Classifying American Sign Language Letters Using Machine Learning

Antonio Antonian, Mitchell Hammack, Colin Kubisiak, Simon Frank

Abstract - **Background: American Sign Language (ASL) is a natural language that serves as the predominant language for Deaf communities in American and Canada. It uses a combination of hand, face, and torso movements in order to convey language. In order to reduce the communication barriers between people who speak orally and people who use ASL, we developed a machine learning model that intakes an image of one of the first 9 letters of the ASL alphabet and returns the English letter.**

Result: The implementation uses images that are 100 by 100 pixels; this requires an initial processing of the input to be the correct size. Then it is fed into a convolutional neural network (CNN) where it propagates through three convolution and pooling layers and then one fully connected layer. The CNN's final output layer calculates the probabilities that the image is one of the 9 ASL letters. If none of the probabilities for each of the letters reach a threshold, then it decides that the image does not contain one of the 9 ASL letters.

Conclusion: This model is able to classify the first 9 letters of the ASL alphabet with very high accuracy and could be a useful tool to help reduce barriers between the deaf community and people who speak orall*y*.

I. Introduction

The Deaf community have always had difficulties communicating due to the inability to speak orally. This issue was solved by developing sign languages that used hand, face, and torso movement to speak. However, there are still communication issues if the receiving party does not know sign language. In order to alleviate this issue we have developed a machine learning model that intakes images of the ASL alphabet and returns the english letter. This eliminates the need for the receiving party to know ASL. Implementing this model into software could allow a person who has no knowledge in the ASL alphabet to read what letters a person is signing and significantly reduce communication barriers.

In order to train our model we had to first collect and process our data. The first source of data was from an in class lab where each group submitted 20 images of each of the 9 ASL letters resulting in 1,844 images. The second source of data was from a kaggle data set [1] which had all the ASL letters, but we extracted only the first 9 letters resulting in 27,000 images. In total we had collected 28,844 images of the 9 letters between our two sources. Next, we processed our data by reducing each of the images down to 100 by 100 pixels and centering the hand in the center. Our images are unsigned 8-bit and 3 channel, meaning each pixel can hold values between 0 and 255 and there are three channels for one for red, green, and blue. However after we imported the data, we would always min-max normalize the data by dividing each pixel by 255 to make all the pixels have a value between 0 and 1.

In addition to the data that was used to train the model for the 9 letters, we also included data that was not any of the targets. Adding this additional data would hopefully allow the model to detect any anomalies. We added 1490 random images from the CIFAR-10 [2] data set and 510 images of the other ASL letters. Again we preprocessed the ASL letters and CIFAR-10 images to be in the correct format of 100 by 100 unsigned 8-bit pixels. However, one difference is that usually we decrease the size of an image to meet the 100 by 100 pixel criteria, but for the CIFAR-10 data we had to increase the size of it since it was initially 32 by 32 pixels. Since it is impossible to recover the initial pixels, we used interpolation to increase the size of the image. We feel that interpolating these images had no impact since the images were so distinct from the ASL characters and the images still represented their original picture. In total we had 2,000 images that all were there to help detect if an image did not contain any of the first 9 letters.

With the pictures in our data set, we felt that it had significant variance due to differences in backgrounds, lighting, coloration, and the hand position. Additionally, there was an equal distribution of each of the 9 letters. So with approximately 3,205 images of each ASL letter that had enough variability, we had a data set that could accurately train a model to learn how to classify the images, but also generalize to new data. We would use a training and testing split in order to check that our model was learning enough to achieve high accuracy, but also be able to generalize on new, unseen data.

II. Implementation

For our model, one of the reasons we choose to use a convolutional neural network (CNN) was due to it being shift invariant. This means that if the ASL letter is in a different part of the image, the CNN can still identify the letter. Therefore, giving it the ability to extract visual features of images on its own without the need of feature engineering. Another reason why the CNN was chosen was due to its high accuracy scores and ability to outcompete other machine learning models. We developed other models and compared their accuracies, but CNN performed the best.

Our model's architecture consists of a 64 filter input convolutional layer that goes into two consecutive 128 filter convolutional layers both with max pooling. For the last convolutional layer, we applied dropout with a rate of 0.5 before the data is flattened. After the data is flattened, we then feed the outputs of the convolutional layers into a fully connected layer with 128 neurons with a dropout rate of 0.5 before we finally feed it into the output layer.

We used batch learning with a batch size of 64, Adam as the optimizer, and early stopping with a patience of 30 epochs. This means that the model will stop learning if the validation loss doesn't improve after 30 epochs. We also used the ModelCheckpoint module within keras to save the state of the model with the best validation accuracy while training.

