Project 2: Traffic Sign Classifier Using Deep learning

Introduction:

In this project, which is the second project assignment in Udacity's Self Driving Car Nanodegree program we are going to create a convolutional neural network(CNN) classifier that can detect 43 different traffic signs from German Traffic Sign Detection Benchmark (GTSRB) data set.

The entire project is based on Google's deep learning framework, Tensorflow and written in Python using the help of Jupyter notebook.

The task at hand is to help students understand the concepts surrounding Deep Learning accompanied with a plethora of supporting lectures and exercise material leading up to the project.

Although a vast and expansive field with a steep learning curve, I found myself adjusting quite comfortably with the curriculum having already taken part in other machine learning courses such as the one by Andrew NG on Coursera.

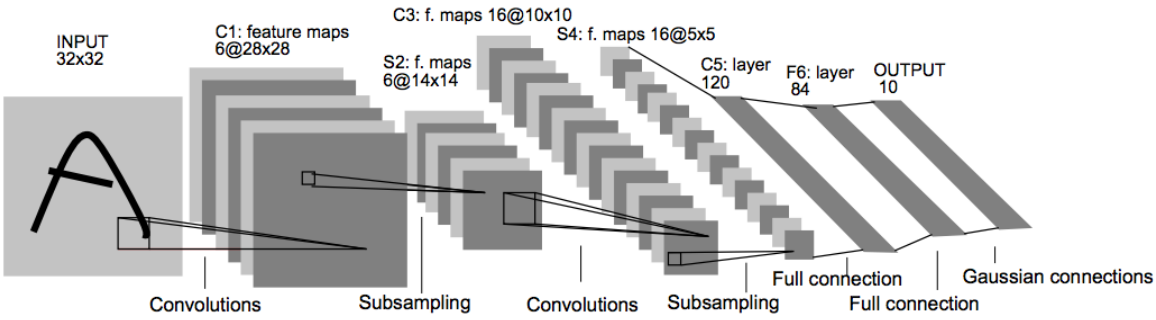
Procedure:

The general guidelines prescribed to complete the project were as follows:

0. Load the Data
1. Dataset Summary and Exploration
2. Design and Test a Model Architecture
3. Test a Model on New Images

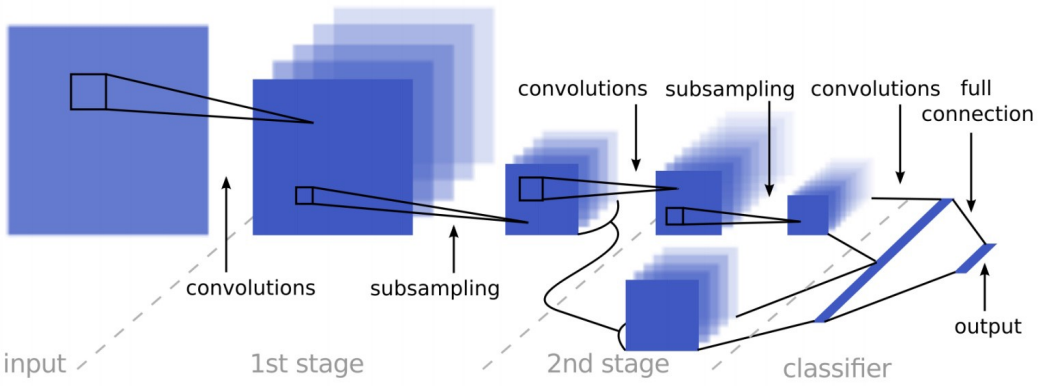
When you start with the GTSRB dataset, the training and testing data is loaded into a numpy array with a shape of (None, 32, 32, 3), where none represents the number of photos in the dataset. In this example, training data is held in the X_train, y_train, and testing in X_test, y_test.

The supporting material for the project introduces the LeNet architecture by Yann LeCun to help implement CNN using Tensorflow. The LeNet model was developed in 1998 to classify images of handwritten digits zero through nine. Even though the task at hand is to classify traffic signs, a computationally much harder task than classifying digits from 0 through 9, the model was suggested to be a good starting point to understand CNN as it is one of the most famous models known. Still the model performed very well — achieving 96.5% validation accuracy right out of the box.



