Self-driving car engineer nano-degree

Project 2: Traffic sign classifier

The goals/ steps of this project are the following:

1) Dataset Exploration.
    a. Dataset Summary.
    b. Exploratory Visualization.
2) Design and Test a Model Architecture.
    a. Preprocessing.
    b. Model Architecture.
    c. Model Training.
    d. Solution Approach.
3) Test a Model on New Images.
    a. Acquiring New Images.
    b. Performance on New Images.
    c. Model Certainty – Softmax Probabilities.

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

In order for the steps to be successful I first load in the data from the accompanying pickle files.

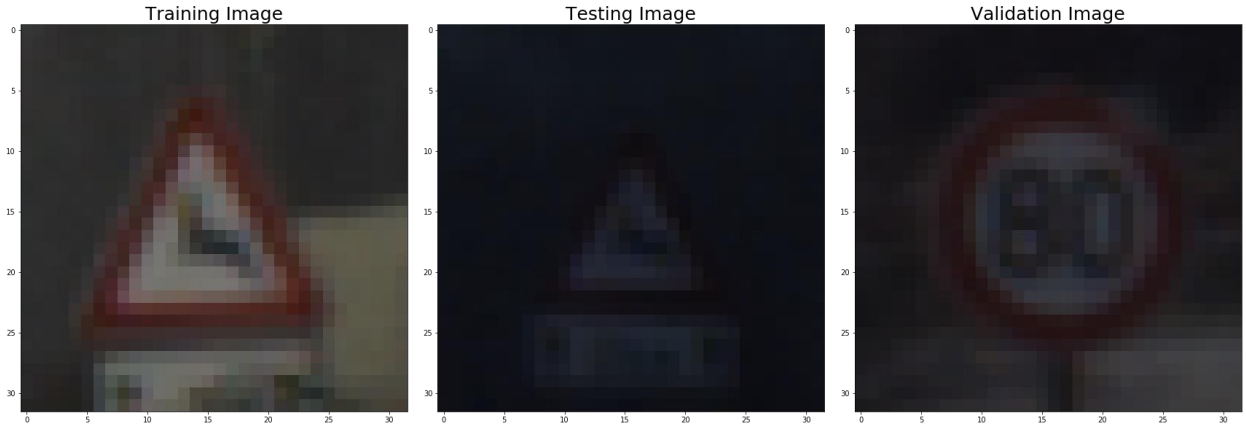
1) Dataset Exploration.
   First I obtain the length of the training data, the validation data, and the test data. Next I obtain the shape of an image from the training data. Then I calculate the amount of unique classes/labels there are in the training data and the testing data. I then use these values to calculate the difference between the two datasets and add the difference to the amount of training data in order to get the total number of classes/labels. I then print out the results:

   Number of training examples = 34799.
   Number of validation examples = 4410.
   Number of testing examples = 12630.
   Image data shape = (32, 32, 3).
   Number of classes = 43.

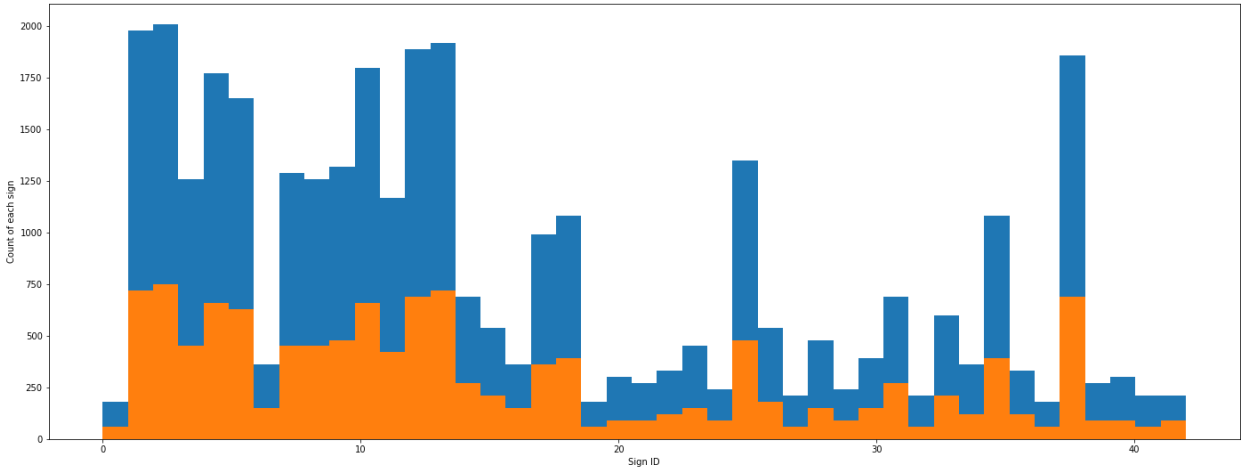
   Next I visualize the data though displaying one image from each dataset (Training, Testing, and Validation). To further explore the datasets I make use of matplotlib's *hist* function to display the data in a histogram. The histogram displays the amount of times each sign appears in the training and testing datasets.

