DATS 6203 Project Final Report

Group 9: Toufik Bouras, Noah Olsen, Teryn Zmuda

Topic: Classifying American Sign Language (ASL) Images

Network: Convolutional Neural Network (CNN) with Pytorch

Overview

The problem we would like to solve is classifying American Sign Language (ASL) images. We wanted to do this project in order to gain experience applying CNN to images. The dataset is published on Kaggle platform and has 87,029 images and is over 1 GB, which is sufficient to train a deep network. We adapted VGG16 and Resnet50 models and created a custom model and compared their performances. We used Pytorch for this project, and Google Cloud Platform (GCP) and (Colab) to allow for appropriate computational power on the images.

Dataset Description

We downloaded the data from kaggle platform. This data contains a total of 87,029 images, each 64 x 64 pixels. The training data includes 87,000 images, with 3,000 files for each class. The validation set is 29 images, which allows for one classification for each image within the 29 classes. Classes include 26 letters from A-Z and three other classes: space, delete, and nothing. The data is located at https://www.kaggle.com/grassknoted/asl-alphabet.

Deep Learning Network and Algorithm

In this project we used a convolutional neural network (CNN) to predict the target variable. We needed to determine the right batch size in order to get the highest performance possible.The batch allows for efficiency: if the data is small then the entire batch of data can be used to compute the gradient, but if it is big then the mini-batch would allow for a greater efficiency.

1. The Convolutional Neural Network(CNN)

The Convolutional Neural Network(CNN) contains different building blocks. It consists of convolutional layers, pooling layers and fully connected layers. This last one is not required but it is widely used. This king of networks is usually used for computer vision tasks. From the input to the output the image is submitted to different filters, these filters contains learnable parameters. Each filter processes a specific feature of the input image, the output of a filters is called a feature map. Usually the filters are followed by a nonlinear activation function. When there is a convolutional layer followed by a fully connected layer we need to flatten the data by stacking up all the columns of the feature map on over the other before passing it through the fully connected layer. In the last layer if we have a classification problem we use a loss function called Softmax then we estimate the cost function for the batch size. We use this last one in order to update the weights and biases in the backpropagation phase.

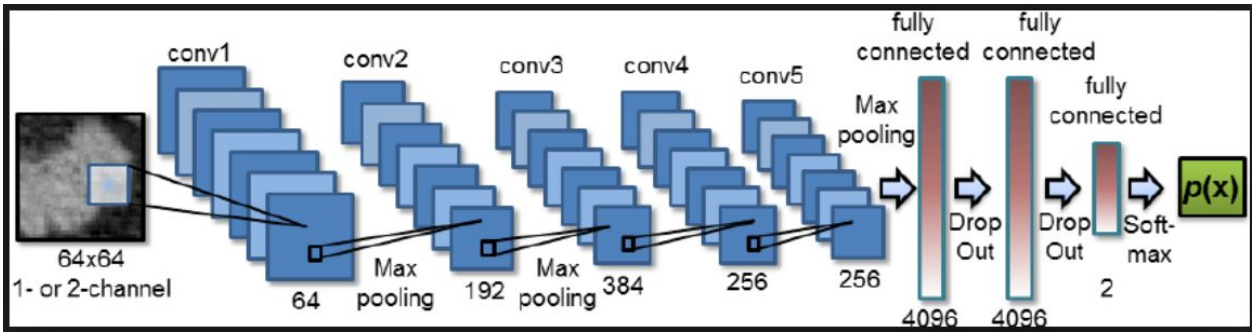

Fig1: Example of CNN

2. CNN: VGG-16 and Resnet

We experimented with two CNN models, including VGG-16 and Resnet50. While AlexNet was one of the first CNN models, the efficiency was not optimal and there was heavy data augmentation. AlexNet was trained on GPU's with only 3GB of memory, so the feature maps were spread to accommodate data. AlexNet is used as a reference and comparison method, and as seen in Fig. 2, VGG-16 has many more layers. VGG-16 was developed by K.Simonyan and A.Zisserman as a deeper network, with more layers with trainable weights. Most of the memory in a VGG-16 is in the beginning layers and the parameters are in the last layers.

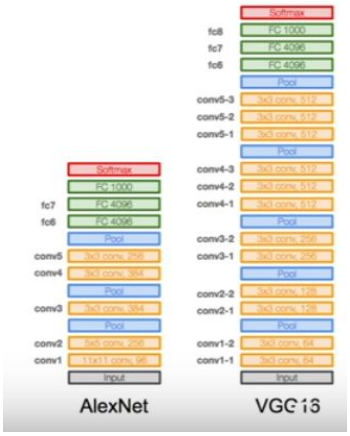

Fig. 2: From AlexNet to VGG 16

The Resnet network is a different approach where the weight layer learns the residual, the output of the transformation. The transformation is represented by f(x) + x.

