperparameters ? and How to tune the Hyperparameters in a Deep Neural Network? https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a.

(14) Repository, U. M. Australian Sign Language signs (High Quality) Data Set https://archive.ics.uci.edu/ml/datasets/Australian+Sign+Language+signs+%5C%28High+Quality%5C%29.

(15) Sarkar, A. Creating vgg from Scratch in Tensorflow https://towardsdatascience.com/creating-vgg-from-scratch-using-tensorflow-a998a5640155.

(16) Shah, T. About Train, Validation and Test Sets in Machine Learning https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7.

(17) tecperson Sign Language MNIST Drop-In Replacement for MNIST for Hand Gesture Recognition Tasks https://www.kaggle.com/datamunge/sign-language-mnist.

(18) TensorFlow Adam API https://keras.io/api/optimizers/adam/.

(19) TensorFlow Dropout layer API https://keras.io/api/layers/regularization_layers/dropout/.

(20) TensorFlow Flatten layer API https://keras.io/api/layers/reshaping_layers/flatten/.

(21) TensorFlow ImageDataGenerator API https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator.

(22) TensorFlow Layer activation functions API https://keras.io/api/layers/activations/.

(23) TensorFlow The Model Class API https://keras.io/api/models/model/.

5 Appendix

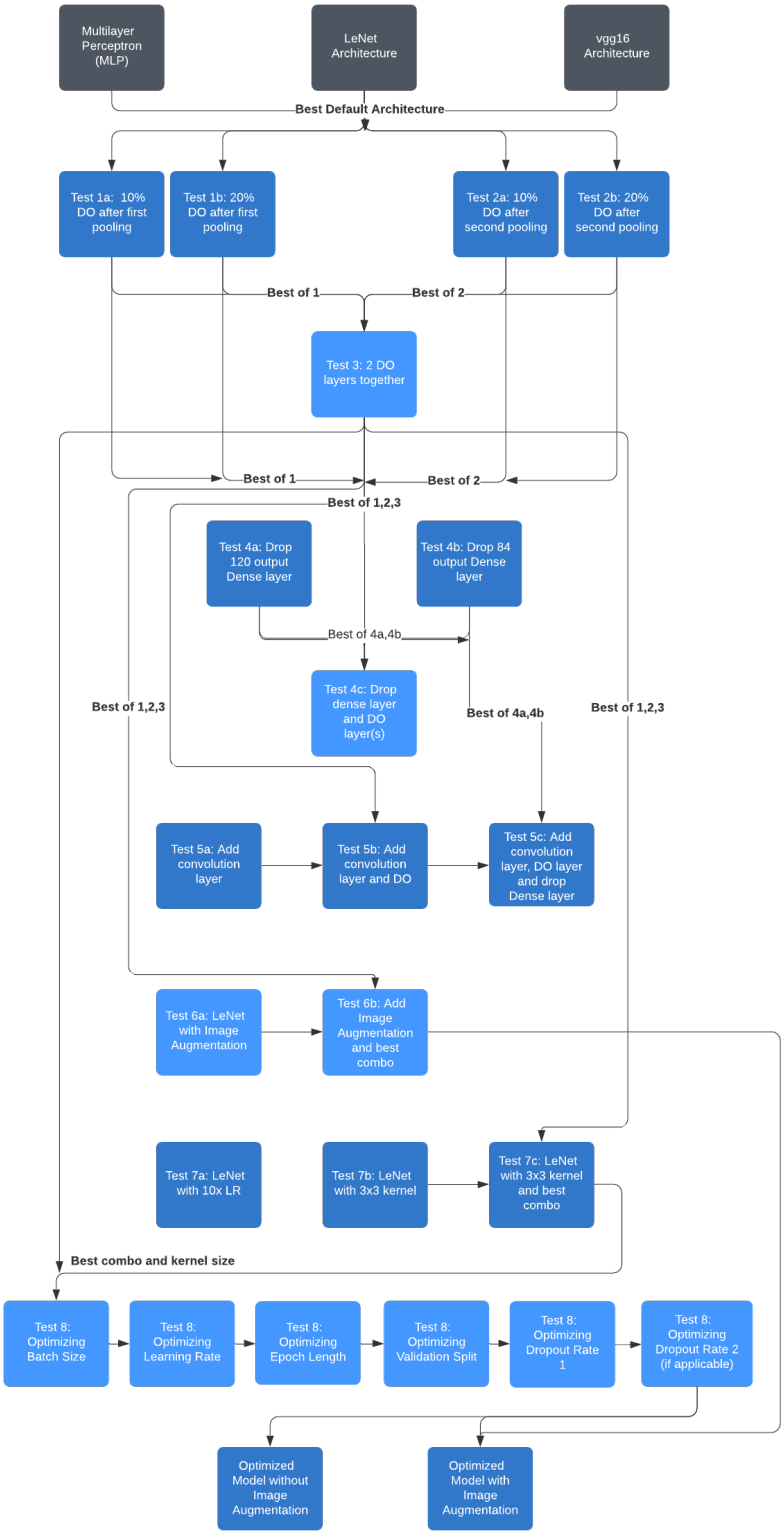

Figure 16: Testing workflow for project to obtain optimized model from a default architecture

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 784)]	0
dense (Dense)	(None, 512)	401920
dense_1 (Dense)	(None, 512)	262656
dense_2 (Dense)	(None, 25)	12825

Total params: 677,401
Trainable params: 677,401