   The visualization results are:

Images:



Histogram:



2) Design and Test a Model Architecture.

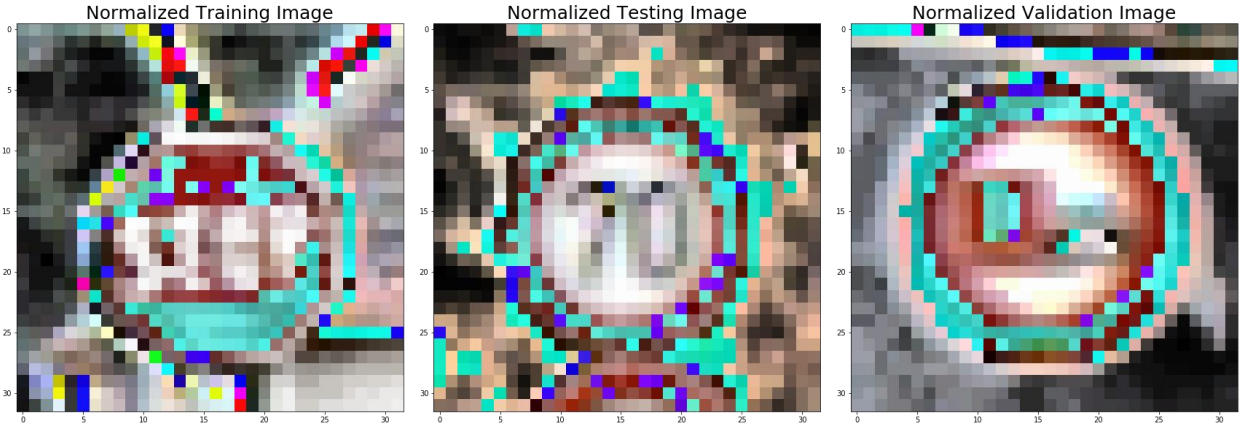
Pre-processing the data:

In order to pre-process the data I created the *norm* function. This function takes in a single image then converts it from BGR color space to HLS color space. Next it uses open cv's *equalizeHist* function in order to produce a histogram equalized image on the L channel. The function then converts the equalized image into the RGB color space in order to the convert it into the HSV color space. Here it uses open cv's equalizeHist function on the V

channel and converts the equalized image back into RGB color space.

Next the function normalizes the pixels on each of the B, G, and R channels, and then merges them back into the image.

An example of the normalized images:



Model Architecture:

I decided to implement the *LeNet* architecture with a μ (Mu) of zero and ∑ (Sigma) of 0.1. The architecture consists of:

- **Layer 1: Convolutional Layer.**
  Takes an input image of shape 32 X 32 X 3 and output an image of shape 28 X 28 X 6.
- **Activation.**
  Computes the rectified linear from Layer 1's output.
- **Pooling.**
  Performs max pooling on the input of shape 28 X 28 X 6 and outputs a shape of 14 X 14 X 6.
- **Layer 2: Convolutional Layer.**
  Takes the output from Layer 1 and output an image of shape 10 X 10 X 16.

- **Activation.**
  Computes the rectified linear from Layer 2's output.
- **Pooling.**
  Performs max pooling on the input of shape 10 X 10 X 16 and outputs a shape of 5 X 5 X 16.

