ASL Recognition

CS144

Edbel Basaldua
Jack Baskin Engineering
University of California
Santa Cruz
ebasaldu@ucsc.edu

Jihwan Seo
Jack Baskin Engineering
University of California
Santa Cruz
jseo11@ucsc.edu

Wen Bin Yu
Jack Baskin Engineering
University of California
Santa Cruz
wyu19@ucsc.edu

ABSTRACT

For this project we aimed to classify ASL(American Sign Language) letters including two other hand signs and a blank image. In our experiments we developed two working models, one utilizing transfer learning by importing the VGG network and the other implementing a standard CNN. The main part of this project was to see how we could translate what was learned in class and apply it to a real classification problem. We showed how good of a CNN we built using our current knowledge. We mainly presented the transition and growth of the model from the project's initial stages through its final stages. This included making adjustments to variables such as changing the complexity of the network, regularizing, measuring against overfitting and data augmenting. Additionally we also documented our exploration with transfer learning using the VGG16 pretrained network and made comparisons to our standalone Convolutional Neural Network.

OBJECTIVE

In this project our objectives were as follows: first we wanted to observe how good of a network we could build given the time constraint. In addition to this, we wanted to see how each variable of a Neural Network affected the performance and this was done through the use of standard metrics and methods. Lastly we wanted to attempt implementing a NN that utilized transfer learning. Smaller goals in this project were mainly to rack up experience with the relevant ML libraries in python.

In regards to the overall approach to the classification problem at hand, we would say that we had a methodical way of changing the structure of the neural network. Most of the adjustments were made based on what was presented in class and what research was done. The implementation wasn't too different from what is typically seen in CNN architectures. The only novel approach we had was probably related to the increase in filters and the placement of dropout in the model. We also improved the performance of the standalone CNN based on the tests from the VGG16 neural network.

DATASET

Data Preparation:

Before the training phase we didn't have to clean the data at all. We made use of two sets of images to help classify the letters and other characters. One was to be used for the training/validation sets and the other was meant to be used for testing batches. Everything in each of the datasets was indispensable. Overall each image was unique to one sign and not dependent on the others. When we first pulled the dataset from Kaggle, we were deciding between removing the extra characters and leaving only the main ASL signs. But since we wanted to make the classification problem a bit more difficult and keep the integrity of the dataset, we decided to leave them in.

Initially we started our project on Colab but our training dataset was too large and we couldn't train the model because the training was indefinitely long. We then tried to reduce our dataset size by training on 1,000 images per character but still ran into training issues. But after further research we found that Kaggle provided an interactive kernel to train models. Kaggle's kernel gave access to far more computing power and we were able to train the entire dataset. By doing so we were able to preserve the integrity of the dataset and not make any changes that would alter any true results. The entire data set consisted of 87,000 images(3,000 images per character).

When we partitioned the data we decided to stick with a 80/20 split, training set and validation set respectively. We felt that since each character had only 3000 instances available we needed a fair amount of character instances to train the model effectively. In order to do that we needed to have a partition where the training set held most of the data.