Non-trainable params: 0

Figure 17: Multilayer Perceptron Architecture

Layer (type)	Output Shape	Param #
input_69 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d_145 (Conv2D)	(None, 24, 24, 6)	156
average_pooling2d_130 (Avera	(None, 12, 12, 6)	0
conv2d_146 (Conv2D)	(None, 8, 8, 16)	2416
average_pooling2d_131 (Avera	(None, 4, 4, 16)	0
flatten_66 (Flatten)	(None, 256)	0
dense_133 (Dense)	(None, 120)	30840
dense_134 (Dense)	(None, 84)	10164
prelogits (Dense)	(None, 25)	2125
logits (Activation)	(None, 25)	0

Total params: 45,701
Trainable params: 45,701
Non-trainable params: 0

Figure 18: LeNet Architecture

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 28, 28, 1)]	0
conv2d_2 (Conv2D)	(None, 28, 28, 64)	640
conv2d_3 (Conv2D)	(None, 28, 28, 64)	36928
max_pooling2d (MaxPooling2D)	(None, 14, 14, 64)	0
conv2d_4 (Conv2D)	(None, 14, 14, 128)	73856
conv2d_5 (Conv2D)	(None, 14, 14, 128)	147584
max_pooling2d_1 (MaxPooling2	(None, 7, 7, 128)	0
conv2d_6 (Conv2D)	(None, 7, 7, 256)	295168
conv2d_7 (Conv2D)	(None, 7, 7, 256)	590080
conv2d_8 (Conv2D)	(None, 7, 7, 256)	590080
max_pooling2d_2 (MaxPooling2	(None, 4, 4, 256)	0
conv2d_9 (Conv2D)	(None, 4, 4, 512)	1180160
conv2d_10 (Conv2D)	(None, 4, 4, 512)	2359808
conv2d_11 (Conv2D)	(None, 4, 4, 512)	2359808
max_pooling2d_3 (MaxPooling2	(None, 2, 2, 512)	0
conv2d_12 (Conv2D)	(None, 2, 2, 512)	2359808
conv2d_13 (Conv2D)	(None, 2, 2, 512)	2359808
conv2d_14 (Conv2D)	(None, 2, 2, 512)	2359808
max_pooling2d_4 (MaxPooling2	(None, 1, 1, 512)	0
flatten_1 (Flatten)	(None, 512)	0
dense_5 (Dense)	(None, 1024)	525312
dense_6 (Dense)	(None, 1024)	1049600
dense_7 (Dense)	(None, 25)	25625