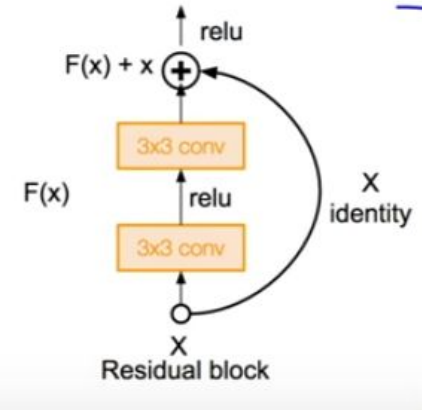

Fig. 3: Resenet Framework

Experimental Setup

As we mentioned in the introduction we used three different models all with the same parametres. These parameters are: Batch_size = 16, number of epochs = 5, and the learning rate = 0.0001. We normalized and augmented the data in order to improve accuracy and reduce overfitting. We used the same lost function(NLLLoss) and the optimizer(Adam) for the three models.

We started with creating a reference model using Keras(in Colab platform). We modified this model by adding different layers including: convolutional, max-pooling, and Dropout layers. After several trials we were able to get a very good model. In this model we included 12 convolutional layers followed by 2 fully connected layers. We normalized all the convolutional layers, after every four conv layers we included a max-pooling layer except the last four layers. In the development of this model we tried to follow the conventions reported by well known AI researchers like Andrew NJ. In the original model we created in Keras, we did not include drop out layers for some convolutional layers, because when we were developing the model we noticed that including several (2-4) conv layers with the same number of input channels and different kernel, stride, and padding sizes yields a better accuracy results than the drop out layers. In the Pytorch model we could not implement the same model we created in Keras(which reached an accuracy of 98.3% in val_set after 40 epochs) because we were confused with the Pytorch framework. Here is a part of the model we created in Keras which performed very well:

```
my_model = Sequential()
my_model.add(Conv2D(32, kernel_size=3, strides=1, activation='relu', padding='same', kernel_constraint=max_norm(3), input_shape=target_dims))
my_model.add(BatchNormalization())
my_model.add(Dropout(0.2))
my_model.add(Conv2D(32, kernel_size=3, strides=1, activation='relu', padding='same'))
my_model.add(BatchNormalization())
my_model.add(Conv2D(32, kernel_size=4, strides=2, activation='relu'))
my_model.add(BatchNormalization())
my_model.add(Conv2D(32, kernel_size=5, strides=1, activation='relu', kernel_constraint=max_norm(3)))
my_model.add(BatchNormalization())
my_model.add(Dropout(0.2))
my_model.add(MaxPooling2D(pool_size = (2, 2),  padding='same'))
```

And here is a part of the model we created in Pytorch:

```
self.conv1 = torch.nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
self.conv1_bn = nn.BatchNorm2d(32)
self.relu = nn.ReLU()
self.conv2 = torch.nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=1)
self.conv2_bn = nn.BatchNorm2d(32)
self.relu = nn.ReLU()
self.pool = torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
```

In the other two models we used what we call transfer learning where we use a model trained on a data of a similar format and adapt it into our data by modifying the last layers of the CNN.
In these two models VGG-16 and resnet-50 we kept the model weights unchanging during the training(so no weight update) and we replaced the last fully connected layer with two other fully connected layers. We only trained the weight of these last two layers. We needed to preprocess the data and specify the number of the classes in the output in order to run these two model properly. Here is a part of this model:

```
# Choose the right argument for x
net = models.vgg16(pretrained=True)
# Freeze model weights
for param in net.parameters():
    param.requires_grad = False

num_ftrs = net.classifier[6].in_features

# Add on classifier
net.classifier[6] = nn.Sequential(
                      nn.Linear(num_ftrs, 256),
                      nn.ReLU(),
                      nn.Dropout(0.4),
                      nn.Linear(256, num_classes),
                      nn.LogSoftmax(dim=1))
```

Results

With our model, we started with very low accuracy and the model learned significantly over its lifetime. The loss also improved overtime, but overall, there was very slow performance.

