Neural Network Final Project Report

Title: Harnessing Deep Learning for American Sign Language Detection

Team: Sign Network | Joon Jung, Srinidhi Manikantan, Heathvonn Styles, Runru Shen

Background

In the United States alone, American Sign Language (ASL) serves as the primary means of communication for over half a million individuals[1]. Each day, approximately 33 babies are born with permanent hearing loss, around 90% of whom are born to hearing parents unfamiliar with ASL[2]. This communication gap can result in lifelong challenges, including difficulties in forming relationships, accessing healthcare, navigating employment discrimination etc.

Fingerspelling, a fundamental aspect of ASL, employs hand shapes to represent individual letters, facilitating communication of names, addresses, phone numbers, and other essential information—akin to typing on a mobile phone. Our project **exclusively focuses on enhancing fingerspelling recognition.**

Recent advancements in Neural Networks (NN) offer promising solutions to these communication barriers. By leveraging NN capabilities, we aim to revolutionize the translation process of sign language, potentially mitigating the aforementioned challenges.

Project Goals

While many existing solutions concentrate on image classification for fingerspelling, their training and testing are performed on the same dataset which may include inherent bias, making inaccurate predictions when utilized in real life settings. This project will be focusing on recognizing alphabets in ASL with a **focus on utilizing different dataset for model training and testing to assess how well the model performs on unseen data** (mimicking a real-world scenario). This can help to account for differences in skin tone, lighting, color grading etc between diverse datasets.

Secondly, based on the knowledge we have gained through this course, we also wanted to **assess how well pre-trained models and convolutional neural networks (CNN) models developed from scratch perform on our dataset.** We wanted to test if pre-trained models, owing to their complexity and training on extensive dataset, produce a better prediction accuracy than a baseline CNN Model.

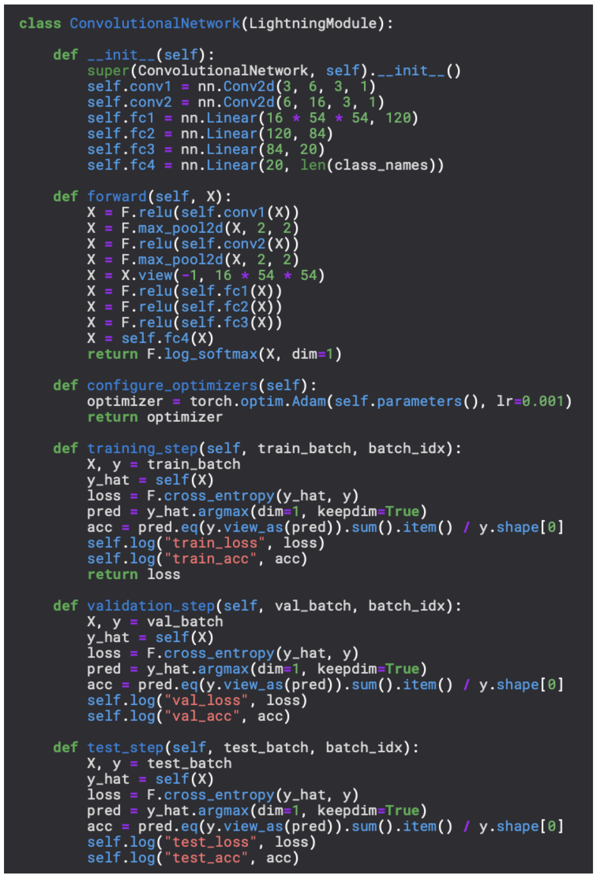
Dataset

The datasets used for this project are obtained from kaggle. **Two different datasets are utilized, the "ASL Alphabet" as training dataset [1] and "American Sign Language Dataset" as the test dataset [2].** The training dataset comprises image data for 29 features, including 26 alphabets and 3 special categories: delete, nothing, and space. Each feature contains 3000 images with the size of 200 x 200 pixels. On the other hand, the test dataset only consists of 26 alphabets, 70 images for each alphabet which totals up to 1800 images of the test dataset.

