# A Deep Learning Approach to American Sign Language Character Classification

Arvind Shankar, Ryan Leavitt, James Dika, Stephen Young

*Abstract* – **This paper describes a supervised machine learning classifier for a subset of American Sign Language (ASL) letters. First, data were collected by students in the class signing the letters 'A' through 'I' and saving them as 100x100 color images. Then, given a dataset of 1844 images with their corresponding labels, we were tasked with building a model capable of classifying images with at least 90% accuracy. Our classifier implementation consisted of first preprocessing the data by partitioning the dataset into training and validation sets, then converting the images to grayscale. Next, we trained a Convolutional Neural Net (CNN) on the grayscale data by passing data in batches and updating the weights in the network through backpropagation using the Adam optimizer. To fine-tune the model, we conducted a large number of experiments to determine the optimal values for many different parameters in the network. Finally, we were able to generate a model capable of classifying images of the ASL letters with nearly 90% accuracy.**

## I. INTRODUCTION

The task given to our group was to apply the information presented in class and design a classifier capable of classifying images of ASL letters 'A' through 'I' with at least 90% accuracy. We were to follow all of the steps of a standard machine learning pipeline: data acquisition, data preprocessing and feature extraction/generation, model selection, model training and hyperparameter tuning, and finally validation.

In the data acquisition stage, the class generated 100x100 color images of the nine ASL letters, of which 1844 were provided to us. We used this provided dataset alongside a "Sign Language MNIST" dataset we found online [1]. Since a balanced dataset is necessary for the model to generalize well, we also looked at the distribution of the datasets and we found that both datasets were evenly distributed across all classes. Next, in the preprocessing stage, we made transformations to the MNIST dataset so that we could use it alongside our provided dataset.

The next stage was preprocessing. This stage is necessary to reduce the dimensionality of the feature space and because normalizing data is usually one of the first pre-processing techniques applied to the data [1]. First, we encoded the provided labels to be integers to work with our CNN. Next, we partitioned the provided dataset into training and test sets, giving us a blind test set to evaluate our models on.

From the provided dataset, we created two new sets. The first had grayscale images (as shown above), the second had grayscale images but pixel values were standardized to have zero mean and unit variance. We also had to preprocess the MNIST dataset to combine it with the provided grayscale dataset. As a result, at the end of preprocessing, we had three datasets. Each of these datasets was partitioned into training and validation sets with an 80/20 split so that we could test our model on a blind set.

After preprocessing, we had to determine which classifier to use. We conducted a series of experiments training and validating a variety of classifiers including Linear Discriminant Analysis (LDA), k-Nearest Neighbors (k-NN), Decision Trees and Random Forests, Support Vector Machines (SVM), Multi-Layer Perceptrons (MLP), and finally CNNs. Out of all of these, the CNN performed the best by far, which made sense because, as the literature suggests, the deep learning approach essentially skips the feature generation step [2].

The final stage was to tune the hyperparameters of the CNN such that we would get the highest validation accuracy. In this stage, we conducted many experiments and determined optimal values for the following parameters: number of convolutional layers, kernel size, number of inputs to the first linear layer, batch size and number of epochs. We also determined in this stage that the un-standardized, grayscale dataset from our preprocessing stage was the best dataset to train our model on, but this result will be discussed later. In the next section we will describe in more detail our final implementation.

## II. IMPLEMENTATION

### *A. Preprocessing*

The images in the provided dataset were color images with 100x100x3 pixels and their values were in the range 0-255. Since the number of pixels is the dimensionality of the feature space (dimensionality is the same as the number of features), we would have 30 000 dimensions without any preprocessing. So, to reduce this dimensionality we decided to use grayscale data. By eliminating the 3 color channels, we reduced dimensionality to 10 000. A sample is shown below:

