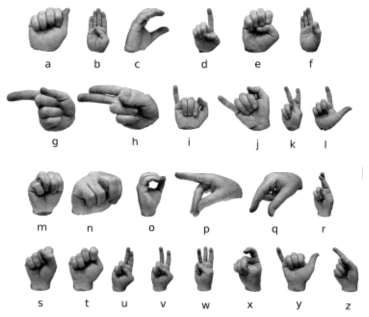


ASL Sign Classification

Machine Learning Group 2 Final Report

Roger Wang wangbrh@bc.edu
Jay Agrawal agrawalj@bc.edu
Donghyuk Kim kimbyi@bc.edu
Azim Keshwani keshwani@bc.edu
Matthew Kim kimcei@bc.edu

1. Abstract

ASL recognition is highly variable and complex, and required multiple steps of learning. Thus, a convolutional neural network was an obvious choice of model. With the help of Pytorch, the process of constructing a neural network with the necessary parameters was greatly alleviated. The training dataset was predicted to be complicated, and widely varied. The results of the model and probabilities of outputs were projected on a confusion matrix. Loss was calculated and minimized during the training stage using a cross-entropy optimizer, then once each probability output was calculated, we ran Pytorch's stochastic gradient descent optimizer based purely on speed and optimization. However, the similarity of some of the hand signs was too great and the dataset unexpectedly too diverse and large, thus the model succumbed to overfitting during the testing stage. The bias and variance tradeoff was apparent, though underestimated due to too much reliance on libraries and less time on different hyperparameters. The big takeaway from the project is that learning sign language is very feasible and has numerous applications. The biggest challenge would be accomodating for the diverse dataset.

2. Introduction

We used a convolutional neural network to build our ASL image classification model. Convolutional neural network is a subclass of a neural network that has one or more hidden layers called convolutional layers. Convolutional layers, like any other hidden layers, receive input, transform it, and output it to the next layer. The transformation that occurs in a convolutional layer is called a convolution. Each convolutional layer has a specified number of filters, also known as kernels, which are small matrices (e.g 3x3) that are initially filled with random numbers. When the convolutional layer receives an input image, represented by a matrix of pixel values, the filter slides over each 3x3 block of pixels from the entire image. The filter performs dot product with each of the 3x3 pixels, and stores the scalar result into a new matrix known as an activation map. Once the filter convolves entire image, we are left with an activation map that is smaller than the input matrix. For instance, suppose the input matrix was 7x7 and the filter size was 3x3. The resulting activation map be 5x5 since a 3x3 filter can fit in 25 different regions of a 7x7 matrix.

Convolutional neural networks may also have a pooling layer, which downsamples the input by reducing the dimensions. One example of a downsampling technique is max pooling, which subsamples input by picking maximum value within the subblock. Downsampling has several advantages. One, the outputs become invariant to small translations in the input and overfitting is prevented. Two, reduction in dimensions improve computational efficiency.

According to research conducted by Cadence Design Systems, convolutional neural networks are typically used for Image Recognition style problems. CNN has several advantages.

First, CNN tend to be more rugged to shifts and distortions in the image as the same weight configuration is used for the entire space of the image. Second, a convolutional layer has fewer memory requirements as the same coefficients can be used throughout different locations of the image space. Furthermore, CNN is faster with training data as it has a few number of parameters (Hijazi).

We used libraries from PyTorch, a Python-based scientific computing package that uses GPU power, to construct our convolutional neural networks. *Batch normalization* was performed after each convolution by normalizing our weight to prevent overfitting. Following our batchnorm layer, we performed downsampling using *maxpooling* to remove the unimportant pixels of the input image by taking the max value in (2, 2) kernels across the image. The *stride* of 1 shows exactly how much we shift through each image pixel in order to develop our convolutional layer. For optimizer, we used the cross entropy loss, also known as a log-loss, which measures the performance of the model with a probability value between 0 and 1.

3. Method

Our approach for implementing our ASL Letter Recognition classifier is to use a convolutional neural network (CNN). A comparable machine learning technique would be using support vector machines (SVM), which would require performing feature engineering and traditional computer vision methods which we feel will be less dependable and interesting than a deep neural network model (DNN). We understand that DNNs are easily prone to overfitting, we need to be careful and make sure to use regularization techniques.

Dataset

We used the ASL Character dataset from Kaggle, an online platform for machine learning and datasets. The dataset contains 87000 images with 29 classes of equal sizes. 26 classes for each letter, 3 for space, delete, and nothing (space indicates a new word and nothing indicates completion).