Pre-processing techniques were applied to the training dataset to augment various hand signs, enhancing the size and variety of the image dataset utilized for training the neural network. Techniques like thresholding, edge tracing, as well as adjustments to image brightness and sharpness were applied to each image in the training dataset. The training dataset was magnificent sevenfold, resulting in 21000 images per feature with a total of 600000 images in the training dataset. The primary object for augmenting the hand signs was to mitigate any inherent biases within the dataset and create a more robust model to improve the overall predictive accuracy of the models.

Models Used

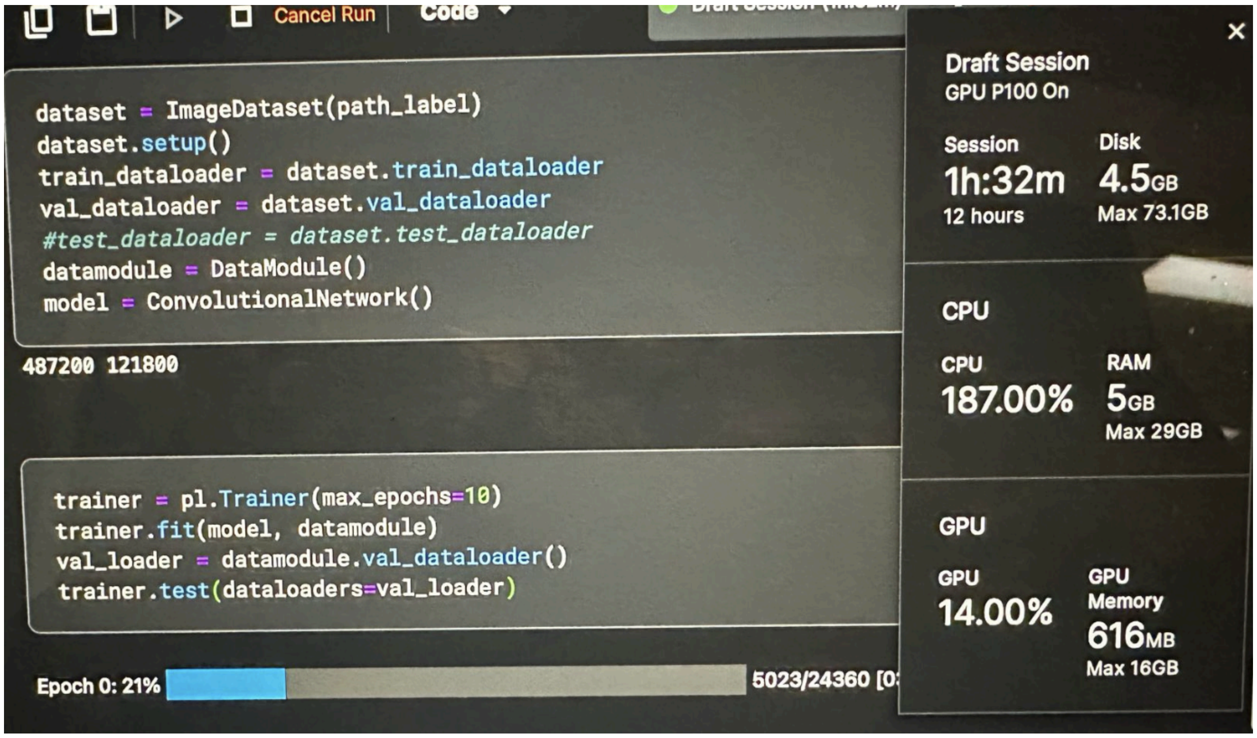
We compared two models in this project. First, we made a CNN made from scratch and we wanted to see how it would perform against a pre-trained model. We chose to compare it to ResNet50 since ResNet is one of the most powerful image processing CNN currently out there. This was the model architecture for the CNN that we created.



```python
class ConvolutionalNetwork(LightningModule):

    def __init__(self):
        super(ConvolutionalNetwork, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 3, 1)
        self.conv2 = nn.Conv2d(6, 16, 3, 1)
        self.fc1 = nn.Linear(16 * 54 * 54, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 20)
        self.fc4 = nn.Linear(20, len(class_names))

    def forward(self, X):
        X = F.relu(self.conv1(X))
        X = F.max_pool2d(X, 2, 2)
        X = F.relu(self.conv2(X))
        X = F.max_pool2d(X, 2, 2)
        X = X.view(-1, 16 * 54 * 54)
        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = F.relu(self.fc3(X))
        X = self.fc4(X)
        return F.log_softmax(X, dim=1)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
        return optimizer

    def training_step(self, train_batch, batch_idx):
        X, y = train_batch
        y_hat = self(X)
        loss = F.cross_entropy(y_hat, y)
        pred = y_hat.argmax(dim=1, keepdim=True)
        acc = pred.eq(y.view_as(pred)).sum().item() / y.shape[0]
        self.log("train_loss", loss)
        self.log("train_acc", acc)
        return loss

    def validation_step(self, val_batch, batch_idx):
        X, y = val_batch
        y_hat = self(X)
        loss = F.cross_entropy(y_hat, y)
        pred = y_hat.argmax(dim=1, keepdim=True)
        acc = pred.eq(y.view_as(pred)).sum().item() / y.shape[0]
        self.log("val_loss", loss)
        self.log("val_acc", acc)

    def test_step(self, test_batch, batch_idx):
        X, y = test_batch
        y_hat = self(X)
        loss = F.cross_entropy(y_hat, y)
        pred = y_hat.argmax(dim=1, keepdim=True)
        acc = pred.eq(y.view_as(pred)).sum().item() / y.shape[0]
        self.log("test_loss", loss)
        self.log("test_acc", acc)
```