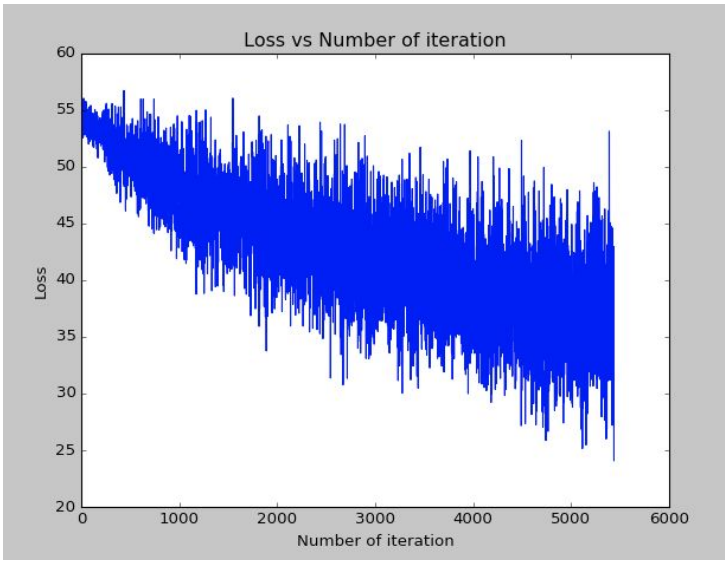

Fig. 4: Our model loss over time

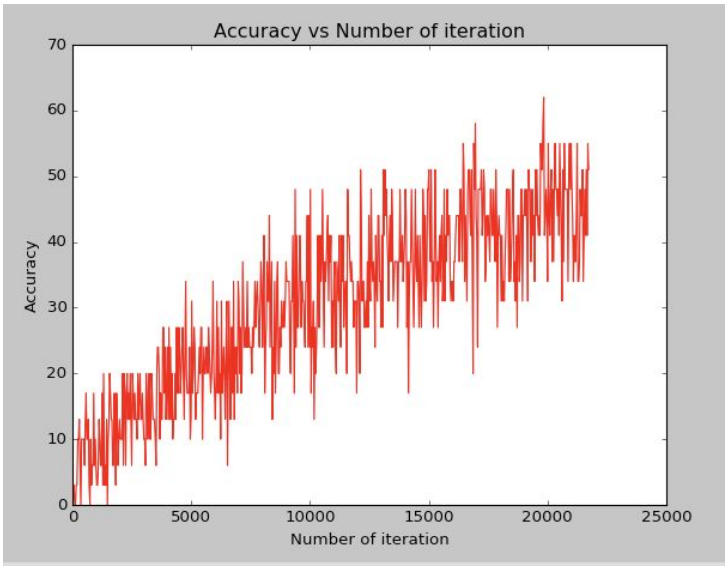

Fig. 5: Our model accuracy over time

Moving forward with VGG-16, the results are shown in Fig. 6 and Fig. 7 and indicate very low accuracy at initiation, which steadies and completes at 60%. Overall, the model is very slow and has a bad performance comparing to the other models.

