Final Project Report for CS 175, Spring 2018

Project Title: ASL Hand Digit Classification
Project Number: 21

Student Names:
Ryan Liu, 73638562, liury1@uci.edu
Ryan Chan, 73612265, chanr5@uci.edu

1. Introduction and Problem Statement:

In ASL, there are hand signs denoting digits 0-9. This project will use Google's TensorFlow API to create a machine learning model to classify images of various hand gestures into classes 0-9. This problem is classified as hand gesture recognition so our inputs will be pictures of various hand signs and we will be outputting a classification of what each sign is. Our work will involve creating both a convolutional network and a Squeezenet, and comparing the results of the two. SqueezeNets are a type of deep neural network designed to run on hardware with limited memory.

For our final product, we would like to input a hand of some pose and have our squeezenet output a classification for what digit this gesture represents. After that point, we will see how a deep neural network performs against our Squeezenet, and discuss the similarities, differences, and costs of both networks.

2. Related Work:

Image classification has been an interesting problem that has existed for a while, and currently, deep convolutional networks have been a popular tool of choice. A joint detection algorithm would have been a good choice for hand pose estimation because it can deal with occlusion and make classifications based on relative position of joints. For our project, we opted go with a similar approach to the course assignments of classifying images. Joint position could be very helpful in recognizing poses when occlusion occurs or if there is noise in the image, however our data did not have noise, so we found it unnecessary.

Deep convolutional networks have been used in class before, and we will create one to see how well a covnet works on this dataset as a baseline. They are found to be good at image recognition problems, but do require significant processing power and a large training dataset to reach good accuracies.

SqueezeNets are a newer type of deep neural network architecture created with the goal of maintaining accuracy while using less resources. In a Squeezenet conference paper, the team managed to achieve AlexNet-level accuracy on ImageNet with 50x fewer parameters while also reducing its memory size to less than 0.5 MB.

Three techniques were used to compress the model to such a compact size. The first technique was to make networks smaller by replacing 3x3 filters with 1x1 filters. This 1x1 filter, also known as the "Squeeze" layer, leads to a dimensionality reduction. This also leads to less overfitting due to a smaller kernel size. The second technique described is to reduce the number of inputs for the remaining 3x3 layers. Expand layers are convolution layers of 1x1 filters and 3x3 filters. By reducing the number of filters in the squeeze layer feeding into the expand layer, the total number of connections entering the 3x3 filters are reduced, thus reducing total number of parameters. The last technique used is to downsample late in the network so convolution layers have large activation maps.

Squeezenet Conference Paper: *https://arxiv.org/pdf/1602.07360.pdf*

3. Data Sets

URL: https://www.kaggle.com/ardamavi/sign-language-digits-dataset

We used the “Sign Language Digits Dataset” found on kaggle. This dataset is a set of 2,062 labeled images of hands holding various hand poses to denote different digits.

There are ten classes, and each image is a 64x64 pixel image in the RGB colorspace. At a total of ~2000 images, we have a sample of ~200 images per classification.

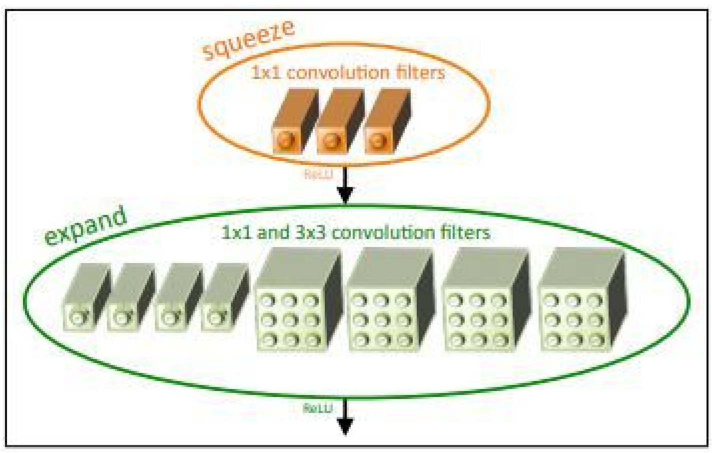
After some testing, we found out that this was not enough data to work with. In an attempt to mitigate this issue, we applied reflections to the images to increase the variance in the images. These reflections doubled the database, bringing us to ~4000 images.