Results

When we started running the models, we learned some important takeaways. As a product of our image augmentation, we had created 7 different variation of each image, when we already had pretty big image dataset. We ended up having 600,000 image dataset, which was massive. When we started running the models, we quickly realized that even with GPU access, our training was taking significantly a long time.


```
dataset = ImageDataset(path_label)
dataset.setup()
train_dataloader = dataset.train_dataloader
val_dataloader = dataset.val_dataloader
#test_dataloader = dataset.test_dataloader
datamodule = DataModule()
model = ConvolutionalNetwork()
```

```
487200 121800
```

```
trainer = pl.Trainer(max_epochs=10)
trainer.fit(model, datamodule)
val_loader = datamodule.val_dataloader()
trainer.test(dataloaders=val_loader)
```

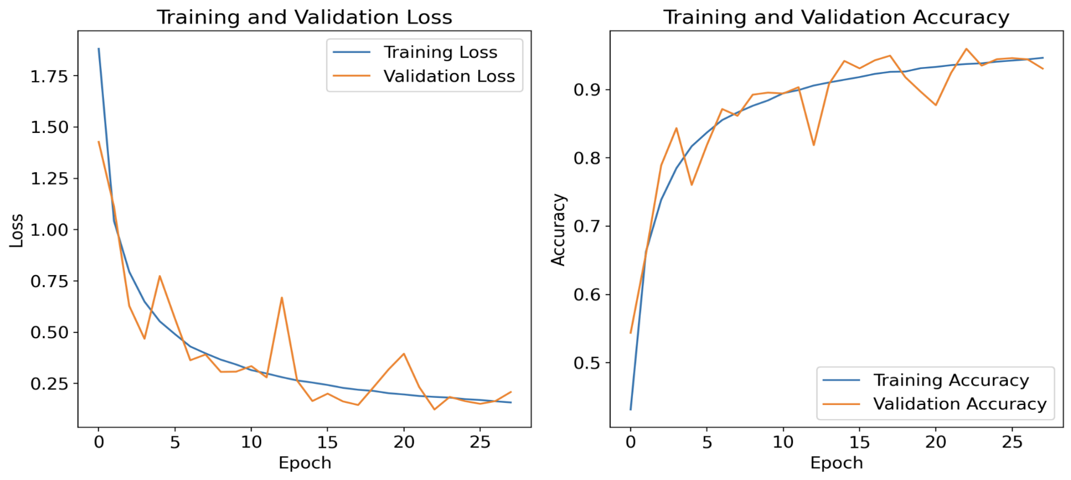
```
Epoch 0: 21%  5023/24360 [0:
```

Draft Session
GPU P100 On
