Traffic Sign Recognition

Creig Cavanaugh – February 2017

Build a Traffic Sign Recognition Project

The goals / steps of this project are the following:

- Load the data set (see below for links to the project data set)
- Explore, summarize and visualize the data set
- Design, train and test a model architecture
- Use the model to make predictions on new images
- Analyze the softmax probabilities of the new images
- Summarize the results with a written report

Rubric Points

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

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf. You can use this template as a guide for writing the report. The submission includes the project code.

You're reading it! The submission contains the project code and web images.

Data Set Summary & Exploration

1. Provide a basic summary of the data set and identify where in your code the summary was done. In the code, the analysis should be done using python, numpy and/or pandas methods rather than hardcoding results manually.

The code for this step is contained in the second code cell of the IPython notebook.

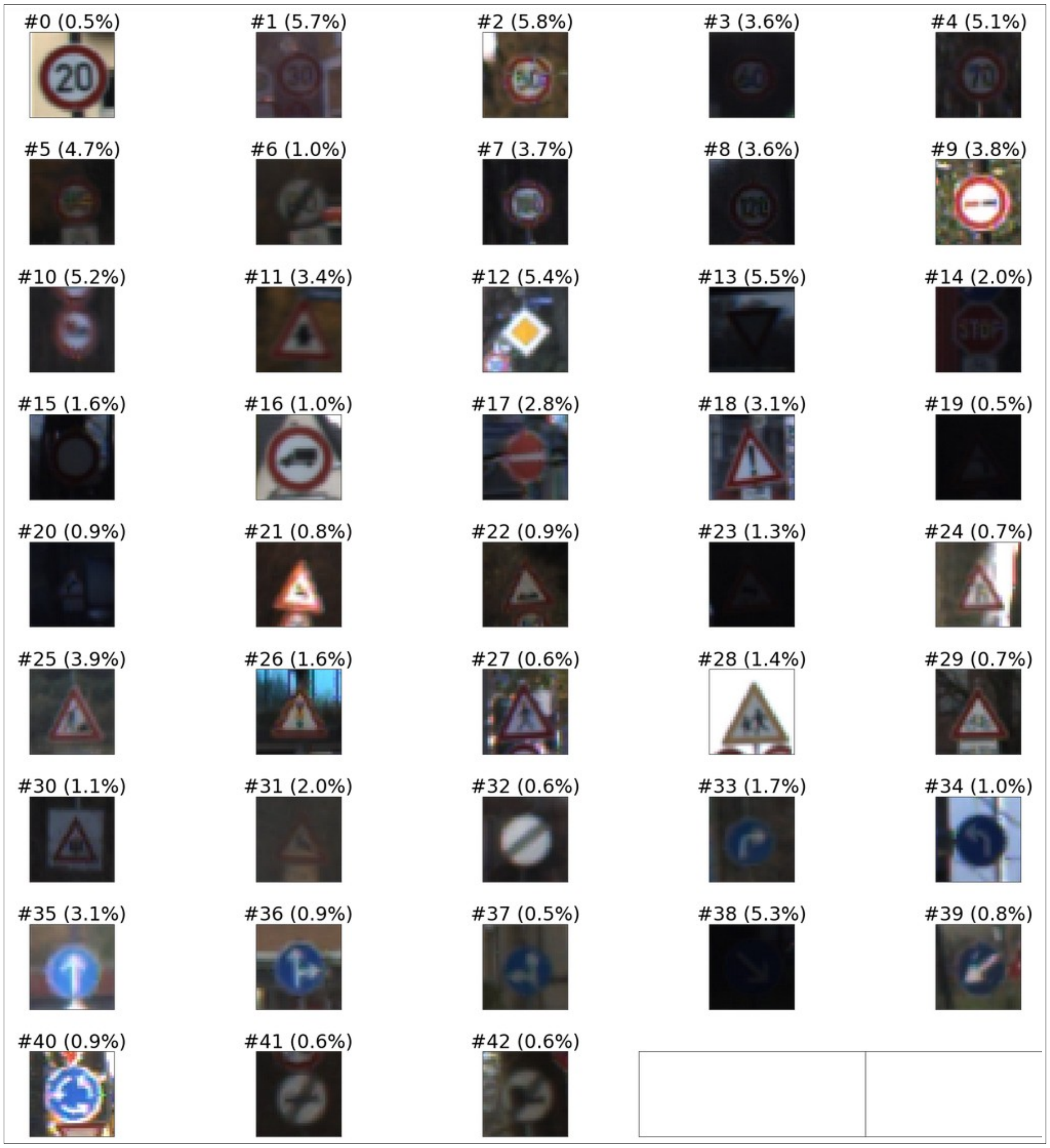
I used the pandas library to calculate summary statistics of the traffic signs data set:

- Training Set: 34799 samples
- Validation Set: 4410 samples

- Test Set: 12630 samples
- Image data shape = (32, 32, 3)
- Number of classes = 43

2. Include an exploratory visualization of the dataset and identify where the code is in your code file.

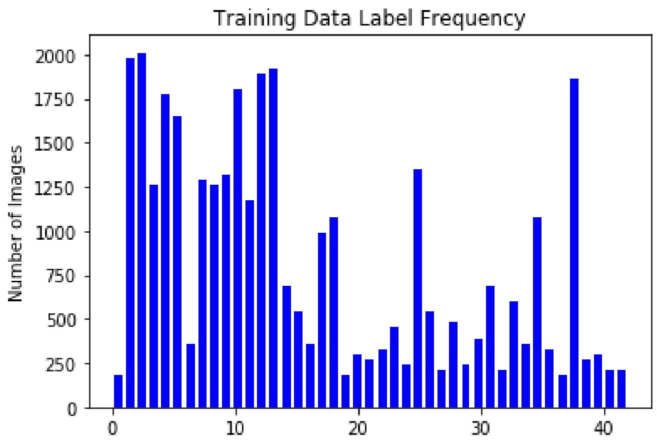
Here is an exploratory visualization of the data set. It contains one image of each type of image class, and provides ClassId label as well as calculated percentage of the training set which contains the specified image type.

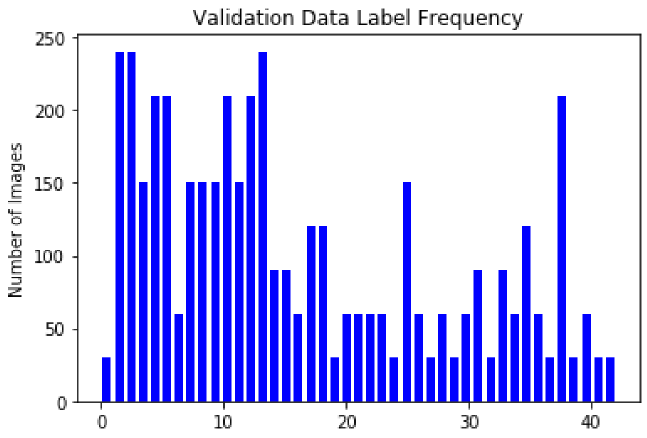


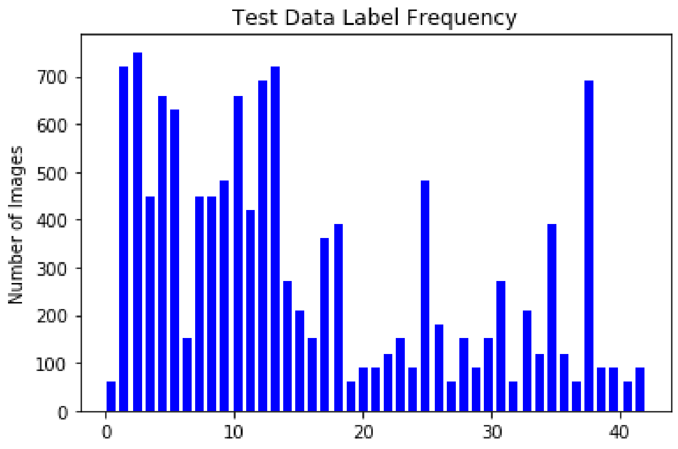
Also generated in the jupyter notebook are histograms which provide a visualization of the image class distribution, one each for the training, validation and test data sets. From the visualization, it appears most of the speed limit signs are well represented, as well as the yield, priority and keep right signs. One could hypothesize the classifier will be able to identify traffic signs more accurately with larger amounts of training images available.

Although I did not create additional image data, calculating the percentage of data for each class of image helps identify areas where it would be beneficial to add additional image. In this case, I might want to add 20km/h speed limit signs to the image dataset (ClassId:0), since it only encompasses 0.5% of the overall dataset (for reference, the average for all images is 2.3%).

An additional observation is the histograms show the image class types appears evenly distributed across the training, validation and test data sets.