Total params: 16,314,073
Trainable params: 16,314,073
Non-trainable params: 0

Figure 19: vgg16 Architecture

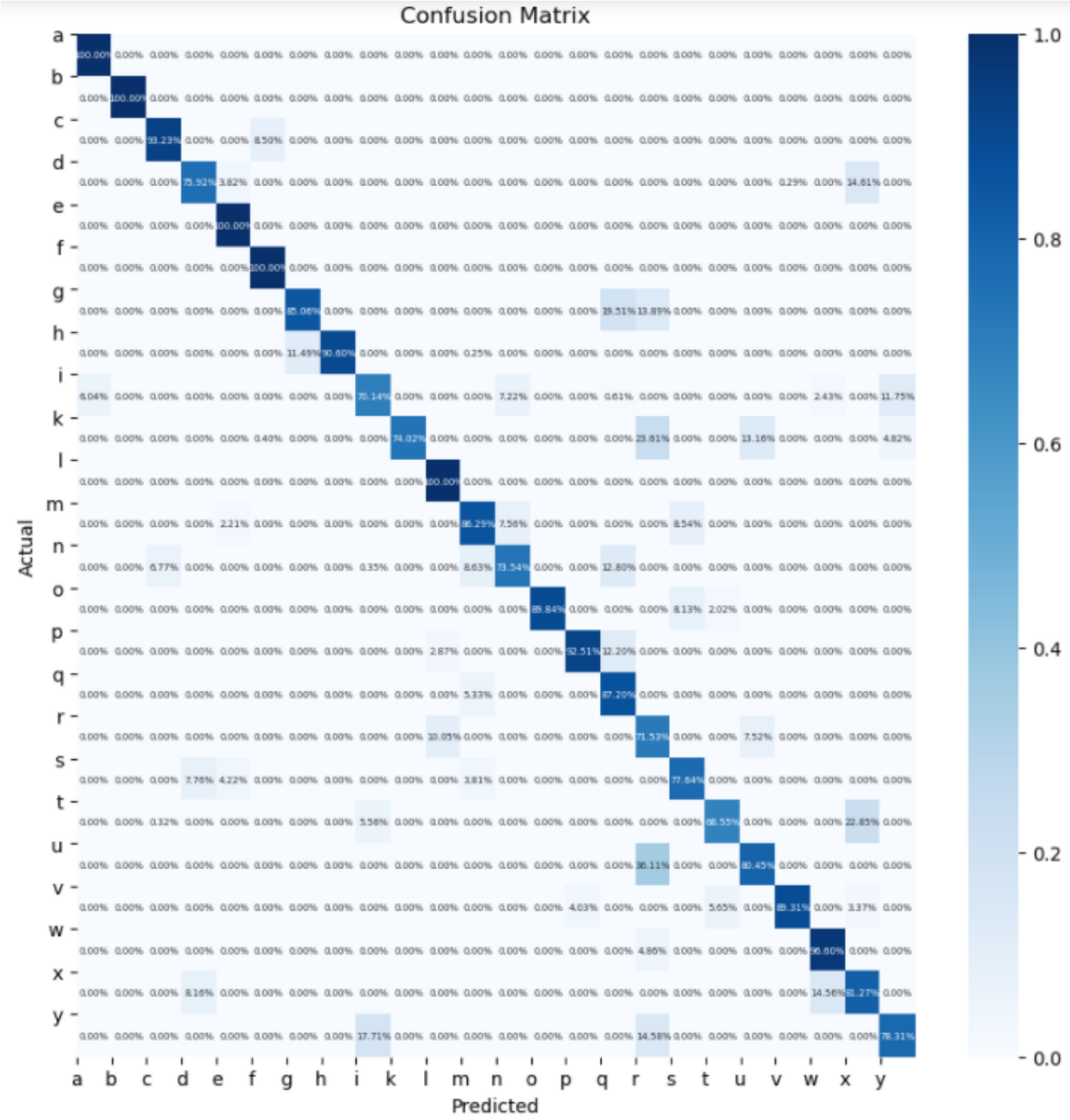

Figure 20: Confusion matrix for the best run's LeNet model

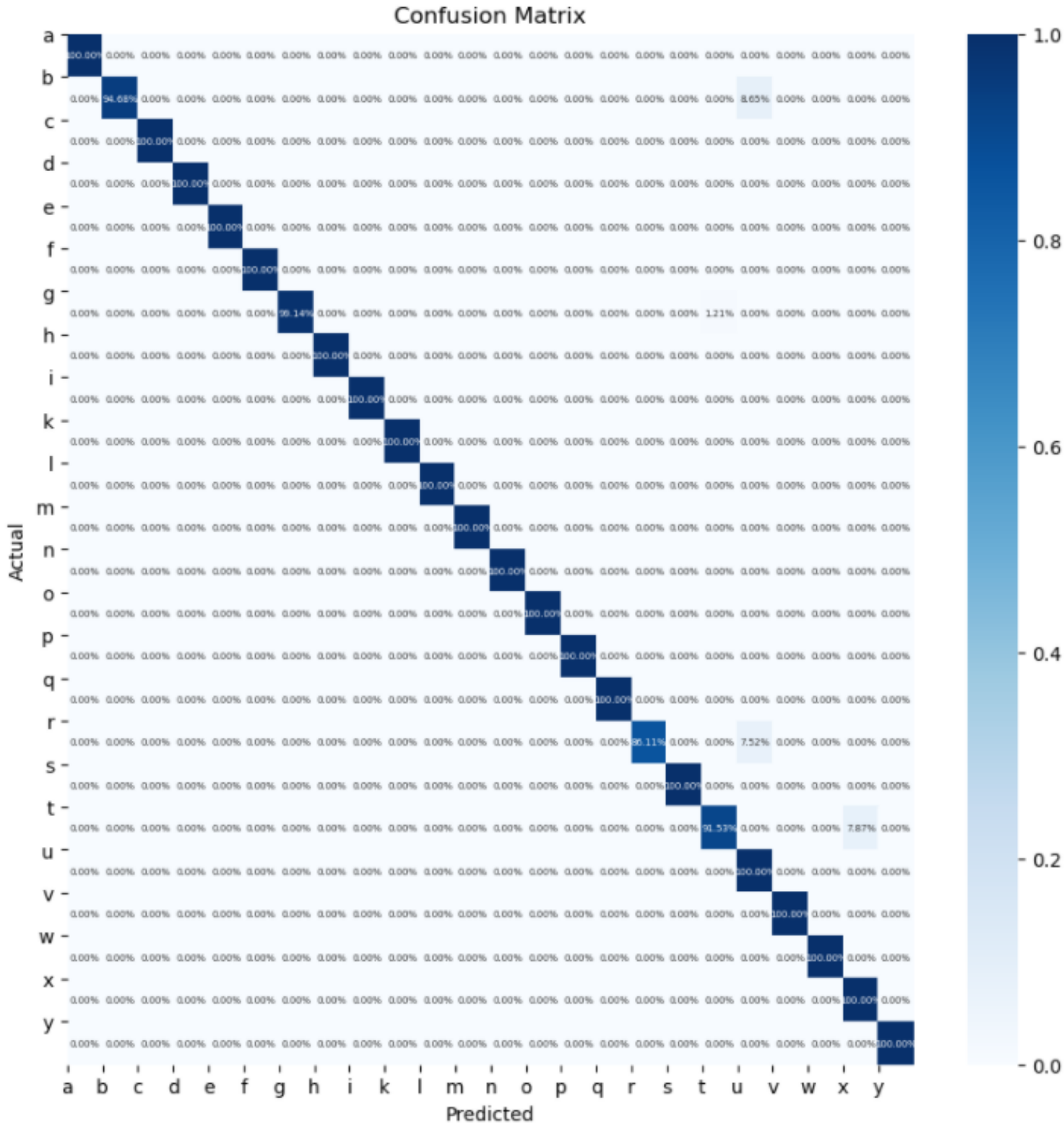

Figure 21: Confusion matrix for the best run's optimized model

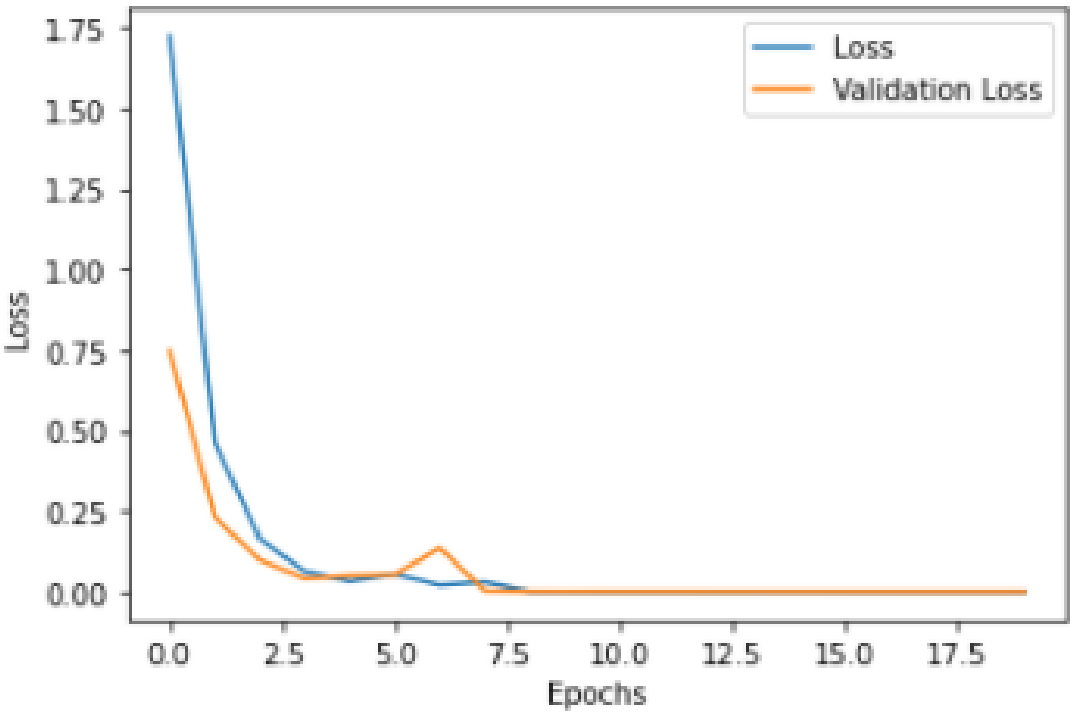

Figure 22: Loss graph for the best run's LeNet model

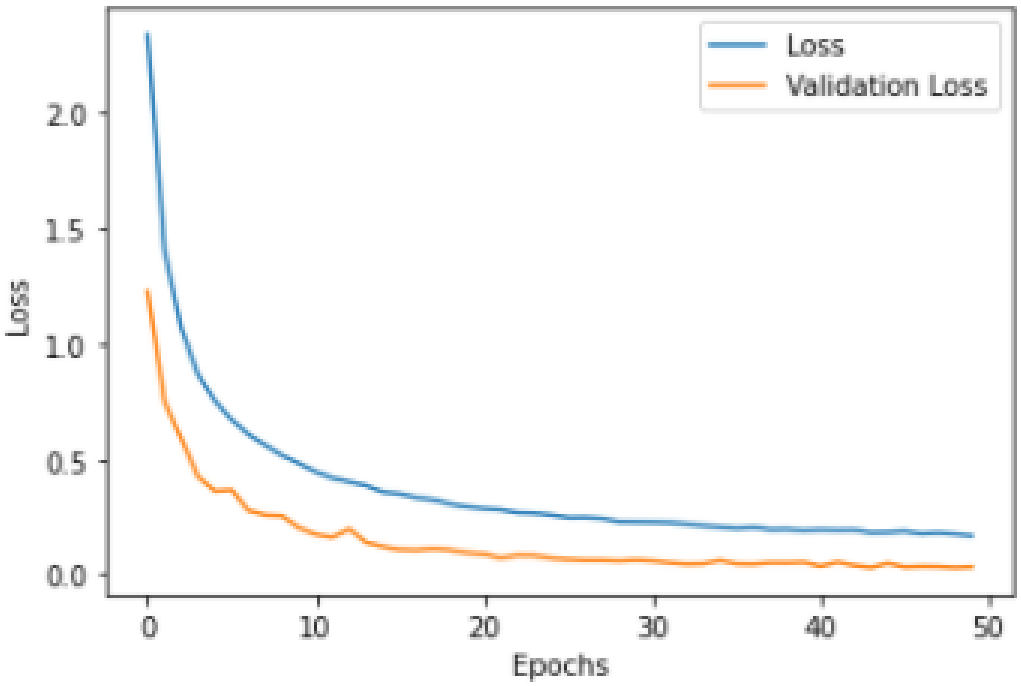

Figure 23: Loss graph for the best run's optimized model

```
# LeNet model with optimized hyperparameters and architecture
def optimizedModel(dropout_rate1 = OPTIMAL_DROPOUT_RATE1, dropout_rate2 = OPTIMAL_DROPOUT_RATE2): # added optimal dropout rate
    # input
    xIn = keras.Input(shape=(28,28,1))