We determined the optimal architecture size, batch size, and optimizer, by performing a rough estimation of a grid search. We would manually add or remove layers, increase or decrease the size of layers and batch size, and choose between optimizers. We first

determined the optimizer by training the data on multiple models with each a different optimizer and seeing which performed best. In the end, Adam almost always performed the best and hence is why we chose it for our final optimizer. Next we experimented with the number of layers by adding a layer and seeing if the model performed better. If it performed better, we would keep the layer and add another, but if it performed worse then we would remove that new layer. We would repeat this process until we found the optimal number of layers. We optimized the size of the layers by experimenting with different combinations of layer sizes and then seeing which performed the best. We could have done a true grid search to tune the hyper parameters, but since the model's accuracy was so high, we did not feel it was necessary.

```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_30 (Conv2D)           (None, 100, 100, 64)      832
_________________________________________________________________
conv2d_31 (Conv2D)           (None, 98, 98, 128)       73856
_________________________________________________________________
max_pooling2d_20 (MaxPooling (None, 49, 49, 128)       0
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 47, 47, 128)       147584
_________________________________________________________________
max_pooling2d_21 (MaxPooling (None, 23, 23, 128)       0
_________________________________________________________________
dropout_20 (Dropout)         (None, 23, 23, 128)       0
_________________________________________________________________
flatten_10 (Flatten)         (None, 67712)             0
_________________________________________________________________
dense_20 (Dense)             (None, 128)               8667264
_________________________________________________________________
dropout_21 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_21 (Dense)             (None, 10)                1290
=================================================================
Total params: 8,890,826
Trainable params: 8,890,826
Non-trainable params: 0
_________________________________________________________________
```

Figure 2.1: A sequential list of the architecture of our model with a summary of the number of parameters.

To identify characters not in our set, we trained our model with negatives that consist of other characters not in our set as well as other random images. All of these negatives were associated with an unknown class label of -1.

III. Experiments

The experimentation can be broken down into 3 types: data preprocessing, model selection, and hyperparameter tuning. Experimental performance for each type consisted of measuring model performance in terms of a confusion matrix and test accuracy.

Data preprocessing consisted of either applying Linear Discriminant Analysis (LDA) or Principal Component Analysis (PCA) to the data as a means of reducing dimensionality and feature consolidation. The number of features and components for dimensionality reduction was chosen based off of the data keeping approximately 95% of its variability. The other preprocessing step was choosing to have the data be grayscale vs color.

The model selection experimentation involved running a number of different machine learning models against the different types of data and recording the model performance via accuracy and a confusion matrix. The models that we chose to test were, K-Nearest Neighbors (KNN), Decision Tree (DT), Random Forest (RF), Support Vector Machines (SVM), Multilayer Perceptron (MLP), and Convolutional Neural Networks (CNN).

For hyperparameter tuning a grid search method was used. The model type, the hyperparameters that were tuned, and the values that were tuned are listed below.

KNN

Parameter tuned | values used

- n_neighbors | [1,2,3,4]
- p | [1,2]
- weights | [uniform, distance]

Decision Tree

- criterion | [gini, entropy]
- max_depth | [None, 15, 25, 20]
- min_samples_leaf | [1, 2, 3]

Random Forest

- criterion | [gini, entropy]
- max_depth | [None, 15, 25, 20]
- min_samples_leaf | [1, 2, 3]

Support Vector Machines

- C | [0.001, 0.01, 0.05, 0.1]
- gamma | [0.0001, 0.0005, 0.001]
- kernel | ['linear', 'poly', 'rbf', 'sigmoid']

Multilayer Perceptron

- hidden_layer_sizes | [(100, 100, 50), (150, 100,50), (50, 50), (150,150, 150), (100, 100, 100, 50)]
- learning_rate_init | [0.001, 0.01, 0.05]

Convolutional Neural Network

- batch_size | [16, 32, 64]
- optimizer | [sgd, nesterov, adam]
- hidden_layer_sizes | [1,2,3]
- Convolutional_layers_sizes | [1,2,3,4]

IV. Conclusion

This table shows the accuracy from each model run and data combination. The model hyperparameters are the ones that performed the best in the grid search.

	KNN	DT	RF	SVM	MLP	CNN
g_pca	24%	12%	21%	20%	11%	
g_lda	54%	57%	57%	64%	62%	
c_org						99%

The Convolutional Neural Network had the best model performance with 99 percent accuracy on the test set. Below is the confusion matrix for the CNN model.

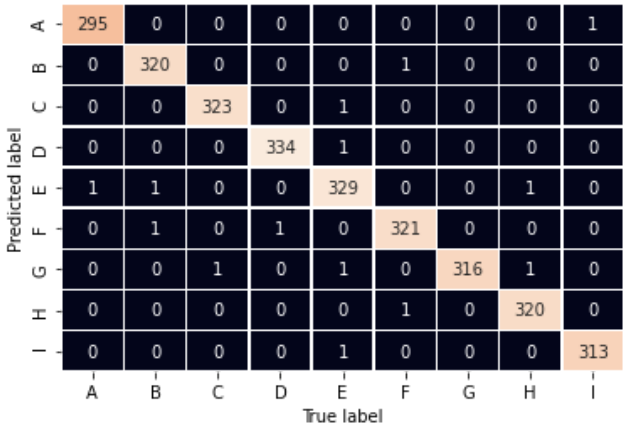

Figure 3.1: The confusion matrix of the model when there are no anomalies to be detected