To follow proper data snooping protocol, we immediately perform an 80/20 train test split on the dataset and set the test set aside for evaluation to prevent any bias from it to influence how we construct our model.

Preprocessing

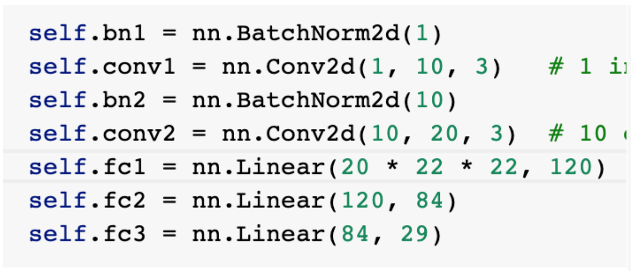
The dataset images are in color and of high dimensions which we resized to 48x48 and changed to grayscale. The data folder contains 29 subclasses, one for each class, each containing 3000 image samples.

We used a Pytorch Dataset and Dataloader to perform our train test split and feed the images into our network. Initially, we loaded all 87000 images into a tensor and crashed our instance because it used more than the 16GB of GPU ram that was allocated to us on Google Colaboratory. Our solution was to store and track the indices of the images and then read them in batches when we train. Storing the indices also made it easy for us to split the data into train/test sets with PyTorch's *SubsetRandomSampler* function.

Network

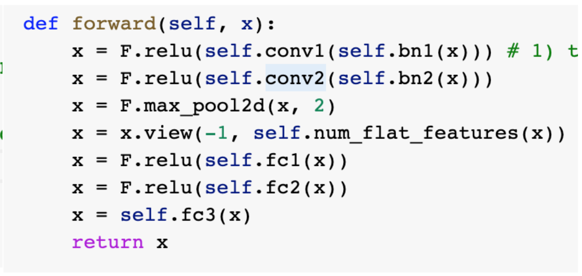
Note: We did not perform extensive cross validation testing on all of the hyperparameters we defined in the next two sections. Instead, we used standard hyperparameters used in practice according to online resources including PyTorch's documentation in the interest of time.

We initially constructed a network with one convolutional layer and one fully connected layer to ensure that we can train a model without errors. We then moved up to 4 sets of convolutional layers and fully connected layers. Each convolutional layer takes in a number of channels and outputs 10 additional channels. For example, our first two convolutional layers have parameters of (1, 10, 3) and (10, 20, 3). We fixed our kernel size to 3x3 with a stride of 1. Our first linear layer takes an input size of all of the weights outputted from the last convolution and proceeds to reduce in size to 29 nodes as it propagates through each linear layer.



```
self.bn1 = nn.BatchNorm2d(1)
self.conv1 = nn.Conv2d(1, 10, 3)   # 1 i
self.bn2 = nn.BatchNorm2d(10)
self.conv2 = nn.Conv2d(10, 20, 3)  # 10
self.fc1 = nn.Linear(20 * 22 * 22, 120)
self.fc2 = nn.Linear(120, 84)
self.fc3 = nn.Linear(84, 29)
```