    # subsequent layers
    out = keras.layers.Conv2D(6, OPTIMAL_KERNEL_SIZE, activation='relu')(xIn)
    out = keras.layers.AveragePooling2D(2, 2)(out)

    out = keras.layers.Dropout(rate = OPTIMAL_DROPOUT_RATE1, seed = SEED)(out) # rate = 30% DO

    out = keras.layers.Conv2D(16, OPTIMAL_KERNEL_SIZE, activation='relu')(out)
    out = keras.layers.AveragePooling2D(2, 2)(out)

    out = keras.layers.Dropout(rate = OPTIMAL_DROPOUT_RATE2, seed = SEED)(out) # rate = 30% DO

    out = keras.layers.Flatten()(out)
    out = keras.layers.Dense(120, activation='relu')(out)
    out = keras.layers.Dense(84, activation='relu')(out)
    out = keras.layers.Dense(25, name='prelogits')(out)
    out = keras.layers.Activation('softmax', name='logits')(out)

    # Creates model
    mod = keras.Model(inputs=xIn, outputs=out)

    return mod

# Declares the model and prepares it for training
optimalMod = optimizedModel()
opt = keras.optimizers.Adam(learning_rate = OPTIMAL_LEARNING_RATE) #lr = default = 0.001
optimalMod.compile(optimizer=opt, loss=loss, metrics=['accuracy'])
optimalMod.summary()
```

Figure 24: Example Code for initializing model architecture and initializing model with Adam Optimization and Sparse Categorical Cross Entropy loss function

```
%%time

optimalModHist = optimalMod.fit(cnn_x_train, y_train, validation_split = OPTIMAL_VAL_SPLIT,
                                batch_size=OPTIMAL_BATCH_SIZE, epochs=OPTIMAL_EPOCH_LEN,
                                shuffle=True, callbacks = [early_stopping, history])
```

Figure 25: Example code for training the model with the initialized architecture from Fig. 24 Utilize EarlyStopping to prevent running for an unneccessary amount of epochs. Utilize History to access accuracy and loss data after training is complete

```
# Evaluate the trained network on the testing dataset
testLoss, testAcc = optimalMod.evaluate(cnn_x_test, y_test)
print("Test accuracy for this model is {}".format(testAcc))
```

Figure 26: Example code to evaluate model trained via Fig. 25 on test data.

```
a = optimalModHist.history['loss']
b = optimalModHist.history['val_loss']
plt.plot(range(len(a)),a)
plt.plot(range(len(b)),b)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Loss','Validation Loss'])
```