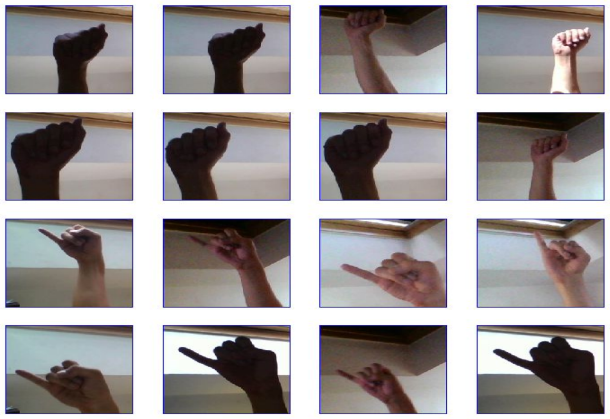


Figure 1.1 *Random Instances of Signs* ***(Letters A and J)***

We didn't think extensive data augmentation would be necessary because when manually looking at the images, there was enough variability in lighting, angles and proximity to make each instance of the training images different (**Figure 1.1**). Our dataset was fairly large and it already included images shifted up, down, left and right. Adding any other form of augmentation may have led to redundancy in the data and produce inaccurate results or lead to major overfitting. In other words it seemed as if the dataset had already been previously augmented in some fashion so it seemed like overkill. Most importantly, the problem itself favors our decision because we want the model to learn the **features** of a certain person's hand alignment, which is **uniquely mapped** to one of the characters. If we distorted or augmented the hand signs from the way they were intended to look, we would have a higher margin of error which would translate into the results. Adding more training data with data augmentation would have made training time longer and we were working with limited computational resources. We felt that all of these choices were in the best interest of minimizing the risk and mitigating the issues that were commonly seen in models.

Test Data Set:
For our test data we took another dataset from Kaggle. This dataset only contained 30 images from each class. We didn't do any data preparation for this dataset as it was already well prepared. We simply made a generator to feed into our model. The difference we noticed from this dataset was that the backgrounds of the pictures varied unlike the original dataset which was mainly taken in a person's room. Our results for the test set held up with some volatility that may be due to the changing backgrounds. This training dataset even had images of the persons face in it. Another difference is that the original dataset contained images that were very dark, making it difficult to detect features.

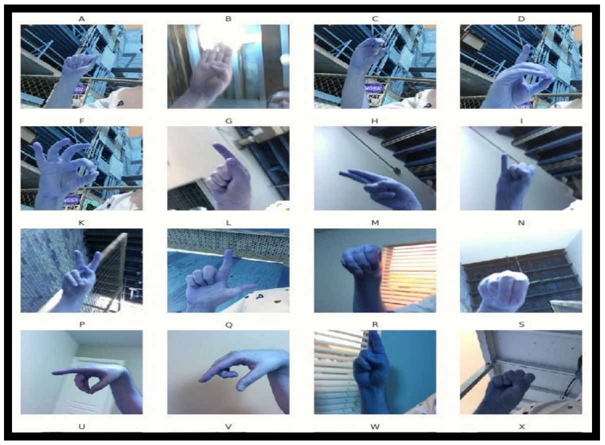


Figure 1.2 *Random instances of testing set images*

MODEL AND ALGORITHMS

In the following sections we present, the challenges, the two models and the transition between each project phase, the interworking architecture and the rationale for some of the changes we made to arrive at our final model. This project was within our scope of completion during the academic quarter. Although we wasted a week by changing our project, we were able to complete this project within the allotted 7 weeks. We also built two models instead of one and spent hours in trial and error with the architecture of the layers, using different dropout rates, pooling windows, sizes of filters, and different optimizers. We also faced challenges such as finding a reliable resource that would give us free GPU.

MODEL CHALLENGES AND DECISIONS

One of the major issues that we ran into when developing the model was a lack of proper resources. In order to work collaboratively we opted to use a cloud based platform, rather than running it locally on a machine and having to keep an external repository updated. Based on previous experience, we made the decision to initially go with Google's **Colab** platform but we ran into a lot of issues. We were only given a set amount of computational capacity which was far less than what we needed, in order to perform the training experiments. Additionally we stopped hosting our dataset on Google drive because it continuously corrupted the images in the datasets. We had a lot trouble settling and finding a platform that wasn't going cost too much and offered optimal performance. Eventually we found out that Kaggle's free platform gave a fair amount of CPU and GPU capacity to conduct our training experiments.

Model Infrastructure:

Across each stage of the project, we developed a few versions of the Convolutional Network to recognize the American Sign Language characters. We made use of several tools and techniques that are typically used to improve the performance of a neural network. This is explained in the following sections.

