Classifying ASL Finger-spelling using Deep Learning

Owen McCormack
Undergraduate Student
University of Massachusetts Lowell
Lowell, United States
Owen_McCormack@student.uml.edu

***Abstract*—This report covers the exploration and results of classifying 28x28 greyscale images of letters signed in American Sign Language.**

I. Introduction

My method sets out to teach a model to identify the differences between signed letters, allowing computers to recognize American Sign Language gestures and classify them as alphabetic letters. With my method a downscaled 28x28 pixel greyscale image of a hand forming any letter sign can be inputted, and the model will determine the corresponding English letter, excluding j and z, with human-like accuracy.

The problem of sign language recognition is not new for impaired users of technology. People with certain disabilities, like the deaf, depend on sign language as their primary means of communication. Because of how often they must sign, many of these people have great fluency and accuracy delivering their signs, matching the level of detail and expression found in any other spoken language. The main difference comes down to the interpretation, signs must be seen visually with our eyes in order to be translated into meaning, as opposed to hearing spoken words through our ears. This difference allows the problem to be approached with various computer vision methods, many of which have changed significantly over the last twenty years.

The approach to sign language recognition, referred to as SLR from here on, has seen many forms but no single method or approach has been discovered that optimally interprets signs both quickly and accurately. Once a strong enough approach to SLR is discovered,g the door is open to many different kinds of gesture recognizing applications, even outside of SLR. Gesture recognition could be used to allow a new kind of input for modern devices. For example, an application's content could be scrolled by swiping the air rather putting pressure on a screen. Another example could be a deaf person signing letters and words at normal speed in front of a webcam instead of typing. Many more potential applications exist that have yet to be explored.

The increased progress and interest in Deep Learning has allowed it to become the go-to approach for solving many image classification problems such as the one this paper describes. Popular libraries like Pytorch, TensorFlow, and Caffe have opened the floodgates for deep learning, giving developers and researchers with a wide range of experience the chance to find their own solutions to complex computer vision problems. However, the means of finding the best model architecture and values for hyperparameters are still not clearly defined. Questions like: how big should my batch size be, or How many fully connected layers should I have, are usually answered through tuning and experimentation rather than a systematic approach. As more applications of deep learning are created and more researchers publish their findings, this lack of objective advice in creating Convolutional Neural Networks, referred to as CNNs from here on, design will change. My method shows that even a small CNN can have significant, near human-like performance, on an image classification problem, indicating that the power of deep learning is significant enough to replace any methods proven to work that came before it.

II. Background

As mentioned before, SLR is a rapidly evolving field. Research papers come out every year with a new approach and promising results. For SLR specifically, the type of information in the dataset varies as much as the approach to learning from the data. Reference [1] describes the various methods researchers have used to gather data, including web camera, high specification camera, camera and color gloves, 4 cameras, Kinect,

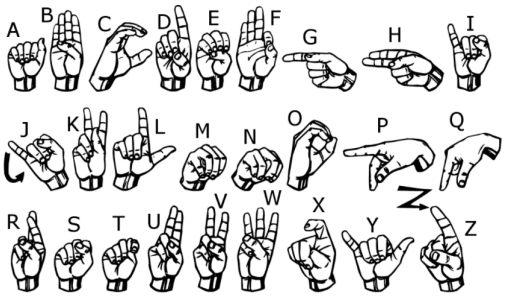

Fig. 1. Letters of American Sign Language

and more. This is critical to the topic because some methods of data acquisition may represent a more accurate portrayal of how someone using an application would get an input sample for prediction. For example, a deaf person is more likely to have a phone that takes limited resolution images and video as opposed to high specification cameras or a 5-dimension tracker. However, the amount of information that these other input methods gather could help identify more valuable features for predicting, like the bend of specific fingers. The color and depth features of the Kinect have made it the most popular source of data for researchers doing SLR, so future studies will probably value it's data rich input the most [1].

The approach to solving SLR problems has varied quite a bit as well, many researchers have used Hidden Markov Models and get good performance, even on videos [2]. Although it's an older strategy, the results of models utilizing HMMs are still able to reach accuracy in the high 80s [1]. The newest up and coming strategy involves using a 3-Dimensional CNN that is able to extract features from both the spatial and temporal dimensions with 3D convolutions [3]. The results from such 3D-CNNs have seen higher performance than previous HMM methods, but are still slightly outperformed by other strategies. One method using a support vector machine referred to as SimpSVM was able to achieve 98.9% accuracy, higher than many other recent models [4]. An important thing to note about some of these results, is that the accuracy of some models is much higher because of how the results are evaluated. Currently there is no common standard or protocol that each of these approaches share [1]. Some studies tested on videos, others simple static images and some models were tested on entirely different variants of sign language such as Chinese Sign Language, making results difficult to compare.