Session 1h:32m 12 hours
Disk 4.5GB Max 73.1GB
CPU
CPU 187.00%
RAM 5GB Max 29GB
GPU
GPU 14.00%
GPU Memory 616MB Max 16GB

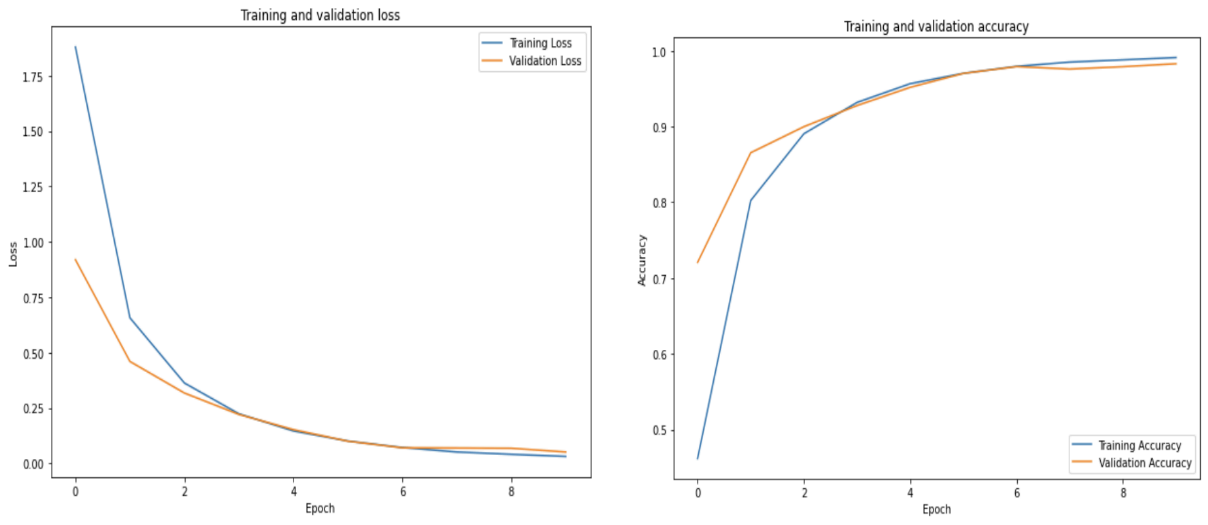
(actual picture while training. You can see that it took hour and 32 minutes to reach 21% of epoch 0, with GPU acceleration, with CPU clocking at 187%. We didn't know how to use more of the GPU, and this code ended up crashing)

So we had to improvise. We pulled the original dataset, which is 1/7th of a size, and then ran models on them first. We thought it would actually be good to run the original, and get to compare later if our image augmented dataset would do significantly better.

CNN scratch performing on original dataset



ResNet50 performing on the original dataset.

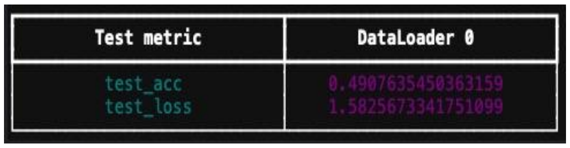


We were able to see that pre-trained model loss and accuracy graph was much smoother, and reaches high accuracy and low loss in fewer epochs.