	Initial Model	***Transfer Learning Model***	***Final Model***
Convolutional layers	3	6	7
Pooling Layers	3	6	7
Hidden layers	1	1	1
Dropout	0	1	1
Optimizer	RMSProp	Adam	Adam
Loss Metric	Sparse Cross entropy	Categorical Cross entropy	Categorical Cross entropy
Learning rate	Default (0.001)	Default (0.001)	0.00099
Testing Acc and Loss	Accuracy: 0.54 - 0.58 Loss: 1.4 - 1.6	Accuracy: 0.60 - 0.80 Loss: 0.75 - 1.75	Accuracy: 0.7 - 0.85 Loss: 0.8 - 1.6
Training Acc and Loss	Accuracy: 0.9344 Loss: 0.1662	Accuracy: 0.95 - 0.99 Loss: 0.1 - 0.2	Accuracy: 0.95 - 0.99 Loss: 0.1 - 0.2

Figure 1.2 *Model Progression*

Initial Model:

The first iteration of our model was very bare bones, nothing complex. Our input was a 200x200 image with 3 channels for RGB values. We had a stack of three convolutional layers with only one pooling layer. The first convolutional layer had four 4x4, the second had eight 3x3 and the last one had sixteen 3x3 filters. The max pooling layer was set between the 1st and 2nd convolution layers and consisted of a 2x2 window. After flattening the last convolution layer, the Neural Network had no hidden layers. The last layer was a Softmax layer with 29 outputs for each of the images to be classified. This initial model and the series of changes made, served as a means of building up the core CNN layers. After this iteration we evaluated the results and made various versions just trying different stacks of Convolutional layers and addition of dense layers.

Intermediate Model Findings and Conclusions:

Towards the mid stages of the model progression we found that a single hidden layer was satisfactory in providing adequate performance. It also mitigated developing a CNN with an absurd amount of parameters. Through our experiments we also noticed that increasing the number of filters across convolutions by some factor greatly improved performance. In addition that in order to give this model a wide "**field of vision**" we needed to keep the window size a decent size, so that features would be recognized. We also discovered that nesting a pooling layer between these convolution layers helped appropriate downsampling a lot better.

Transfer learning:

In one of our models, we decided to make a model based on a pretrained network and implement "transfer learning". During our initial research we came across transfer learning and noted that it was a popular way of using a pretrained model to solve another problem. With our TA's recommendation, we gave VGG16 a try. We decided to implement this by taking the weights and altering the softmax layer to 29 classes instead of 1000. We used 8 layers including the input, hidden, and softmax layers. This model produced results that weren't as great as our other model. It performed slightly more poorly and it took many epochs to get there. This model took 40 epochs to reach around 70% validation accuracy compared to the 85% validation accuracy in 23 epochs in our final model. Sometimes this model was inconsistent as it had a consistent 60% testing accuracy and 65-80% testing accuracy on different runs. Given more time we would have liked to see if we could optimize this transfer learning model even further.

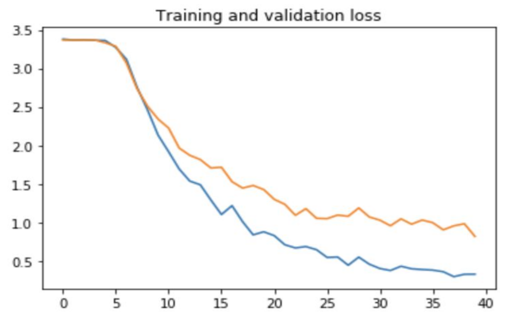


Figure 2.1(VGG Model) *Training and validation loss* ***(Blue for training and orange for validation)***

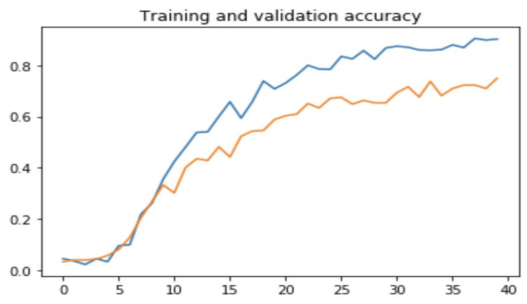
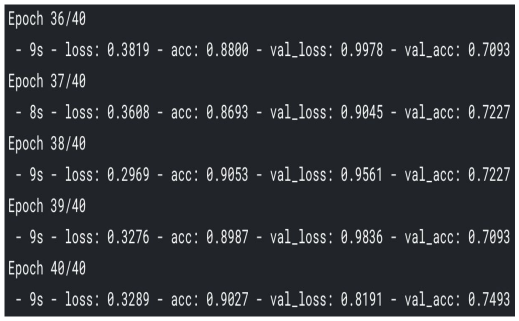