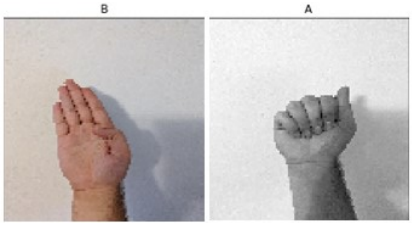

Figure 1 'B' before pre-processing, 'A' after rgb2gray

The provided data and labels were stored as two separate NumPy ndarray objects, so first we loaded those in. Regarding the labels, we found that we would get code errors if we did not encode them, so we encoded them as integers. This task was accomplished by first flattening the "labels" matrix into a one-dimensional vector and then using the "LabelEncoder" from sklearn to create a one-dimensional labels vector with integer entries. Next, we partitioned the data into 80% training and 20% validation sets using the "train_test_split" function from sklearn. When generating the three transformed datasets, we applied the same transformations to both training Sand validation partitions.

To generate the first transformed dataset (the grayscale provided data), we used the "rgb2gray" function from sklearn. In our preprocessing step, this function takes in an Mx100x100x3 matrix and returns an Mx100x100 matrix of grayscale images, where M is the number of images. Because a CNN accepts two-dimensional matrices as inputs, we used this dataset for training and testing the final classifier. However, in order to standardize this grayscale provided set, we needed to flatten it into a two-dimensional matrix of size Mx10 000. After reshaping each image and storing them into a new matrix, we used the "scale" function from the sklearn preprocessing library to generate the standardized grayscale set.

Unfortunately, we still only had 1844 images in the provided set - far fewer images than features. That is why we added the MNIST data. This dataset included 28x28 grayscale images of all ASL letters except 'J' and 'Z,' for a total of 27 455 images. The labels were encoded as integers values zero through eight. Since we only need the first nine letters, we first had to extract rows with the label values less than nine to get the image data. From that subset, the labels were contained in the first column, so that column became the labels for the new image data subset. We then normalized this data subset by dividing all the pixel values by 255 to bring them in the range [0,1]. Next, we used the OpenCV to resize the images to be 100x100. After that step, we concatenated this reduced and resized MNIST data with the grayscale data, and concatenated the MNIST labels with the encoded, provided labels. We then had the relevant MNIST data combined with our provided grayscale data, giving us a new training set with 14 512 images. This was the only dataset we experimented on that had more samples than features.

As for other methods of preprocessing to reduce dimensionality, early in this project we tried to do Principal Component Analysis (PCA), Histogram of Oriented Gradients (HOG) for dimensionality reduction and feature generation. None of the datasets generated from these models performed well in validation and they had low scores for Calkinski-Harabasz Index, Davies-Bouldin Index, and Silhouette Index. LDA feature spaces also performed terribly in validation. So, we decided not to use any of these methods for dimensionality reduction.

### *B. Model Selection – Building the CNN*

After seeing poor validation results on our datasets with a variety of classifiers, we decided to investigate a deep learning approach for our model. The literature suggested that deep learning could be used in cases where there is little to no preprocessing and that we could pass in basically raw input and not have to deal with feature extraction or generation. Since we had no prior experience with deep learning methods, we had to follow tutorials and documentation to build a functioning CNN. We used the PyTorch [3] framework since its methods were simple to use and we could also use a GPU to accelerate the network functions which allowed us to greatly speed up training and validation. Once we had a functional CNN, we then started to optimize the CNN parameters through experimentation.

The first CNN that we successfully trained and tested had the following parameters: 3 convolutional layers, each having a 3x3 kernel, in which the first had 1 input channel and 32 output channels, the second had 32 input and 64 output channels, and the third had 64 input and 128 output channels. Between each of the convolutional layers was a two-dimensional pooling layer with ReLu activation and a 2x2 pool size. Next, there were two fully-connected linear layers. The first had 12 800 input features, 512 output layers, and used the ReLu activation function since that is the standard for fully-connected layers in CNNs [4]. The second had 512 input features and 9 output features (since there are 9 output classes). The output layer used a SoftMax activation function, which outputs a vector representing a probability distribution that sums to one and in which the highest value corresponds to the predicted class labels. The PyTorch argmax function, which returns the index with the highest value in the input, can then be used to determine the predicted class label. Finally, we chose the loss function (what we are trying to minimize by adjusting weights in the network) to be cross-entropy loss. This choice was based on the resources suggesting that it is the most suitable choice alongside SoftMax activation for a multi-class system [4].