- Original LeNet Model

The project rubric also recommended reading a journal article written by Pierre Sermanet and Yann LeCun describing a neural network-based approach specifically for classifying traffic signs. The Sermanet-Lecun model, although very similar to the original model seemed to have a lot of the specifics regarding the architecture missing but I was able to obtain crucial information on other methodologies to improve accuracy of the model. This included data pre-processing methods such as converting the image to grayscale, applying normalization to the images and finally explaining the analogy of augmenting data by copying and slightly modifying underrepresented classes for better classification.



- Sermanet-Lecun Model

My dataset pre-processing consisted of:

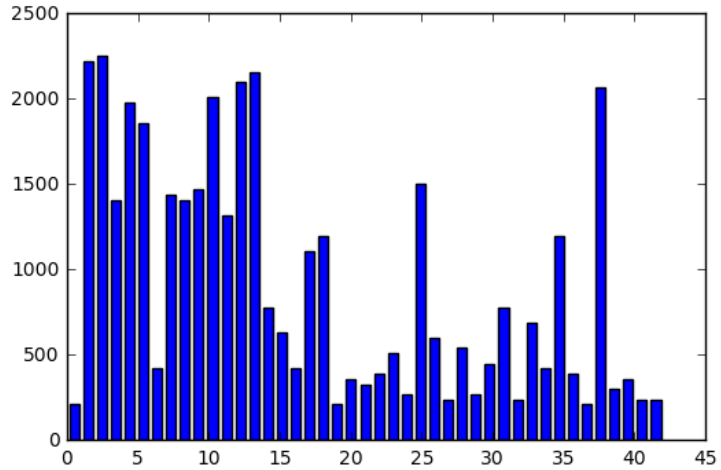
1. Converting to grayscale - This worked well for Sermanet and LeCun as described in their traffic sign classification article. It also helps to reduce training time, which was nice when a GPU wasn't available.
2. Next I applied histogram equalize, a method in image processing of contrast adjustment using the image's histogram.
3. Finally I normalized the data to the range (-1,1). This was done using the line of code X_train_normalized = (X_train - 128)/128. The resulting dataset mean wasn't exactly zero which is ideal, but it was reduced from around 82 to roughly -0.65.

- RGB VS Grayscale representation

- Normalized vs Original representation

Out of the lot, data augmentation was by far the most time-intensive and computationally intensive strategy to carry out, but it was also said to be one of the most important to achieve stellar accuracy. To make life easier, I used an Amazon Web Service GPU instance to run these computationally intensive tasks rather than running on my local CPU which is comparatively much slower.

Prior to data augmentation, the number of data points (images) per class looked like this:



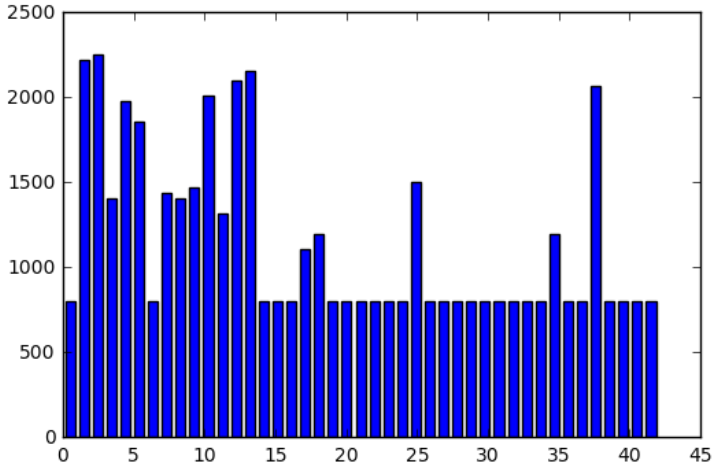
- Prior to data augmentation

The minimum number of data points out of the 43 classes turned out to be 210.