Figure 2.2(VGG Model) *Training and validation accuracy* ***(Blue for training and orange for validation)***



```
Epoch 36/40
 - 9s - loss: 0.3819 - acc: 0.8800 - val_loss: 0.9978 - val_acc: 0.7093
Epoch 37/40
 - 8s - loss: 0.3608 - acc: 0.8693 - val_loss: 0.9045 - val_acc: 0.7227
Epoch 38/40
 - 9s - loss: 0.2969 - acc: 0.9053 - val_loss: 0.9561 - val_acc: 0.7227
Epoch 39/40
 - 9s - loss: 0.3276 - acc: 0.8987 - val_loss: 0.9836 - val_acc: 0.7093
Epoch 40/40
 - 9s - loss: 0.3289 - acc: 0.9027 - val_loss: 0.8191 - val_acc: 0.7493
```

Figure 2.3 (VGG Model) *Epoch Snapshot*

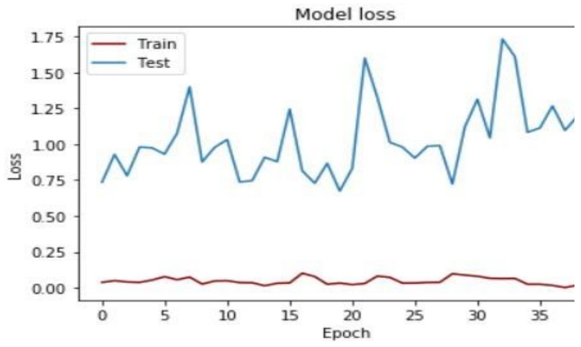


Figure 2.4: (*Training and test loss for test set*)

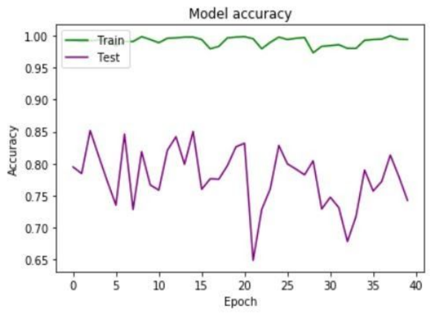


Figure 2.5: (*Training and accuracy for test set*)

FINAL MODEL

After running a series of experiments and running into a few obstacles, we managed to come up with a CNN that used 7 layers of convolution and 7 layers of max pooling. In each of the Convolutions the window size was kept at **(3x3)**. We felt this was appropriate because each ASL character has a unique gesture that is comprised of intricate finger placement and angling. This was also done because the dataset has variability in proximity, lighting and rotation. In each of the layers the filter number is increased by a factor of (**filter size*2) ,** this was arbitrary but was done due to the better performance of increasing filter number. The pooling layers were all set up to downscaling by a factor of 2. This was done mainly to prevent against overfitting, which is something our models had trouble with. In addition it helped reduce the number of parameters. The strides in the pooling layers were set to the default to the size of the pooling. In regard to the activation functions we used the **ReLu,** mostly due to its conventional usage, but of course we understood that it works well because of its maximal properties and to prevent our gradient from vanishing. Flattening was also done by convention to create a 1D input vector. The final model only relied on a single hidden layer, this was set to the same size at the flattening which was done due to it limiting the number of parameters. We later understood that this was almost similar to having the flattening layer twice but the hidden was slightly different. This dense layer was removed temporarily to see if it actually changes the performance of the model. In our findings we determined that inclusion of a hidden layer did in fact improve the accuracy.