Now, for the augmented dataset, again, since we had such big dataset, we were not able to run more than couple of epochs. For reference, it took 2 hours and 43 minutes to go through epoch 0. Not only the time constraint, we also ran into resource problems. Anything more than 2 epochs would create some problem with CUDA memory running out, general RAM running out, or actually a computer crashing (Ruby had to go through this).

So we decided to pull the weights in the middle of training, before it crashes, and we would just use those. Since we were not able to go through more than few epochs, we thought showing loss and accuracy graph wouldn't be appropriate, since you can not tell the progress. Instead, we decided to show the actual performance of the model after we pull the weights.

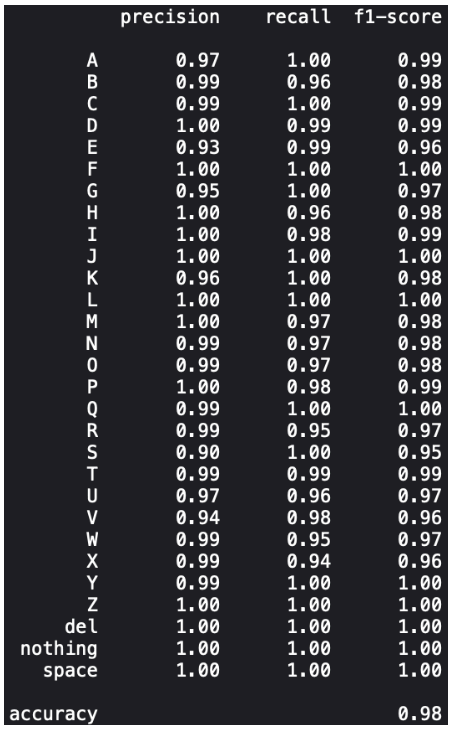
We came to a surprising result. Even after creating 600,000 images and taking almost 3 hours to train, the model made from scratch did not perform the best.



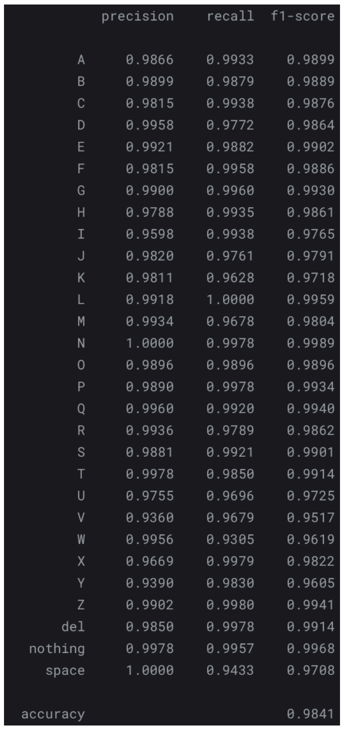
```
┌─────────────────────┬─────────────────────┐
│     Test metric     │    DataLoader 0     │
├─────────────────────┼─────────────────────┤
│      test_acc       │ 0.4907635450363159  │
│      test_loss      │ 1.5825673341751099  │
└─────────────────────┴─────────────────────┘
```

only 49% accuracy.