Figure 27: Example code used to retrieve and plot model loss graphs like Fig. 22 and Fig. 23

```
#Plot optimized confusion matrix
alphabet = ['a', 'b',  'c',  'd', 'e', 'f',
                'g', 'h', 'i',  'k',  'l',
                 'm',  'n',  'o',  'p',  'q',
                 'r', 's',  't',  'u',  'v',
                 'w','x','y']
conMat = confusion_matrix(y_test, preds)
plt.figure(figsize=(10,10), dpi=100)
heatmap(conMat/np.sum(conMat,axis=1), annot=True, annot_kws={"size": 5}, fmt='.2%', cmap='Blues')
plt.title("Confusion Matrix")
plt.xticks(range(len(alphabet)), alphabet)
plt.yticks(range(len(alphabet)), alphabet, rotation='horizontal')
plt.xlabel("Predicted")
plt.ylabel("Actual")
```

Figure 28: Example code to plot a confusion matrix like Fig. 20 or Fig. 21 for a given model

```
# Shows 6 random incorrect predictions with labels
preds = optimalMod.predict(cnn_x_test)
preds = np.argmax(preds, axis=1) # gives the corresponding label
preds_alphabet = numToAlpha(preds)
incorrectPreds = np.argwhere(preds_alphabet != labels_test_alphabet)
print("There are {}/{} incorrectly labeled signs".format(incorrectPreds.shape[0],cnn_x_test.shape[0]))
plt.figure(figsize=(9,7))

for i in range(6):
    plt.subplot(2,3,i+1)
    randIncInd = np.random.choice(incorrectPreds.reshape(-1))
    plt.imshow(cnn_x_test[randIncInd].reshape(28,28), cmap = 'gray')
    plt.title("Letter {} classified as {}".format(labels_test_alphabet[randIncInd], preds_alphabet[randIncInd]))

plt.show()
```

Figure 29: Example code for visualizing incorrectly predicted image samples for a given model like Fig. 14 and Fig. 15

	test	training loss	training acc	testing loss	testing acc
0	LeNet original	0.000319	1.000000	0.854271	0.847741
1	vgg16 reshaped	3.177366	0.043417	3.204896	0.020078
2	1a: 10% DO	0.009809	0.997815	0.554227	0.893893
3	1b: 20% DO	0.015237	0.995338	0.408104	0.914668
4	2a: 10% DO	0.012371	0.996746	0.681100	0.860987
5	2b: 20% DO	0.110056	0.962994	0.332451	0.900725
6	3: 2 DO layers	0.035469	0.988247	0.352887	0.905745
7	4a: remove 120 dense	0.001345	1.000000	1.073727	0.828779
8	4b: remove 84 dense	0.000398	1.000000	0.734963	0.866983
9	4c: remove 84 dense, add 20% DO	0.076893	0.974746	0.432936	0.900586
10	5a: Add Conv 32 out	0.000269	1.000000	0.988132	0.862660
11	5b: Add Conv 32 out and 20% DO	0.090768	0.971201	0.343845	0.899610
12	6a: Add Image Aug	0.145891	0.952796	0.109960	0.970998
13	6b: Add Image Aug and 20% DO	3.176303	0.043562	3.204536	0.034300
14	7a: 10x LR	0.000024	1.000000	1.962970	0.814975
15	7b: 3x3 kernel	0.000689	1.000000	0.604427	0.886503
16	7b: 3x3 kernel and 20% DO	0.009542	0.996649	0.432969	0.893893
17	6b: Add Image Aug and 20% DO	0.257268	0.912694	0.073057	0.983826
18	Final Test: Optimized Model	0.092606	0.970388	0.041253	0.984105