The optimizer chosen was Adam ,since it had an adaptive property and allowed for implementation of momentum. The learning rate using Adam was changed multiple times, it was kept between **(0.0001- 0.001).** Through the testing we found that a very small learning rate was not very good for the CNN. It took a lot longer to converge and needed to be ran for a large amount of epochs. Conversely using a bigger learning rate allowed the model training to completely overstep and never find the a true minimum but took less epochs to converge. After some trial and error the best performance came from a learning rate of **(0.000999).** There probably is a better learning rate but we only changed it by a small factor each time we tested. Our time was limited so we needed a way to cover a wide range of values. In regard to epochs we started out with a base of **15** and worked up to **40**. This was done due to the assessment of convergence on each increase to the number of epochs. Our learning rate performed the best with this level of epochs. The last measure we took to better performance was the integration of dropout into the final model. Dropout values were set at a **Min of 20%** and a **Max of 60%.** Through testing we found that a dropout rate of **55%** yielded great results from what was previously observed. It reduced the training error and helped counter the overfitting a bit.

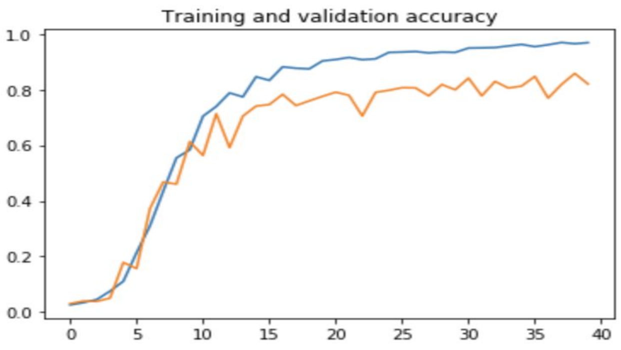


Figure 3.1(Final Model) *Training and validation accuracy* ***(Blue for training and orange for validation)***

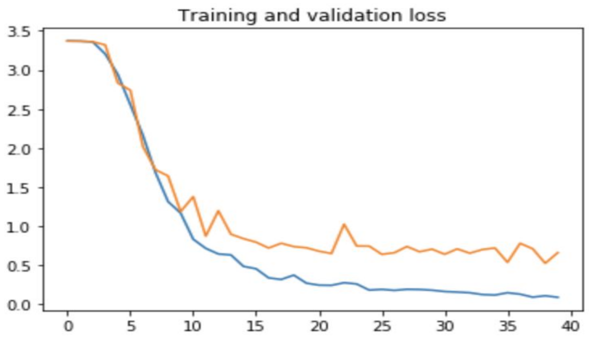
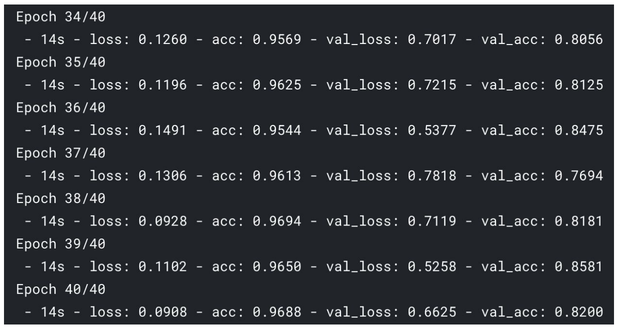


Figure 3.2(Final Model) *Training and validation loss* ***(Blue for training and orange for validation)***



```
Epoch 34/40
 - 14s - loss: 0.1260 - acc: 0.9569 - val_loss: 0.7017 - val_acc: 0.8056
Epoch 35/40
 - 14s - loss: 0.1196 - acc: 0.9625 - val_loss: 0.7215 - val_acc: 0.8125
Epoch 36/40
 - 14s - loss: 0.1491 - acc: 0.9544 - val_loss: 0.5377 - val_acc: 0.8475
Epoch 37/40
 - 14s - loss: 0.1306 - acc: 0.9613 - val_loss: 0.7818 - val_acc: 0.7694
Epoch 38/40
 - 14s - loss: 0.0928 - acc: 0.9694 - val_loss: 0.7119 - val_acc: 0.8181
Epoch 39/40
 - 14s - loss: 0.1102 - acc: 0.9650 - val_loss: 0.5258 - val_acc: 0.8581
Epoch 40/40
 - 14s - loss: 0.0908 - acc: 0.9688 - val_loss: 0.6625 - val_acc: 0.8200
```

Figure 3.3 (Final Model) *Epoch Snapshot*

RESULTS AND ANALYSIS

In the training phase of the project we used two metrics in order to determine how well each iteration of our model was doing. We compared the accuracy between the training and the validation sets and also calculated our loss to see how well it performed. In order to get an unbiased confirmation of our results, we tested the model against another dataset on Kaggle and produced our test results. Our model didn't perform as well as the training set but that was expected. The new dataset brought in new variables (mentioned in the dataset section) and allowed us to test for overfitting. We slightly overfitted but performed well.