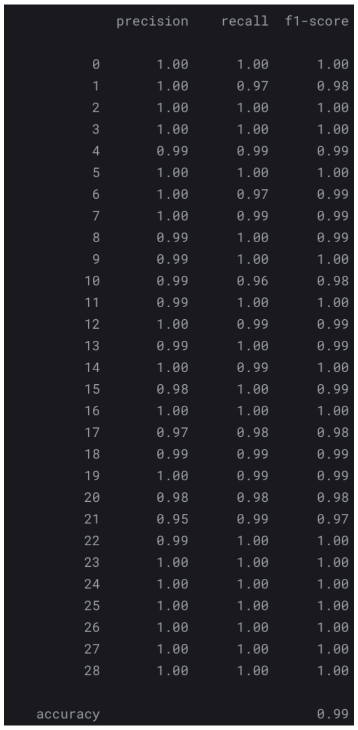
We were able to train couple of epochs for the ResNet50 at last. We put in all the resources in making ResNet 50 model work, since the scratch CNN model performed poorly. This was the results of training, instead of the loss and accuracy graph.



```
          precision    recall  f1-score

       A       0.97      1.00      0.99
       B       0.99      0.96      0.98
       C       0.99      1.00      0.99
       D       1.00      0.99      0.99
       E       0.93      0.99      0.96
       F       1.00      1.00      1.00
       G       0.95      1.00      0.97
       H       1.00      0.96      0.98
       I       1.00      0.98      0.99
       J       1.00      1.00      1.00
       K       0.96      1.00      0.98
       L       1.00      1.00      1.00
       M       1.00      0.97      0.98
       N       0.99      0.97      0.98
       O       0.99      0.97      0.98
       P       1.00      0.98      0.99
       Q       0.99      1.00      1.00
       R       0.99      0.95      0.97
       S       0.90      1.00      0.95
       T       0.99      0.99      0.99
       U       0.97      0.96      0.97
       V       0.94      0.98      0.96
       W       0.99      0.95      0.97
       X       0.99      0.94      0.96
       Y       0.99      1.00      1.00
       Z       1.00      1.00      1.00
     del       1.00      1.00      1.00
 nothing       1.00      1.00      1.00
   space       1.00      1.00      1.00

accuracy                           0.98
```



```
          precision    recall  f1-score

       A     0.9866    0.9933    0.9899
       B     0.9899    0.9879    0.9889
       C     0.9815    0.9938    0.9876
       D     0.9958    0.9772    0.9864
       E     0.9921    0.9882    0.9902
       F     0.9815    0.9958    0.9886
       G     0.9900    0.9960    0.9930
       H     0.9788    0.9935    0.9861
       I     0.9598    0.9938    0.9765
       J     0.9820    0.9761    0.9791
       K     0.9811    0.9628    0.9718
       L     0.9918    1.0000    0.9959
       M     0.9934    0.9678    0.9804
       N     1.0000    0.9978    0.9989
       O     0.9896    0.9896    0.9896
       P     0.9890    0.9978    0.9934
       Q     0.9960    0.9920    0.9940
       R     0.9936    0.9789    0.9862
       S     0.9881    0.9921    0.9901
       T     0.9978    0.9850    0.9914
       U     0.9755    0.9696    0.9725
       V     0.9360    0.9679    0.9517
       W     0.9956    0.9305    0.9619
       X     0.9669    0.9979    0.9822
       Y     0.9390    0.9830    0.9605
       Z     0.9902    0.9980    0.9941
     del     0.9850    0.9978    0.9914
 nothing     0.9978    0.9957    0.9968
   space     1.0000    0.9433    0.9708

accuracy                         0.9841
```



```
              precision    recall  f1-score

           0       1.00      1.00      1.00
           1       1.00      0.97      0.98
           2       1.00      1.00      1.00
           3       1.00      1.00      1.00
           4       0.99      0.99      0.99
           5       1.00      1.00      1.00
           6       1.00      0.97      0.99
           7       1.00      0.99      0.99
           8       0.99      1.00      0.99
           9       0.99      1.00      1.00
          10       0.99      0.96      0.98
          11       0.99      1.00      1.00
          12       1.00      0.99      0.99
          13       0.99      1.00      0.99
          14       1.00      0.99      1.00
          15       0.98      1.00      0.99
          16       1.00      1.00      1.00
          17       0.97      0.98      0.98
          18       0.99      0.99      0.99
          19       1.00      0.99      0.99
          20       0.98      0.98      0.98
          21       0.95      0.99      0.97
          22       0.99      1.00      1.00
          23       1.00      1.00      1.00
          24       1.00      1.00      1.00
          25       1.00      1.00      1.00
          26       1.00      1.00      1.00
          27       1.00      1.00      1.00
          28       1.00      1.00      1.00

    accuracy                           0.99
```

The first two were for the training set. You can see that it contains three additional classification categories called del, nothing, and space. It was one of the things we have done to increase the accuracy, so that the model actually learns just the background space, or nothing meaning no alphabet spelling associated with it. By learning those, we thought it won't be forced to think and classify those as letters when they are not. On the right, you can see the test result, with row names actually represented in numbers, achieving accuracy of 99%.