4. Description of Technical Approach

We started our project by processing the data gathered from kaggle. We ended up using code from kaggle to process the data into a format that we could work with. The database seemed a little too small for us, so we then decided to add in reflections of each image of the database, effectively doubling the data that we have to work with. This data is to be split into a 80:20 split for training and testing.

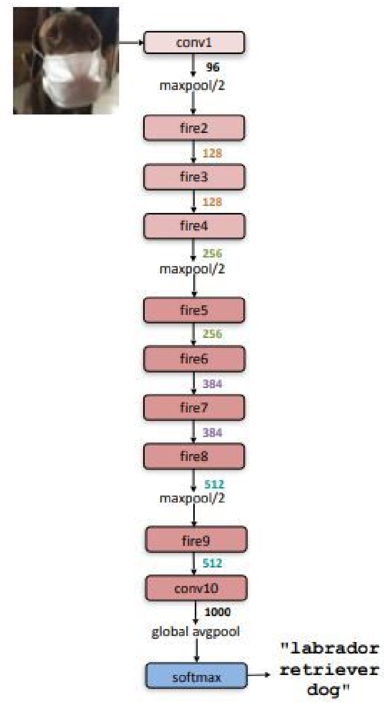
Next, we will create a convolutional network as a baseline of comparison. We used a similar architecture to the one we used in assignment 4 with multiple convolutional layers with max pooling and relu activation. After the convolutional layers we added the first fully connected layer then performed batch normalization and finished the network with a final fully connected layer. This was our baseline because we were able to get it above 70% on the CIFAR-10 dataset, so we expected to get an accuracy close to that because we are also attempting classification. We use a learning rate of .001 over 20 epochs with a batch size of 100. Our final CNN architecture is:

1. Convolution layer
2. Pool layer
3. Relu layer
4. Convolution layer
5. Pool layer
6. Relu layer
7. Convolution layer
8. Pool layer
9. Relu layer
10. Flatten
11. Fully connected
12. Batch Normalization
13. Fully connected layer



Fire Module Architecture

After creating the CNN, we decided to research to learn as much as we could about squeezenets, how they work, and how to create one. What we learned was that Squeezenets are essentially a clever configuration of 1x1 convolution layers and 3x3 convolution layers. The example we looked at created something called a "fire module." A fire module consisted of a squeeze and an expand layer, using a 1x1 convolution followed by a mix of 1x1 and 3x3 convolution layers, as described earlier in the report. We looked to the github link embedded in the report to see how they created the SqueezeNet, and replicated it in TensorFlow. The full architecture as that we created is modeled after the conference paper, and is such:



- layer 1 - conv 1
- layer 2 - maxpool
- layer 3-5 - fire modules
- layer 6 - maxpool
- layer 7-10 - fire module
- layer 11 - maxpool
- layer 12 - fire 9 + dropout
- layer 13 - final convolution and avg pooling

Squeezenets prefer a smaller learning rate, so we chose 0.0001 to be our learning rate. To balance such a small value, we increased the batch size to 500 and the number of epochs to 300 to be able to reach high accuracies with this model.

After creating and running our two models, we looked at validation scores and runtimes to evaluate performance.

SqueezeNet written in Caffe: https://github.com/DeepScale/SqueezeNet

5. Software

Technology	**High Level Description**
Python	Main coding language used in this project.
TensorFlow	Open Source Machine Learning framework
CNN	Popular neural network used to solve image recognition/ classification problems.
SqueezeNet	Newer neural network configuration designed to be lightweight.
Preprocessing	Taken from kaggle. Resizes images to 64x64 and converts RGB images into grayscale images.

6. Experiments and Evaluation

Setup:

We decided that the best metric to compare our networks would be validation scores and time taken to run. To do this, we split the data into a 80/20 split between training and testing. Each Epoch, we will create a different train and validation set, and use mini batches to train. We also decided to use AdamOptimizer for both measures, but chose the best learning rate for each learner.