Fig. 1. Network Layers



```
def forward(self, x):
    x = F.relu(self.conv1(self.bn1(x))) # 1) t
    x = F.relu(self.conv2(self.bn2(x)))
    x = F.max_pool2d(x, 2)
    x = x.view(-1, self.num_flat_features(x))
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

Fig. 2. Forward Propagation

After the second convolutional layer, we performed max pooling and downsampled our channels. Further, due to unsatisfying out of sample error (See Evaluation), we then batch normalized the values in each convolutional layer to prevent the weights from becoming too large and hopefully regularize our model.

Training

We used the cross-entropy loss function because we are outputting the probabilities of each class. For the optimizer, we used Pytorch's stochastic gradient descent optimizer because of its performance and speed compared to other optimizers. In our optimizer, we specified a learning rate of 0.001, momentum of 0.9, and weight_decay of 0.0001. With these hyperparameters, our model is able to converge and is regularized. We then trained our model with images in batches of 64 and a limit of 10 epochs (passes through the entire dataset).

Evaluation

We have the running loss of the model over the full 10 epochs of training (Fig. 3). We stopped the training loop in the 4th epoch when the loss was below 0.01. Because the problem our model is trying to solve is not high impact with consequences such as a model that classifies whether a brain tumor is benign vs malignant, penalize false positives and false negatives unequally.

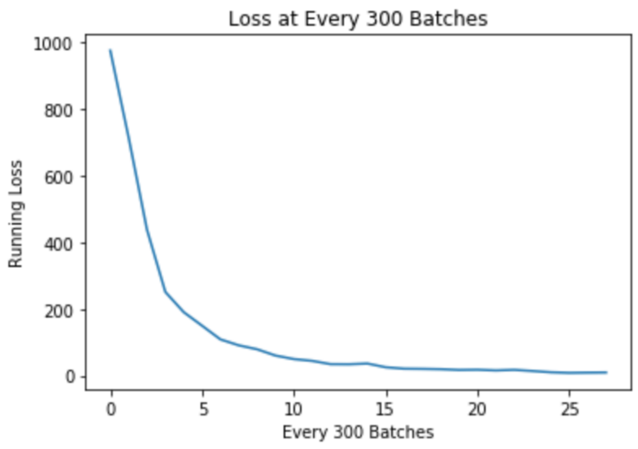
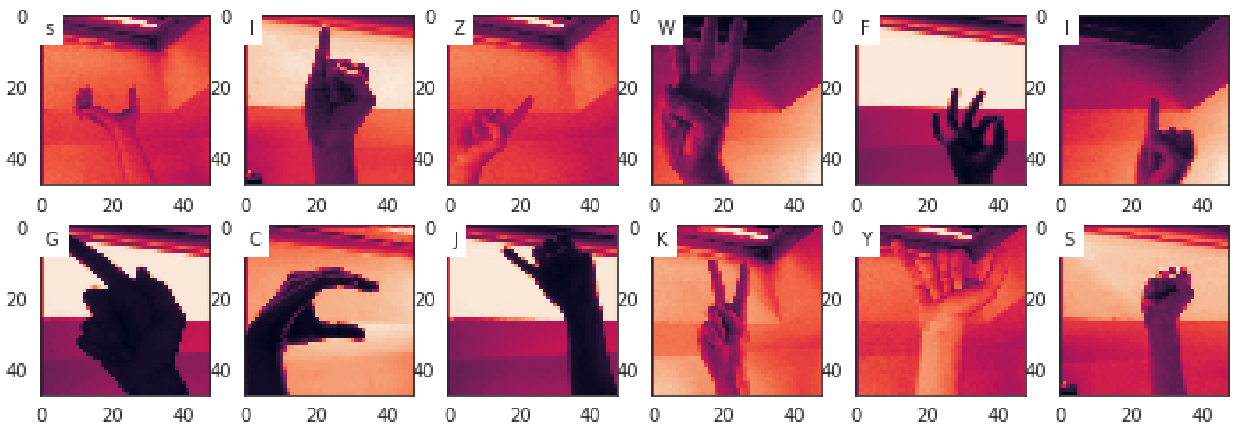


Fig. 3. Training Loss

Our overall mean accuracy in the test set we had set aside is 0.875. Initially, we believed this to be a relatively good score because it is the out of sample error, indicating that we have a good model. The following confusion matrix in Fig. 4 shows the by class predictions for each class. If we had more time, we would dive deeper and find the bias and variance tradeoffs of the model and run several models with different hyperparameters. In the following results section, we will outline how we are wrong about the out of sample error.

4. Results



Here we have a sample of some of the input data, each image includes their correct label in the top left hand corner. Each image is portrayed in a 48 by 48 grayscale.

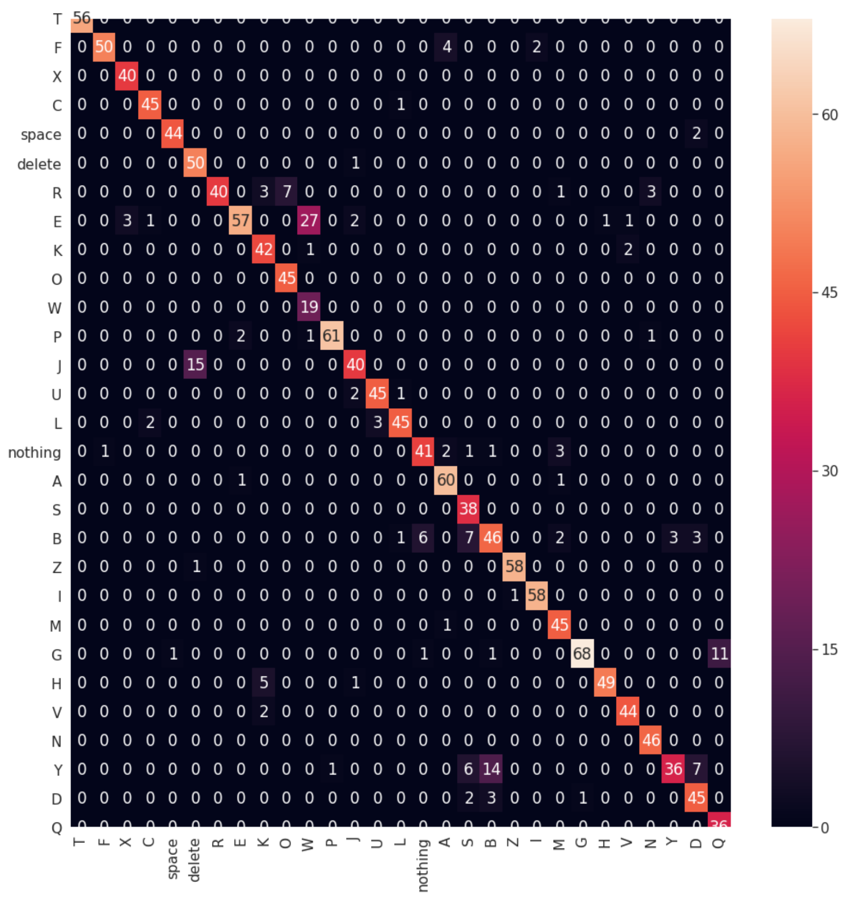


Fig. 4. Actual Label (Y-axis) vs Prediction (X-axis)

Here we have a confusion matrix which portrays the number of times a given ASL letter was predicted correctly. For example, in the top left hand corner the letter T was predicted correctly all 56 times it was tested which means it was corrected perfectly. On the other hand, the letter J was incorrectly predicted as ‘delete’ 15 times and only predicted corrected 40 times. We have a clear diagonal for our confusion matrix, it shows that on average our model is fairly accurate with some letters being slightly less consistent than others, namely W, J, and Y. We anticipate these inconsistencies may be due to similarities in certain hand signs as well as test pictures that contain a lot of noise. For example, the letter W was predicted to be E more times than W itself, which is most likely due to the fact that these hand signs are very similar.