In general, training a CNN involves passing the full dataset through the network and using the

backpropagation algorithm to update the weights in a way that reduces the cost function. We use the method of gradient descent to find the direction in which to update the weights. To optimize gradient descent, we chose to use the Adam optimizer, which computes adaptive learning rates for each parameter in the network [5]. In the next section, we will discuss the parameters we varied in training to get the best CNN.

## III. EXPERIMENTS

Our experimentation stage consisted of training and evaluating many CNNs while varying the following parameters: the size of the network layers, the kernel size, which training set was used, minibatch size, and the number of epochs. Each CNN we fit was then evaluated on the validation partition of the grayscale provided data, since that would be representative of the real test set.

### *A. Network Architecture*

To determine the optimal number of layers and kernel size, we trained and evaluated 4 different CNNs on the provided grayscale dataset with a constant batch size of 256 and 200 epochs. We used 3 and 4 convolutional layers, each with kernel sizes 3 and 5. The best CNN was the one with 3 convolutional layers and kernel size 3 by far, so we chose those values for those parameters and continued experimenting.

To find the ideal number of inputs to the linear layer of the network, we trained 2 CNNs with different values for the output channels, therefore changing the number of inputs to the first linear layer, while keeping the other parameters constant. We thought it was possible that reducing the size of the last convolutional layer would cause less overfitting [6], but we got a significantly higher validation score in the model with more inputs to the linear layer, so we decided to stick with that layer size.

### *B. Training Set*

One important decision was whether to use an expanded dataset that included thousands of extra training samples from MNIST. In the preprocessing stage we generated 3 training sets to find which performed best: 1) 14 512 samples of MNIST data mixed with provided grayscale data, 2) 1475 samples using only the grayscale provided data, and 3) a standardized set of 1475 samples using only the grayscale provided data after it has been transformed to have mean 0 and unit variance. We chose to train our final model on whichever training set gave the best score on the grayscale provided validation set.

This experiment consisted of training 3 CNNs with each parameter constant except for the training dataset. Based on the validation scores of the three models created using the different datasets, the best performance was obtained using just the grayscale provided data. The combined dataset did not perform as well on the validation set. The standardized dataset performed the worst of the three, likely due to the linear layers using ReLu activation functions, which cause negative values in this dataset to drop out. At this point, we decided to train the model with only the grayscale provided dataset.

### *C. Batch Size and Number of Epochs*

The final hyperparameters we needed to be optimized to create the model were batch size and number of epochs. These parameters are closely related since the smaller the batch size gets, each epoch requires more computations as more iterations are required for one pass over the dataset.

For this test, batch sizes 8, 32, 64, were each tested with 300, 500, and 1000 epochs for a total of 9 different models. All these models yielded validation accuracy over 80%, but they also had training accuracies over 99%, indicating overfitting [7]. To combat overfitting, we increased batch sizes to 128 and 256 and repeated the tests for the previous epoch values with 6 new models. Per the TA's suggestion, we also trained 4 more CNNs for a very large number of epochs (1250 and 1500) to see if we could increase validation accuracy. In this series of tests, the CNN with a batch size of 128 trained for 1000 epochs produced a validation accuracy of 89.7%. This model was the best-scoring one across all tests and was saved as "CNN_23.pt."

Fig. 2 below shows the training and validation accuracies of a CNN (with the same parameters as the saved model) as epochs increase to 1000. The validation accuracy peaked around 540 epochs and then flattens, which does not exactly match up with the model we saved. In 2 subsequent re-runs of the batch/epoch tests, models with batch size 128 trained for 1000 epochs consistently performed best. This result indicates that there is some randomness in the CNN, since training two models with the same parameters can yield slightly different validation scores.