Results:

```
Epoch: 1/20     Training Loss: 0.929    Validation Loss: 0.973   Accuracy: 77.21%
Epoch: 2/20     Training Loss: 0.744    Validation Loss: 0.654   Accuracy: 85.94%
Epoch: 3/20     Training Loss: 0.421    Validation Loss: 0.454   Accuracy: 89.58%
Epoch: 4/20     Training Loss: 0.272    Validation Loss: 0.358   Accuracy: 89.70%
Epoch: 5/20     Training Loss: 0.214    Validation Loss: 0.269   Accuracy: 94.06%
Epoch: 6/20     Training Loss: 0.191    Validation Loss: 0.176   Accuracy: 98.06%
Epoch: 7/20     Training Loss: 0.140    Validation Loss: 0.123   Accuracy: 98.06%
Epoch: 8/20     Training Loss: 0.101    Validation Loss: 0.114   Accuracy: 98.18%
Epoch: 9/20     Training Loss: 0.084    Validation Loss: 0.078   Accuracy: 99.15%
Epoch: 10/20    Training Loss: 0.065    Validation Loss: 0.058   Accuracy: 99.52%
Epoch: 11/20    Training Loss: 0.061    Validation Loss: 0.048   Accuracy: 99.88%
Epoch: 12/20    Training Loss: 0.034    Validation Loss: 0.036   Accuracy: 99.76%
Epoch: 13/20    Training Loss: 0.035    Validation Loss: 0.027   Accuracy: 100.00%
Epoch: 14/20    Training Loss: 0.019    Validation Loss: 0.022   Accuracy: 100.00%
Epoch: 15/20    Training Loss: 0.017    Validation Loss: 0.016   Accuracy: 100.00%
Epoch: 16/20    Training Loss: 0.013    Validation Loss: 0.012   Accuracy: 100.00%
Epoch: 17/20    Training Loss: 0.012    Validation Loss: 0.010   Accuracy: 100.00%
Epoch: 18/20    Training Loss: 0.009    Validation Loss: 0.009   Accuracy: 100.00%
Epoch: 19/20    Training Loss: 0.012    Validation Loss: 0.007   Accuracy: 100.00%
Epoch: 20/20    Training Loss: 0.007    Validation Loss: 0.007   Accuracy: 100.00%
running time: 0:00:07.471027
```