When we tested our network on new images that we captured ourselves, the model failed to provide accurate results. Our neural network overfitting the training data that was passed into it as it was fairly inaccurate when providing our own test images. We attest much of the overfitting to the size and the diversity of our dataset. For this reason, we come to the conclusion that our model is slightly too complex for our dataset, our out of sample accuracy appears to be very good when comparing with the images that were provided from the dataset, but when we input our own images the accuracy drops significantly. This indicates to us that we should have had a more diverse dataset with each image being more dissimilar from one another to account for the different kinds of images our model would expect to see in a real world situation. At first, we believed 87,000 images to be enough, but it appears to be too little as we report our data.

5. Conclusion

To a certain extent, we achieved our goal of creating a model that can accurately read ASL hand signs. Due to the diversity of the dataset, we weren’t able to accommodate for certain variations in pixels and movements, specifically for letters ‘j’,’z’,and ‘w’. We hypothesize that had we allocated more time to experimenting with different hyperparameters, we would have expected the nature of the dataset and not underestimate the bias-variance tradeoff. Furthermore, in order to better account for the variations in an expected real world dataset, we could look to incorporate other diverse datasets which have different backgrounds, color schemes, and distances between hand and camera. This way we could have a better out of sample error.

6. References

Barczak, Andre & Reyes, Napoleon & Abastillas, M & Piccio, A & Susnjak, Teo. (2011). A New
2D Static Hand Gesture Colour Image Dataset for ASL Gestures. Res Lett Inf Math Sci.15.
https://www.researchgate.net/publication/266066851_A_New_2D_Static_Hand_Gesture_Colour_Image_Dataset_for_ASL_Gestures

Balaban, Stephen. “Deep Learning and Face Recognition: the State of the Art.” arxiv.org. Cornell University, February 10, 2019. https://arxiv.org/pdf/1902.03524.pdf

Hijazi, Samer, et al. “Using Convolutional Neural Networks for Image Recognition.” Cadence, 2015.

H, Akash. “ASL Alphabet.” *Kaggle*, 22 Apr. 2018, www.kaggle.com/grassknoted/asl-alphabet.

PyTorch. “Neural Networks” Neural Networks - PyTorch Tutorials 1.3.1 Documentation, 2017, pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html.