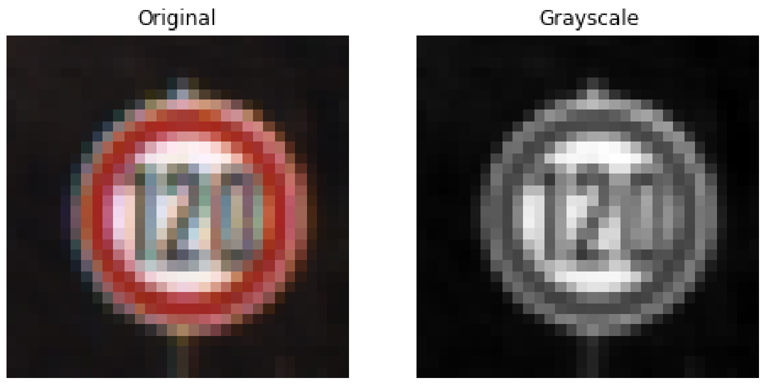
Design and Test a Model Architecture

1. Describe how, and identify where in your code, you preprocessed the image data. What techniques were chosen and why did you choose these techniques? Consider including images showing the output of each preprocessing technique. Pre-processing refers to techniques such as converting to grayscale, normalization, etc.

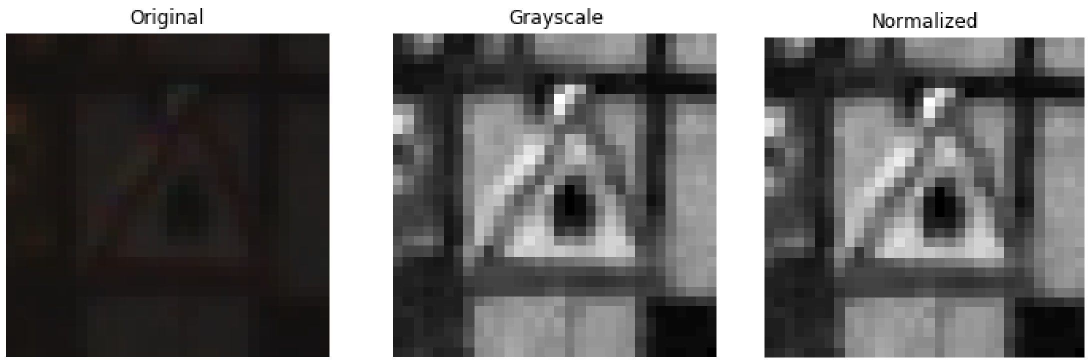
The code for this step is contained in the fourth code cell of the IPython notebook.

As a first step, I decided to convert the images to grayscale based on the accuracy improvements Sermanet and LeCun describe in their article “Traffic sign recognition with multi-scale convolutional networks”.

Here is an example of a traffic sign image before and after grayscaling.



After implementing normalization and testing the model with the normalized data, I didn’t see the validation accuracy change much after testing multiple times. It appears as part of the grayscale process normalization occurs to some extent. As seen below, there is a large improvement in image detail when converted from color to grayscale, yet there is no noticeable difference between the grayscale and normalized image.   I wonder if normalizing would have a greater effect to color images?



2. Describe how, and identify where in your code, you set up training, validation and testing data. How much data was in each set? Explain what techniques were used to split the data into these sets.

The code for splitting the data into training and validation sets is contained in the ninth code cell of the IPython notebook.

To cross validate my model, I randomly split the training data into a training set and validation set. I did this by using train_test_split from sklearn, and used a 20% test size.

My final training set had 31367 number of images. My validation set and test set had 7842 and 12630 number of images.

3. Describe, and identify where in your code, what your final model architecture looks like including model type, layers, layer sizes, connectivity, etc.) Consider including a diagram and/or table describing the final model.

The code for my final model is located in the tenth cell of the ipython notebook.

My final model is primarily based on the original LeNet CNN developed by LeCun, with the addition of implementing dropout. The model consists of the following layers:

Layer	Description
Input	32x32x1 Grayscale image
Convolution 5x5	32x32x1 in, 1x1 stride, valid padding, outputs 28x28x6
RELU	
Max pooling	28x28x6 in, 2x2 stride, valid padding, outputs 14x14x6
Convolution 5x5	14x14x6 in, 1x1 stride, valid padding, outputs 10x10x16
RELU	
Max pooling	10x10x16 in, 2x2 stride, valid padding, outputs 5x5x16
Flatten	5x5x16 to 400
Fully connected	400 in, 120 out
RELU	
Dropout	Dropout with probability of keeping 90%
Fully connected	120 in, 84 out
RELU	
Dropout	Dropout with probability of keeping 90%
Fully connected	84 in, n_classes (43) out

4. Describe how, and identify where in your code, you trained your model. The discussion can include the type of optimizer, the batch size, number of epochs and any hyper parameters such as learning rate.

The code for training the model is located in the eleventh cell of the ipython notebook.