Figure 30: Full Results from Run 1: Optimized Model with 1 20% dropout layer(random seeds) after second pooling, 1 (accidentally added) convolution layer, image augmentation and optimized hyper parameters (seen in Fig. 13. 'Optimized' model used 1 dropout layer after second pooling instead of first pooling incorrectly because a first run (not saved) showed better performance with 1 layer after second pooling

	test	training loss	training acc	testing loss	testing acc
0	LeNet original	0.000356	1.000000	0.641477	0.878834
1	vgg16 reshaped	3.177478	0.045942	3.201718	0.022867
2	1a: 10% DO	0.012577	0.996503	0.657679	0.872560
3	1b: 20% DO	0.019979	0.994075	0.510889	0.901841
4	2a: 10% DO	0.012457	0.996309	0.718474	0.878277
5	2b: 20% DO	0.032945	0.989219	0.384276	0.899610
6	3: 2 DO layers	0.027539	0.991695	0.327521	0.923592
7	4a: remove 120 dense	0.001126	1.000000	0.543059	0.874094
8	4b: remove 84 dense	0.031326	0.990287	0.569954	0.873815
9	4c: remove 84 dense, add 20% DO	0.034731	0.988733	0.390826	0.902398
10	5a: Add Conv 32 out	0.000054	1.000000	0.960747	0.866704
11	5b: Add Conv 32 out and 20% DO	0.093266	0.969161	0.341435	0.880368
12	6a: Add Image Aug	0.096555	0.967948	0.051226	0.979922
13	6b: Add Image Aug and 20% DO	0.148480	0.950428	0.050605	0.981595
14	7a: 10x LR	0.144857	0.968287	3.042614	0.635806
15	7b: 3x3 kernel	0.000110	1.000000	0.608290	0.892499
16	7b: 3x3 kernel and 20% DO	0.011779	0.996309	0.443731	0.916202
17	Final Test: Optimized Model	0.063183	0.980368	0.048058	0.988567

Figure 31: Full Results from Run 2: Optimized Model with 1 20% dropout layer(random seeds) after second pooling, 1 (accidentally added) convolution layer, image augmentation and optimized hyper parameters (seen in Fig. 13. 'Optimized' model used 1 dropout layer instead of 2 incorrectly as well because a first run (not saved) showed better performance with 1 layer

	test	training loss	training acc	testing loss	testing acc
0	Multilayer Perceptron	0.000666	1.000000	0.969787	0.819994
1	LeNet original	0.000204	1.000000	0.791391	0.872699
2	vgg16 reshaped	3.177631	0.045020	3.199109	0.034300
3	1a: 10% DO	0.007363	0.997572	0.618351	0.871863
4	1b: 20% DO	0.016975	0.994852	0.434261	0.920245
5	2a: 10% DO	0.018894	0.994318	0.484493	0.891523
6	2b: 20% DO	0.028269	0.991161	0.370297	0.910625
7	3: 2 DO layers	0.034148	0.989461	0.366539	0.917596
8	4a: remove 120 dense	0.000280	1.000000	0.818193	0.858059
9	4b: remove 84 dense	0.065541	0.981351	0.734760	0.862800
10	4c: remove 84 dense, add 2 DO layers	0.019708	0.994949	0.408556	0.906999
11	5a: Add Conv 32 out	0.162883	0.965276	0.713223	0.856804
12	5b: Add Conv 32 out and 1 DO layer	0.026329	0.991258	0.672266	0.881065
13	6a: Add Image Aug	0.094404	0.967948	0.117184	0.973369
14	6b: Add Image Aug and 1 DO layer	0.109086	0.963431	0.029160	0.990658
15	7a: 10x LR	3.177610	0.044534	3.203620	0.022867
16	7b: 3x3 kernel	0.045240	0.985091	0.494812	0.894451
17	7b: 3x3 kernel and 1 DO layer	0.019123	0.993929	0.406987	0.903932
18	Optimized Model without Image Augmentation	0.033625	0.992244	0.349753	0.940881
19	Optimized Model with Image Augmentation	0.131211	0.961209	0.053009	0.989124

Figure 32: Full Results from Run 4: Optimized Model with 1 20% dropout layer(constant seed) after first pooling, image augmentation and optimized hyper parameters (seen in Fig. 13. Optimized model based on test workflow shown in Fig. 16 and adjusted throughout individual run as intended