As some of the classes are represented way more than the rest, in extreme cases almost bya factor of ten. This is one of biggest problems to be overcome while running a machine learning algorithm. Due to this lack of balance in the data, the output will tend to become biased toward the classes with more data points. One way to tackle this would be to throw out data from the more heavily represented classes, but doing so gets rid of perfectly good data which could be critical in classifying the images. The alternate is generating more data points for the classes that are under-represented so as to reach a threshold value.

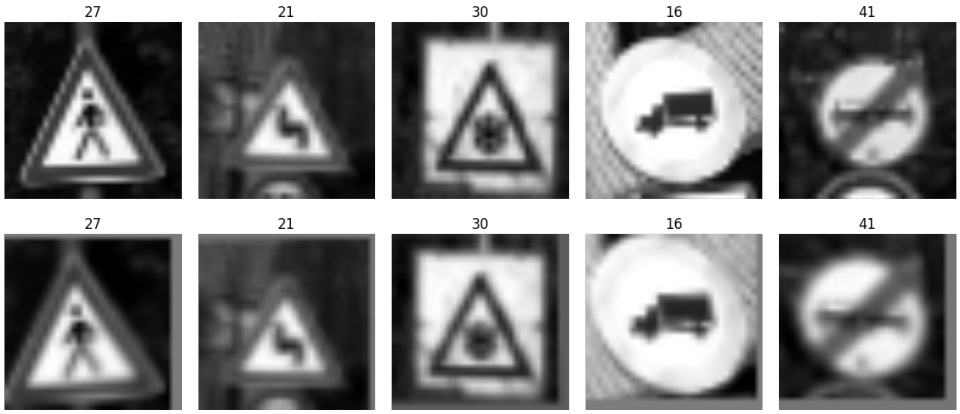
The approach I chose was to generate augmented data in the form of “jittered” images, which is nothing but applying small and random shifts to the original image (horizontal, vertical), scaling, warping and random adjustments in brightness. The final aim was to generate “jittered” copies until the threshold sample size of 800 was achieved by looping through the entire data through each class and implementing this approach to every class below the threshold size.

This led to the distribution of data points across the 43 classes to appear as follows:



- Post data augmentation

Here are a few outputs after implementing the process of jittering:



- “Jittered” Images

I then used the SciKit Learn’s shuffle and train_test_split function to shuffle the dataset and to create a validation set out of the training set. I used 20% of the testing set to create the validation set

Next is the step of coding up the model. As discussed before, I began by implementing the same architecture from the LeNet Lab, with no changes since my dataset is in grayscale. This model worked quite well to begin with (~96% validation accuracy). The layers are set up like this:

1. 5x5 convolution (32x32x1 in, 28x28x6 out)
2. ReLU
3. 2x2 max pool (28x28x6 in, 14x14x6 out)
4. 5x5 convolution (14x14x6 in, 10x10x16 out)
5. ReLU
6. 2x2 max pool (10x10x16 in, 5x5x16 out)

7. 5x5 convolution (5x5x6 in, 1x1x400 out)
8. ReLu
9. Flatten layers from numbers 8 (1x1x400 -> 400) and 6 (5x5x16 -> 400)
10. Concatenate flattened layers to a single size-800 layer
11. Dropout layer
12. Fully connected layer (800 in, 43 out)

Once this was complete the only task at hand was to tune the so called “knobs” in every machine learning model, the hyper parameters to produce a model with the highest validation accuracy. This approach is regarded the most challenging step when it comes to any machine learning problem as its mostly done in a trial and error fashion. This is very using a fast GPU instance via AWS paid its dividends as I could save precious time trying out many different strategies to improve the performance of the model. Finally, after trying out a bunch combinations of hyper parameters I was able to achieve a validation accuracy over 99%.

I used the Adam optimizer (already implemented in the LeNet model) to train the model with batch size of 32, 60 epochs, learning rate of 0.0005 and dropout rate of 0.5(to avoid overfitting).