CovNet Performance

In our CNN, we say that training loss steadily decreases until around epoch 13, where it reaches 100% accuracy. However, validation loss continues to decrease. This CNN also has an insanely short runtime, taking ~7.5 seconds.

```
Epoch: 0/300    Training Loss: 2.303    Validation Loss: 2.303  Accuracy: 10.18%
Epoch: 10/300   Training Loss: 1.853    Validation Loss: 1.859  Accuracy: 23.76%
Epoch: 20/300   Training Loss: 1.589    Validation Loss: 1.581  Accuracy: 38.91%
Epoch: 30/300   Training Loss: 1.487    Validation Loss: 1.252  Accuracy: 50.79%
Epoch: 40/300   Training Loss: 1.075    Validation Loss: 1.158  Accuracy: 53.45%
Epoch: 50/300   Training Loss: 0.916    Validation Loss: 1.001  Accuracy: 59.39%
Epoch: 60/300   Training Loss: 0.928    Validation Loss: 1.067  Accuracy: 57.94%
Epoch: 70/300   Training Loss: 0.948    Validation Loss: 0.864  Accuracy: 65.45%
Epoch: 80/300   Training Loss: 0.692    Validation Loss: 0.801  Accuracy: 69.21%
Epoch: 90/300   Training Loss: 0.720    Validation Loss: 0.676  Accuracy: 72.61%
Epoch: 100/300  Training Loss: 0.692    Validation Loss: 0.826  Accuracy: 68.36%
Epoch: 110/300  Training Loss: 0.478    Validation Loss: 0.649  Accuracy: 75.52%
Epoch: 120/300  Training Loss: 0.511    Validation Loss: 0.592  Accuracy: 78.06%
Epoch: 130/300  Training Loss: 0.514    Validation Loss: 0.514  Accuracy: 81.09%
Epoch: 140/300  Training Loss: 0.436    Validation Loss: 0.462  Accuracy: 81.94%
Epoch: 150/300  Training Loss: 0.501    Validation Loss: 0.459  Accuracy: 82.42%
Epoch: 160/300  Training Loss: 0.443    Validation Loss: 0.481  Accuracy: 81.45%
Epoch: 170/300  Training Loss: 0.499    Validation Loss: 0.524  Accuracy: 79.76%
Epoch: 180/300  Training Loss: 0.314    Validation Loss: 0.489  Accuracy: 82.79%
Epoch: 190/300  Training Loss: 0.267    Validation Loss: 0.349  Accuracy: 87.64%
Epoch: 200/300  Training Loss: 0.298    Validation Loss: 0.360  Accuracy: 88.48%
Epoch: 210/300  Training Loss: 0.247    Validation Loss: 0.308  Accuracy: 88.97%
Epoch: 220/300  Training Loss: 0.197    Validation Loss: 0.280  Accuracy: 91.15%
Epoch: 230/300  Training Loss: 0.128    Validation Loss: 0.349  Accuracy: 88.61%
Epoch: 240/300  Training Loss: 0.251    Validation Loss: 0.240  Accuracy: 92.61%
Epoch: 250/300  Training Loss: 0.207    Validation Loss: 0.218  Accuracy: 93.21%
Epoch: 260/300  Training Loss: 0.182    Validation Loss: 0.226  Accuracy: 92.12%
Epoch: 270/300  Training Loss: 0.213    Validation Loss: 0.171  Accuracy: 95.03%
Epoch: 280/300  Training Loss: 0.194    Validation Loss: 0.220  Accuracy: 92.12%
Epoch: 290/300  Training Loss: 0.137    Validation Loss: 0.199  Accuracy: 92.36%
Epoch: 300/300  Training Loss: 0.231    Validation Loss: 0.214  Accuracy: 93.21%
running time: 0:18:05.768809
```

SqueezeNet Performance

In our SqueezeNet performance, it takes many more epochs due to the necessary choice of a smaller learning rate. It peaks at a 95% accuracy, but seems to average an accuracy below that. We can see both the training loss and the validation loss drop with diminishing returns towards the end, but the accuracy stays in the ~92% range. We suspect some overfitting is happening towards the last 50 epochs, but that accuracy is still quite solid for any machine learner. This training takes 300 epochs and mini batches of 500. Because of this, it takes significantly longer, at 18 minutes to fully run.

7. Discussion and Conclusion

During the project, we realized the importance of having a dependable dataset. If we were to do it again, we would ideally have larger dataset of hands with different shades, different angles, possible occlusion, and just more variance overall. We attempted to mitigate the problem by adding reflections and effectively doubling our dataset, but we recognize there are better ways to add variance.

The Convolutional Network we created worked very well, and we can see why this is a popular choice for image recognition and classification problems. We were surprised with how easily it reached 100% validation accuracy. It reached 100% well within 20 epochs, with a runtime of ~7.5 seconds.This makes us a little suspicious of our dataset more than anything, but it could also be a testament to the strength of convolutional neural networks.

Our SqueezeNet, on the other hand, was able to reach an 89% accuracy. Because SqueezeNet architectures need a smaller learning rate, it took 300 epochs and ~17 minutes to reach that accuracy. The network works well, especially for a network designed to be so compact that it can be run on phones. It can score decently with a much smaller network, given enough epochs of training. However, while SqueezeNets have proven to be a viable lightweight option, we would still say that SqueezeNets are easily outclassed by Convolutional Neural Networks in terms of sheer performance.

Moving forward, we would like to see if a different configuration of fire modules would work better for this dataset. It took a long time to get our own SqueezeNet up and running, but if we had more time, we would like to see what parameters suit this network best, and see if we can build a better SqueezeNet configuration that takes less time or less epochs to reach ~90% accuracy.