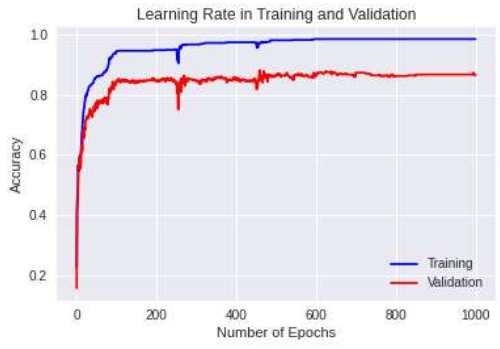


Fig. 2 Training and Validation Accuracy over 1000 Epochs

### *D. Evaluation of Experiments*

All of these experiments were done with the goal of finding the optimal values for the hyperparameters of our CNN. Our tests covered many different aspects of the model and methodically checked reasonable ranges of

values for each parameter. There were some model parameters, such as the loss function and the gradient descent optimizer, that were kept constant. This choice was made based on what we gathered from the literature. However, further testing could be done to see what effect these have on the model.

## IV. CONCLUSIONS

Our final CNN achieved 89.7% accuracy on the validation set. Below is the confusion matrix for that evaluation:

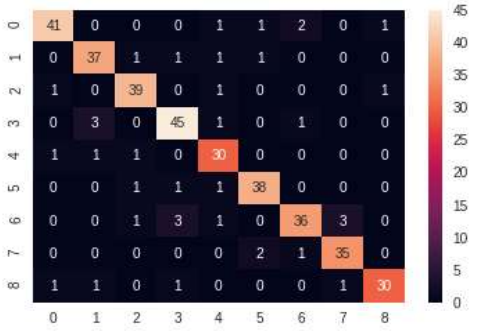


Fig. 3 Confusion matrix for final CNN on the validation set

This confusion matrix shows us that our CNN does not misclassify that many of the input letters. Additionally, we examined the per-class accuracy of our classifier to see how well it classifies each letter:

Table 1: CNN Validation Accuracy on each Letter

| Letter | Accuracy | Letter | Accuracy |
|---|---|---|---|
| A | 89.1% | F | 92.7% |
| B | 90.2% | G | 81.8% |
| C | 92.9% | H | 92.1% |
| D | 90.0% | I | 88.2% |
| E | 90.9% | | |

This table shows us that our classifier can classify six out of the nine ASL letters with at least 90% accuracy. We also see that it does particularly poorly with the letter ‘G,’ likely bringing down our overall accuracy.

After following all the steps in a standard machine learning pipeline, we trained a deep learning model to be capable of classifying the ASL letters ‘A’ through ‘I’ with 89.7% overall accuracy. With the input images preprocessed to be grayscale, we conducted a series of experiments to determine that we needed a CNN with 3 convolutional layers, 2D pooling layers, 3x3 kernels, trained in batch sizes of 128 for 1000 epochs, and updated with the Adam stochastic optimizer to minimize cross-entropy loss. Ultimately this CNN proved to be a suitable model for this image recognition task.

## REFERENCES

[1] S. Bhanja and A. Das, "Impact of Data Normalization on Deep Neural," 2019.

[2] M. D. Zeiler and R. Fergus, "Visualizing and Understanding," NYU, New York, 2014.

[3] "Creating a Convolutional Neural Network in Pytorch," [Online]. Available: https://pythonprogramming.net/convnet-model-deep-learning-neural-network-pytorch/. [Accessed 04 4 2020].

[4] C. Nwankpa, W. Ijomah, A. Gachagan and S. Marshall, "Activation Functions: Comparison of Trends in," arXiv, 2018.

[5] B. J. Lei and D. P. Kingma, "ADAM: A Method for Stochastic Optimization," arXiv, 2017.

[6] Y. Bengio, "Practical Recommendations for Gradient-Based Training of Deep," arXiv, 2012.

[7] D. Masters and C. Luschi, "Revisiting Small Batch Training for Deep Neural Networks," 2018.

[8] S. L. MNIST, "Sign Language MNIST," Kaggle, 20 10 2017. [Online]. Available: https://www.kaggle.com/datamunge/sign-language-mnist/metadata. [Accessed 20 4 2020].