Now that I was satisfied with my model I went ahead to evaluate the model against the validation portion of the training data set. This data had been put aside from the beginning because the model can become biased toward the data used to determine validation accuracy as the data could bleed into the output due to repeated testing to ascertain the ideal parameters and could thus lead to an overfitted model. Luckily enough, the testing accuracy turned out to be highly satisfactory at around 95%.

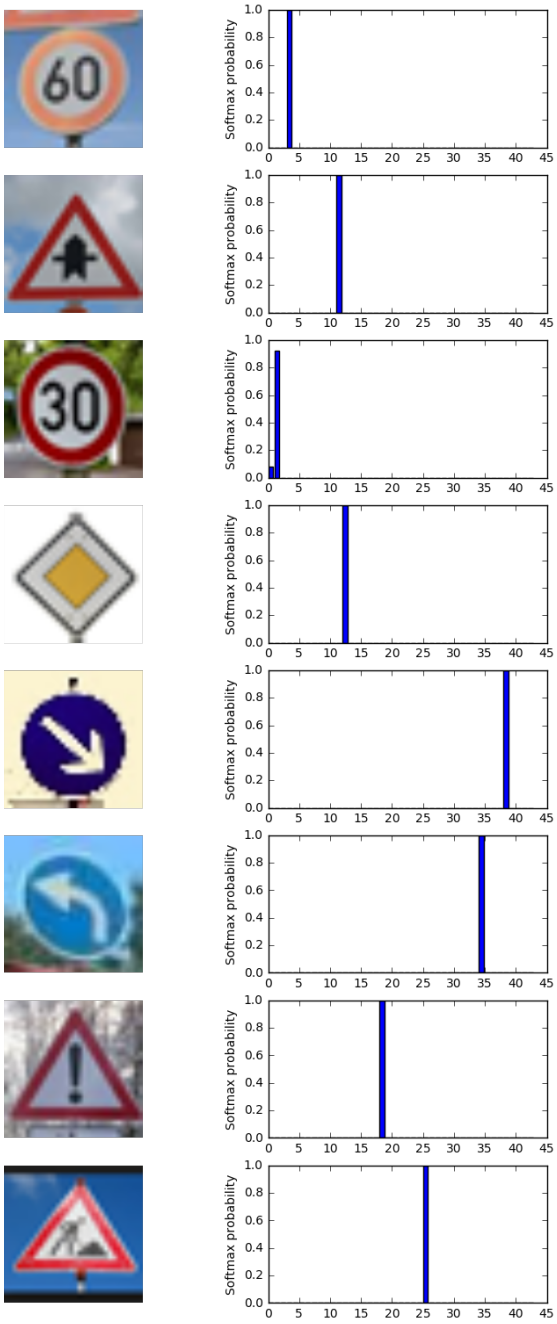
The final task was to test this model on new images taken from the net and check its accuracy. We were instructed to check the accuracy on 5 image chosen at random but preferably ones which differ from each other significantly. I chose the following images:

- My image set obtained from the web

Instead of just showing the predicted class number for each sign, I implemented the example from the original dataset to display the top three predictions including the certainty of each prediction (in parentheses). And here's how the model performed on them:

The model appears to have predicted the new signs perfectly, except for the last image. This is a good sign that the model performs well on real-world data. And while it's reasonable to assume that the accuracy would not remain so high given more data points, judging by the low fidelity of a number of images in the training dataset it's also reasonable to assume that if the real-world data were all as easily distinguishable as the eight images chosen that the accuracy would remain very high.

I then went on to display the model's softmax probabilities to visualize the certainty of its predictions using tensorflow's "tf.nn.top_k" function.



Conclusion:

At this point in time there are still a few things I could experiment with. For example, I never ran colour images through the model because I was trying to save some time and processing power and the grayscale images performed reasonably well as indicated by the paper by Sermanet-LeCun. I didn't play around much with the jittering parameters either. I could also try implementing other models other than LeNet model and check its accuracy. The possibilities are endless but for now it's time to move on to the next task – Transfer learning and behavioral cloning.