Many approaches were taken on the ASL MNIST dataset from Kaggle that I used. Most users chose TensorFlow to create a simple CNN however, there were some who used perceptrons, or some kind of clustering.

III. Approach

Since I was unfamiliar with deep learning and Convolutional Neural Networks at the beginning of this project, I decided to try an approach that was slightly different from what other users had submitted in most of their kernels. I implemented my own small CNN using Pytorch with strong results. My python code is broken up into various parts inside a Jupyter Notebook called Sign Language MNIST. Jupyter was a good fit for my approach because I could break the script up into smaller chunks, doing things like checking the distribution of the data, displaying one of the data images, preprocessing the data and more in each cell. Running the cells individually allowed for easy debugging and tuning of different hyperparameters.

In PyTorch, neural networks are defined in the forward method of a Net class. Developers simply have to define an architecture for the forward layers of the net, and then PyTorch will calculate all of the gradients for back propagation with a call to the backward method. My architecture has seven hidden layers total: it is made up of three convolution layers, two max pooling layers, and finally two dense layers. "Fig. 2" shows the definition of my network. I decided on three convolutions because that is the maximum possible with 3x3 kernels and max pooling for a 28x28 image, which is what the dataset provides. With a 3x3 kernel, each convolution cuts the length and width of the input image down by two pixels. A 3x3 kernel with stride 1 was chosen since the input image was very small, meaning each pixel has a lot of importance. Each convolutional layer is followed by a Rectified Linear Unit, or ReLU, nonlinearity as shown in (1).

$$\begin{aligned} h &= \max(0, a) \\ a &= Wx + b \end{aligned} \tag{1}$$

ReLU yields much better performance when compared to sigmoid or tanh as an activation function since there is much less of a risk of the vanishing gradient problem, when gradients become too large or small and prevent further optimization of weights. After the activation, comes the max-pooling. Max-pooling down samples the image pixels by picking the max in a kernel, making a much lower resolution image that generalizes the

```
class Net(nn.Module):
def __init__(self):
  super(Net, self).__init__()
  self.conv1 = nn.Conv2d(1, 64, 3)
  self.pool = nn.MaxPool2d(2,2)
  self.conv2 = nn.Conv2d(64, 64, 3)
  self.fc1 = nn.Linear(64*3*3, 128)
  self.fc2 = nn.Linear(128, 24)
  self.dropout = nn.Dropout()

def forward(self, x):
  x = self.pool(F.relu(self.conv1(x)))
  x = self.pool(F.relu(self.conv2(x)))
  x = F.relu(self.conv2(x))
  x = x.view(-1, 64)
  x = F.relu(self.fc1(x))
  x = self.dropout(x)
  x = self.fc2(x)
return x
```

Fig. 2. Pytorch Net Class

important pixels and saves computation time. Every max-pooling layer cuts both the width and height in half while leaving the depth untouched, allowing the 28x28 input to drop all the way down to 3x3 pixel image.

The architecture described thus far first takes in the input 1x28x28 greyscale tensor, then the first convolution is applied, combing over the input with a 3x3 kernel and outputting a 64x26x26 tensor. Then max-pooling is applied, cutting the tensor down to 64x13x13. After that a second convolution with another 3x3 kernel brings the dimensions down to 64x11x11 followed by max-pooling down to 5x5x64. Now the final convolution is applied, again with a 3x3 kernel, this time lowering the dimension to 64x3x3. From this point on the network will take its tensor and squeeze it down to be linear, proceeding into dense layers.

My network contains only two dense layers: the first takes the remaining 3x3x64 dimensional output from the final pooling layer and stretches it out into a 128x1 tensor. Before proceeding to the second linear layer, dropout is applied with a probability of 0.5. Dropout keeps the model regularized by randomly zeroing out some of the elements before they are sent to the next dense layer [5]. Results using dropout with probability lower than 0.5 achieved higher accuracy on the validation set with less training but, were most likely overfitting the training data too much. After dropout has been applied the model shrinks the 128x1 tensor down into a 26x1

```
tensor([18, 20,  8,  5, 18, 16,  2, 12, 24, 22])
```

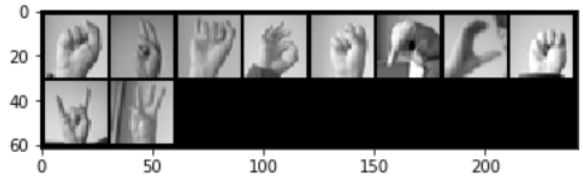

Fig. 3. Example mini-batch with labels

tensor, providing the final score for each of the labels. The values that come out of the network for the labels are somewhat difficult to interpret on their own, therefore a log softmax function is applied in the first step of the loss function to provide a more understandable output.