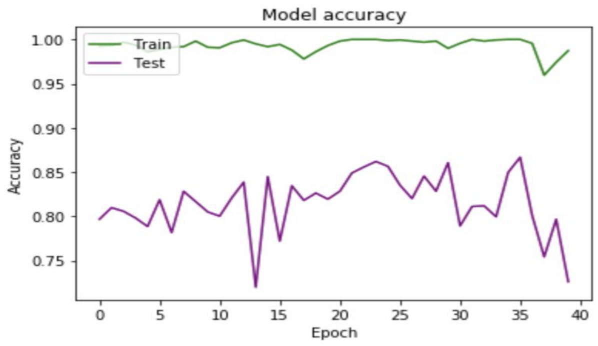


Figure 3.4: (*Training and test accuracy for test set*)

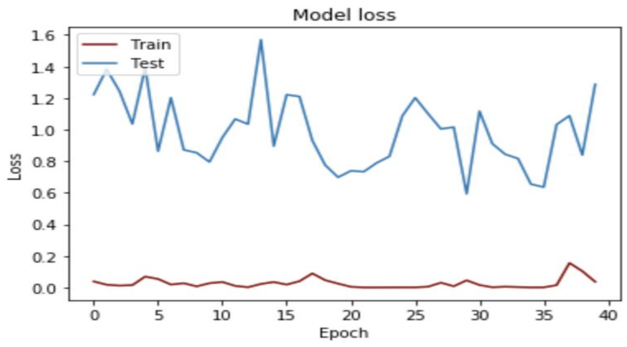


Figure 3.5: (*Training and loss for test set*)

CONTRIBUTION

Wen Bin Yu: Created and tested the VGG16 transfer learning model. Tested and tried different architectures for the model, such as adjusting learning rate, dropout, epochs, batch sizes, etc. Incorporated the extra dataset for testing. Helped with the poster design. Contributed to all deliverables.

Jihwan Seo: Set up the initial project on Google Colab and modeled the project off of our third homework assignment. Colab didn't have enough computation power so wrote a script to parse the dataset. We eventually ended up moving to the Kaggle Kernel and was able to train on the whole dataset. Experimented with different parameters on both training models but ended up choosing Edbel's version of the CNN model. Helped with the poster design. Made sure we were on track with deliverables. Helped with the visualization of the models performance. Created the visualization of sub sampling across the layers.

Edbel Basaldua: Created and tested the Final CNN model, implemented various versions of the initial model, tweaked the models by adjusting values for learning rate, dropout, epochs, convolutions, etc. Helped with the poster design and outline of the Final report. Created some of the visualization of the models performance. Found and used an open source script to Provide a 3D visualization of the final model. Contributed to all deliverables.

PROJECT EXPANSION/FINAL NOTES

We had a bit of difficulty starting out and changed our project late, mostly due to a lack of useful data for a different application of Machine Learning. As a result of the loss of time, we were a bit behind so our project is not as accomplished as it could have been.

If allotted more time we would have liked to take this CNN to the next level and recognize live ASL footage to predict and translate conversations. A video is just a sequence of many pictures and a CNN is completely capable of achieving that. This application seems far more useful for real world issues rather than feeding images and predicting characters. We could translate ASL for people who don't know how to sign and can allow people to communicate back and forth. We learned the basics of neural networks and deep learning, and if we continue pursuing our studies in this field this would be achievable. I think a project of this nature would be very practical in the field of Artificial Intelligence and could help provide social good.

REFERENCES

- Rafael Espericueta CMPS 144 Assignment 3 Winter 2019, Professor Narges Nozouri
- https://www.kaggle.com/grassknoted/asl-alphabet
- https://www.kaggle.com/danrasband/asl-alphabet-test