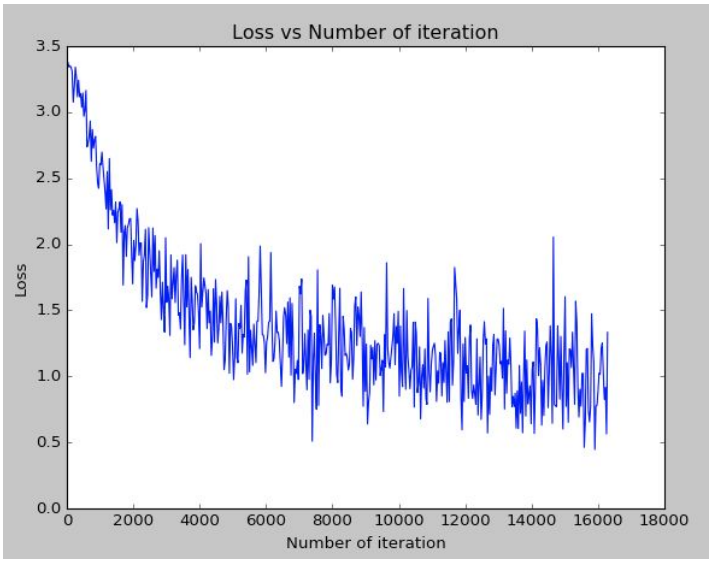

Fig. 6: VGG-16 loss over time

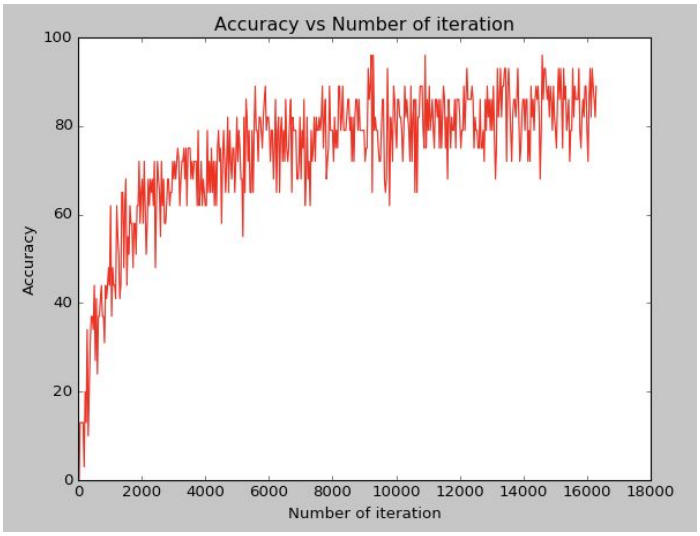

Fig. 7: VGG-16 accuracy over time

The adapted Resnet50 model started off with very low accuracy but then scaled rapidly and had impressively quick performance. As demonstrated in Fig. 7, we ended with 90% accuracy.

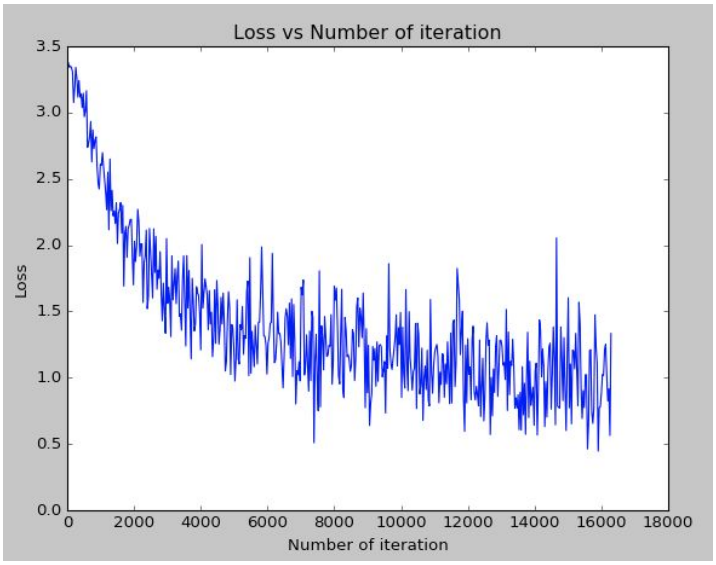

Fig. 6: Resnet50 model loss over time

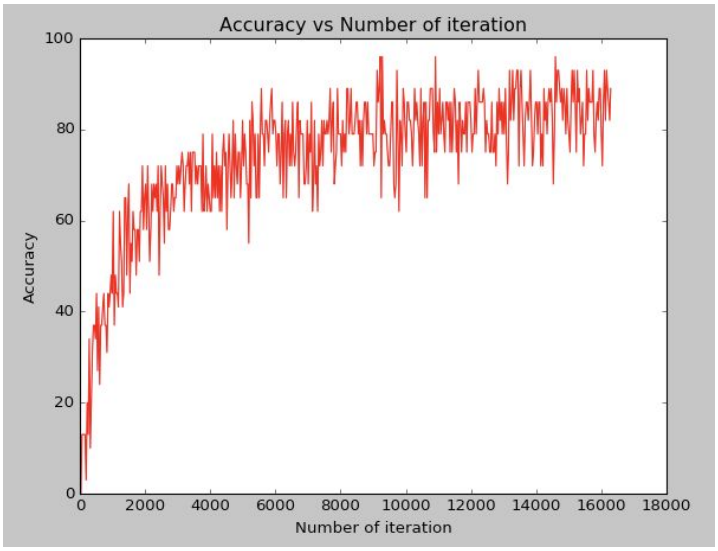

Fig. 7: Resnet50 model accuracy over time

Summary and Conclusion

Using transfer learning and keeping the model weights unchanged for the CNN architectures VGG-16 and Resnet, we were able to demonstrate accuracy as iterations are introduced. the Resnet-50 model has outperformed the other models over the 5 epochs. At first sight it did not make a sense for such a deep neural network to perform this well in this simple problem with a good data, but in fact Resnet architecture allows to keep the apparent features throughout the network while learning the details of the images.

References

Akash. (2018, April 22). ASL Alphabet. Retrieved from https://www.kaggle.com/grassknoted/asl-alphabet

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. CoRR, abs/1409.1556 . Retrieved from http://arxiv.org/abs/ 1409.1556

Stanford School of Engineering, CNN Architecture Lecture. Aug. 11, 2017. Retrieved from https://www.youtube.com/watch?v=DAOcjicFr1Y