- **Flatten.**
  Flattens the output from the pooling step into a 400-d vector.
- **Layer 3: Fully Connected Layer.**
  Takes in an input vector of 400-d and outputs a 120-d vector.
- **Activation.**
  Computes the rectified linear from Layer 3's output.
- **Layer 4: Fully Connected Layer.**
  Takes an input vector of 120-d and outputs a vector of 84-d.
- **Activation.**
  Computes the rectified linear from Layer 4's output.
- **Layer 5: Fully Connected Layer.**
  Takes an input vector of 84-d and outputs a 10-d vector.

Model Training:

- **Features and Labels.**
  Here I create tensorflow placeholders.
- **Training Pipeline.**
  Here I set the learn rate to 0.0009 that is uses with tensorflow's tf.train.AdamOptimizer. And use tensorflow's *tf.nn.softmax_cross_entropy_with_logits* function to compute the softmax cross entropy between the logits and labels.

Next I use tensorflow's *tf.reduce_mean* function to compute the mean of elements across the dimensions of a tensor. Lastly I use tensorflow's *minimize* function in order to minimize loss.

- **Model Evaluation.**
  Here I evaluate the accuracy of the model through the *evaluate* function.
- **Train the Model.**
  Here I train the model for 30 Epoch's and a batch size of 64. Then I display the accuracy of the model on the validation dataset and the training dataset for each epoch, with a final validation accuracy of 95.5% and training accuracy of 99.9%.

After training the model I compute the accuracy of the testing dataset which is displayed as having an accuracy of 92.2%.

3) Test a Model on New Images.
   I searched and downloaded 5 German traffic sign images from the web and stored them in the web_signs folder. These signs are:
   I. Priority right-of-way at the next intersection.
      The model should have no problem in classifying this particular sign as it is in the center of the image facing front.
   II. Stop.
      The model should have no problem in classifying this particular sign as it is in the center of the image facing the front.

III. Double Curve.
The model would most likely struggle to classify this image as it is smaller than the rest of the signs and is not facing the front 100%.

IV. Bumpy road.
The model could potentially struggle to identify this sign due to it being a bit darker, and the bolts is a bit more visible.

V. Bicycles Crossing.
The model should not have any problems classifying this sign as it is front facing and clear.

To load the images I resize them to 32 X 32 and then display them. Next I use the *norm* function to normalize the new images. Then I run the predictions on the new images and output the accuracy for these images along with the accuracy of the testing dataset.

Which is a test accuracy of 92.2% and an accuracy of 60.0% on the images from the web.

Softmax:

The top five softmax probabilities for the predictions on the German traffic signs found on the web has been computed and displayed as:

TopKV2(values=array([[1.00000000e+00, 6.03900885e-09, 2.79966328e-11,
1.59454524e-12, 9.20706094e-15],
[9.99856830e-01, 8.36526087e-05, 4.13126691e-05,
1.68728729e-05, 8.63153502e-07],
[9.54870820e-01, 3.65681387e-02, 1.73558143e-03,
1.73236092e-03, 1.32001634e-03],
[9.99935269e-01, 6.47544075e-05, 5.93188389e-12,
4.80558215e-12, 1.91898060e-12],
[9.99909163e-01, 5.86670358e-05, 3.21493353e-05,
4.45230297e-09, 4.06874889e-09]], dtype=float32), indices=array([[11, 30, 21, 1
9, 26],
[14, 13, 15, 29, 1],
[38, 34, 25, 40, 36],
[22, 25, 20, 1, 26],
[25, 29, 22, 13, 26]]))

This indicates that the model:

1. Correctly classified the first (“Priority”) sign.
2. Correctly classified the second (“Stop”) sign.
3. Wrongly classified the third(“Double curve”) sign, and instead classified it as a “Keep right” sign.
4. Correctly classified the fourth (“Bumpy Road”) sign.
5. Wrongly classified the fifth(“Bicycles crossing”) sign, and instead classified it as an “Road work” sign.

Conclusion

I choose to use the LeNet5 model architecture due to it accepting images of shape 32 X 32 which is the shape of the images contained within the pickle files. As well as the “short” amount of processing time that the network requires/uses.

From the results obtained I found that the network is perfectly capable of classifying the German traffic signs.