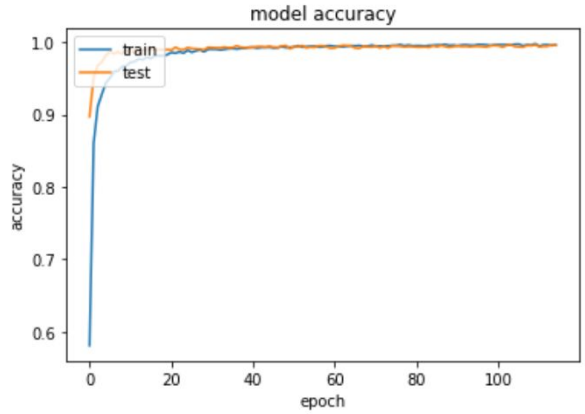

Figure 3.2: Model accuracy curve without unknowns

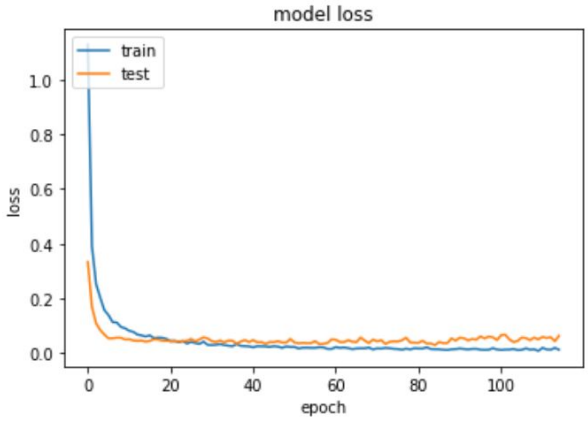

Figure 3.3: Model loss curve without unknowns

When the CNN did not have to detect anomalies, the CNN scarcely missed any predictions and the ones it did miss were spread out. This shows that the model generalizes well to the test data. From the table we can see that no other model or experiment came close to the level of accuracy of the CNN. This result is to be somewhat expected as it is well known that CNN model architectures perform well on image data given that they take into consideration the spatiality of images. However, when the model had to detect if the image had one of the letters in it at all, then it performed a little worse.

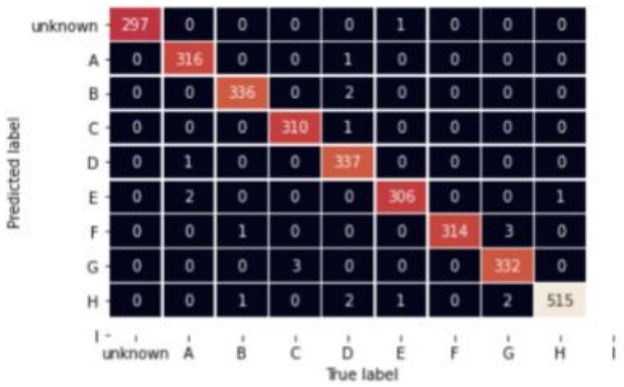

Figure 3.3: Confusion matrix of the model when there anomalies to be detected

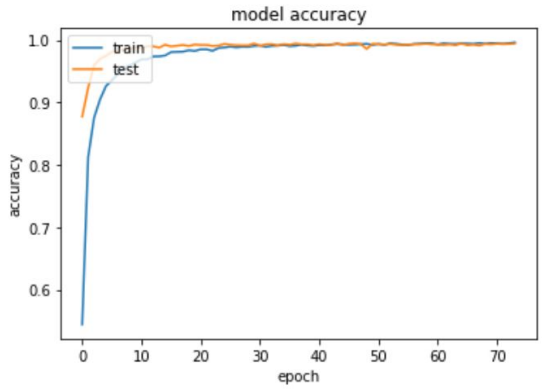

Figure 3.4: Model accuracy curve with unknowns

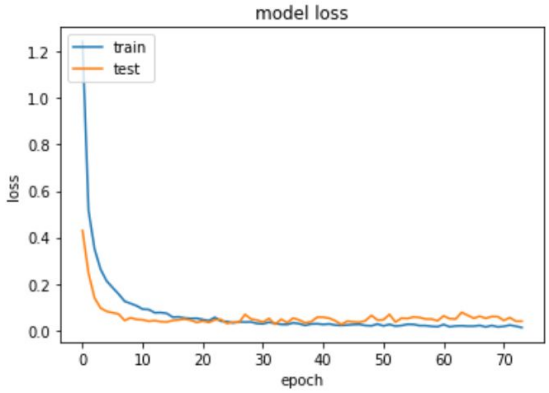

Figure 3.5: Model loss curve with unknowns

It can also be taken away that PCA when applied to the grayscale images had significantly worse performance as compared to LDA. LDA was more suited for the images and models in this project and produced better results. The main takeaway from this project being that the CNN was vastly superior to any other model for this dataset and task.

V. References

[1] ASL Alphabet, https://www.kaggle.com/grassknoted/asl-alphabet
[2] CIFAR-10 dataset, https://www.cs.toronto.edu/~kriz/cifar.html