I chose to use Cross Entropy as the loss function for my model. In Pytorch, nn.CrossEntropyLoss() applies a log softmax to the final tensor to get values that sum to 1 and then finds the final loss value by applying negative log likelihood. Since the data used is close to uniform, shown in the dataset section, no special weights are applied to any of the labels.

$$\text{Loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) \quad (2)$$

$$\text{Loss}(x, class) = -x[class] + \log\left(\sum_j \exp(x[j])\right) \quad (3)$$

Although this model could use regular softmax, log softmax punishes bigger mistakes in the likelihood space and resulted in better fitting weights. I also used Adam as an optimizer in order to update the weights every step. Adam improves upon regular stochastic gradient descent by averaging the gradients and their second moments [6].

While the network has been broken down as if one image is going through, my network actually takes in minibatches of ten images at a time, so the shape of each tensor as it goes through the network is actually 10 by depth by height by width.

IV. Dataset

The dataset I used for my approach comes from a Kaggle challenge originally posted a year ago titled "Sign Language MNIST". The dataset has two individual files: sign_mnist_train.zip and sign_mnist_test.zip. Both files contain a number of rows, each with 784 columns of 0-255 integer values representing every pixel in a

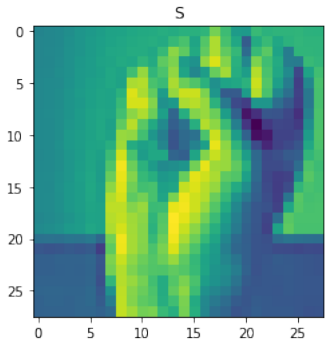

Fig. 4. Image assembled from a training data row

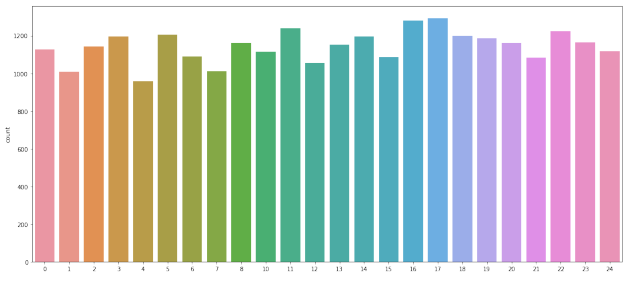

Fig. 5. Distribution of training data

single 28x28 pixel greycale image. The only difference between them, besides the training set having a label column, is that the training set contains 27,455 samples while the test set contains 7,172 samples for use. The labels in the test set take an integer value representing their corresponding letter, e.g. A = 0, B = 1, and so on. Letters J and Z are absent from the dataset since these numbers require motion which would require video footage. The images in the dataset come from a set of 1,704 unique color images of signs that have been manipulated with ImageMagik in various ways in order to extend the amount of data.

The distribution of the training data is close to uniform, all labels have around 1000 samples. The distribution of the test set varies a bit more, with some labels having close to 500 samples while others have less than 200.

In order to process all of the data I used a pandas DataFrame as well as a custom PyTorch Dataset along with three different Dataloaders, for training, validation, and test. The Dataset seperates each row of the data file into image and label by dropping the label column off of the dataFrame and then reshaping the remaining columns to be 28x28 before inserting them into a tensor.

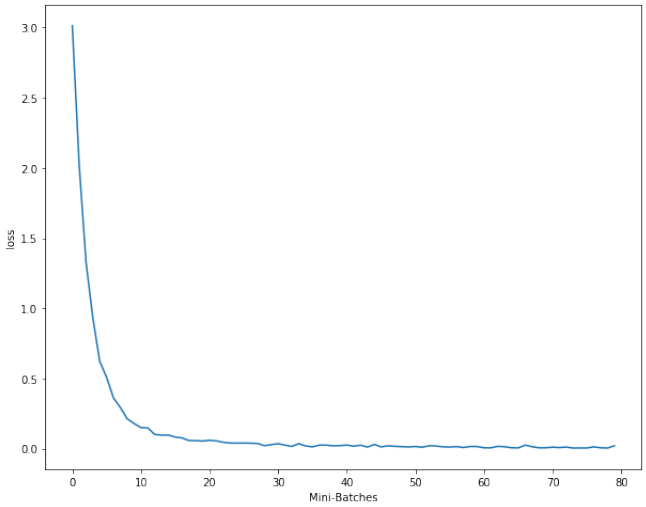

Fig. 6. Training results

V. Evaluation

As mentioned previously, a separate testing data csv file was used to evaluate the final accuracy of my model. Every image in the test set was run through the network and predicted using a custom dataloader fed in minibatches from the test set. Before moving on to the test set, I ran my model against the validation set, made up of 20% of the training data. In order get more details about the results, Scikit Learn's confusion_matrix and classification_report methods were used. Both of these reports effectively break down the correctness of a model's prediction and allow more insight into what went wrong.