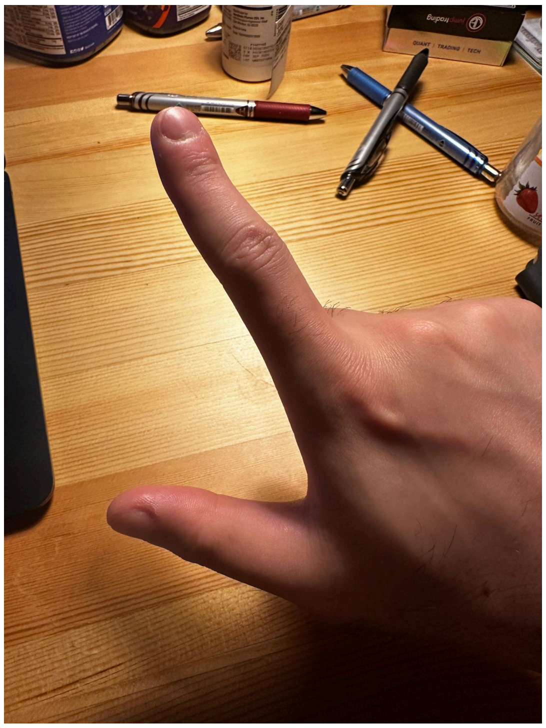
Lessons Learnt

We initially employed data augmentation to expand our dataset, anticipating enhanced variation and improved model performance. However, we discovered that **simply increasing the dataset size isn't always the optimal strategy**. Our efforts were constrained by challenges related to memory and computational power. Hence, we recognize the importance of exercising caution when selecting or augmenting datasets, considering these limitations.

Future Direction

In our project, we dive into various aspects of image classification for ASL. With more time, we would like to shift our focus towards **identification of ASL in videos.** Pose detection, where joints are recognized in order to identify specific movements, will be a necessity since analyzing videos frame by frame is far from ideal. Our goal is to not only recognize individual gestures or motions with images for clips of videos, but also to **understand the combination of these elements to interpret complete ASL in daily settings**. With future research we hope that effective translation can be achieved between ASL and English to be applicable in daily lives. Additionally, just as there are many different languages, we can **extend the same method to other sign languages** like British sign languages, French sign languages, and beyond.

Bonus: we never got time to actually implement and see if the model we trained would do well predicting L sign spelling off of my hand, which can not make 90 degree angle. We would've had to go back and run, train, and wait more than 3 hrs to go through Epoche 0. I pulled the weights around 3-4 epochs for the results that I showed above, so we would have to spend at least 10-12 hrs again just to test my hand images. If we had more time, we would love to try that as future directions. :)



Will our model be able to predict this as an L sign? Maybe one day in future we may know the answer to that…

Reference

1. Mitchell, Ross E., et al. (2006). How Many People Use ASL in the United States? Gallaudet University, Gallaudet Research Institute, Washington, D.C.
2.

https://www.nidcd.nih.gov/health/statistics/quick-statistics-hearing

Data Sources

[1] Nagaraj, A. (2018). ASL Alphabet [Dataset]. Kaggle. https://www.kaggle.com/grassknoted/aslalphabet_akash

[2] Thakur, A. (2019) American Sign Language Dataset [Dataset]. Kaggle. https://www.kaggle.com/datasets/ayuraj/asl-dataset/code