For training, my model ran through the training data 20 times, tracking the loss every set of 500 mini-batches, since each mini-batch was only 10 images. Once the training completed I would use the weights to predict the validation set and then adjust my architecture accordingly. After going through many different variations of layers and hyper parameters in my model, I locked everything in and ran with the test data once for the final results.

According the results in "Fig. 7" I was able to achieve an overall accuracy of about 92%,. My model struggled the most with recognizing letters I(8), R(17), and N(13), but managed to become very good at recognizing B(1), D(3), and F(5). Looking at the accompanying signs for the letters proves interesting; each sign with lower accuracy has a different sign that is very similar to it. This could mean that my network needed more layers to properly differentiate the features found in these images,

	precision	recall	f1-score	support
0.0	0.89	1.00	0.94	331
1.0	1.00	0.95	0.97	432
2.0	0.93	0.97	0.95	310
3.0	0.97	0.97	0.97	245
4.0	0.97	0.97	0.97	498
5.0	0.97	1.00	0.98	247
6.0	0.93	0.84	0.89	348
7.0	0.94	0.94	0.94	436
8.0	0.84	0.89	0.86	288
10.0	0.93	0.94	0.93	331
11.0	0.88	0.99	0.93	209
12.0	0.89	0.85	0.87	394
13.0	0.86	0.76	0.81	291
14.0	0.96	0.89	0.92	246
15.0	0.94	0.98	0.96	347
16.0	0.81	1.00	0.89	164
17.0	0.84	0.74	0.79	144
18.0	0.88	0.94	0.91	246
19.0	0.89	0.77	0.83	248
20.0	0.94	0.95	0.95	266
21.0	0.93	0.83	0.88	346
22.0	0.89	0.99	0.94	206
23.0	0.87	0.96	0.91	267
24.0	0.89	0.84	0.86	332
micro avg	0.92	0.92	0.92	7172
macro avg	0.91	0.92	0.91	7172
weighted avg	0.92	0.92	0.92	7172

Fig. 7. Sklearn Classification Report

or that my model could have used more epochs in training. For example, The sign for I is very similar to both S and Y, R is close to D, and N is nearly identical to M and relatively close to S. More time spent tuning the hyper parameters such as the learning rate, epochs, and batch size could potentially rise the accuracy even higher.

VI. Conclusion

I think the greatest takeaway from my approach and results is the true power and accessibility of modern deep learning concepts and frameworks in Computer Vision applications. Prior to starting this project I had no experience with machine learning or computer vision. My level of understanding changed over the course of the semester not only because of my enrollment in both classes at UML, but also because so many different helpful resources were available on-line for free at varying skill levels. In particular, the lectures from Stanford's 2017 Deep Learning course available on their YouTube channel were essential for gaining fundamental understanding of deep learning as a whole, and it's importance within computer vision. PyTorch also had a wealth of documentation and tutorials available on their website that guided me through the process of writing code that builds my own small network, while still providing me a solid understanding of how my network actually works.

Another important takeaway is that the applications of deep learning are real and pave a bright future for users. My small network is capable of classifying signs which has many different real-life use cases. It's empowering to imagine what researchers, start-ups, and even hobby programmers will be able to create as the technology and ideas progress. Deep learning is paving a new future for computer vision, ideally one where the challenges of sign language recognition are trivial.

References

[1] Suharjito, R. Anderson, F. Wiryana, M. C. Ariesta, and G. P. Kusuma, "Sign language recognition application systems for deaf-mute people: A review based on input-process-output," *Procedia Computer Science*, vol. 116, pp. 441 – 448, 2017. Discovery and innovation of computer science technology in artificial intelligence era: The 2nd International Conference on Computer Science and Computational Intelligence (ICCSCI 2017).

[2] H. Wang, X. Chai, Y. Zhou, and X. Chen, "Fast sign language recognition benefited from low rank approximation," 07 2015.

[3] S. Ji, W. Xu, M. Yang, and K. Yu, "3d convolutional neural networks for human action recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, pp. 221–231, Jan 2013.

[4] P. Q. Thang, N. D. Dung, and N. T. Thuy, "A comparison of simpsvm and rvm for sign language recognition," in *Proceedings of the 2017 International Conference on Machine Learning and Soft Computing*, ICMLSC '17, (New York, NY, USA), pp. 98–104, ACM, 2017.

[5] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.