To train the model, I used the Adam optimizer, which is the optimizer used in the base LeNet implementation. Adam is similar to Gradient Decent, but has the added benefit of momentum which effectively increases step size towards minimum error. According to Yoshua Bengio in his paper "Practical Recommendations for Gradient-Based Training of Deep Architectures", momentum "removes some of the noise and oscillations that gradient descent has, in particular in the directions of high curvature of the loss function."

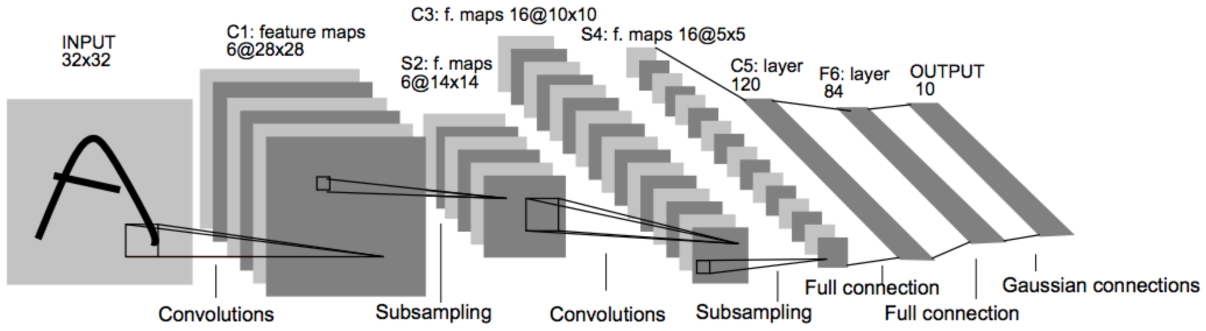
I iterated across various hyper parameter combinations in search of better validation accuracy. Below are some of the combinations which led to my final hyper parameter values.

Val Accuracy	Batch Size	Epochs	Learning Rate	Dropout	Sigma
0.928	128	30	0.0005	0.9	0.08
0.914	128	30	0.0005	0.9	0.1
0.902	128	30	0.0005	0.8	0.08
0.893	256	30	0.0005	0.9	0.08
0.851	64	30	0.0001	0.75	0.08
0.842	128	20	0.0001	1	0.08
0.798	64	10	0.0001	1	0.08
0.783	64	30	0.001	0.5	0.08
0.773	128	10	0.0001	1	0.08
0.632	128	20	0.01	0.75	0.08
0.485	256	20	0.0001	0.75	0.08
0.348	256	30	0.0001	0.5	0.08
0.256	128	20	0.0001	0.5	0.08
0.181	256	10	0.0001	0.75	0.08
0.054	128	30	0.0005	0.9	0.2
0.054	256	10	0.01	0.75	0.08

5. Describe the approach taken for finding a solution. Include in the discussion the results on the training, validation and test sets and where in the code these were calculated. Your approach may have been an iterative process, in which case, outline the steps you took to get to the final solution and why you chose those steps. Perhaps your solution involved an already well known implementation or architecture. In this case, discuss why you think the architecture is suitable for the current problem.

The code for calculating the accuracy of the model is located in the eleventh and twelfth cell of the Ipython notebook.

As described above, I started with the LeNet-5 Convolutional Neural Network architecture, added dropout to the model, and iterated through various combinations of hyper parameters, using trial and error. I decided to add dropout in order to increase the robustness of the model and connections, with the hope that it would help with images that may have higher levels of distortion or otherwise be hard to read. I ended up using 75% keep rate, which I feel is a bit higher than I would have liked, but seemed to provide the best results. Perhaps a larger dataset (including more noisy images) would have allowed better results while keeping a 50% dropout rate.



LeNet-5 Architecture

In the end, I increased the number of Epochs to 50 which seemed to provide a good validation accuracy without over fitting.

My final hyper-parameters:

- Batch Size = 128
- Epochs = 50
- Learning rate = 0.0005
- Dropout = 0.75
- mu = 0
- sigma = 0.08

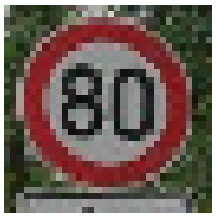
My final model results were:

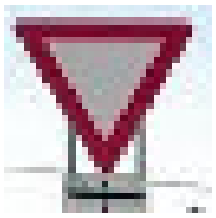
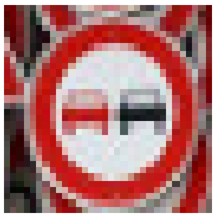
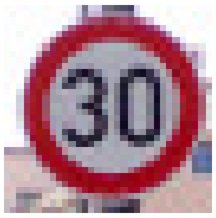
- Validation set accuracy of 97.5%
- Test set accuracy of 92.6%

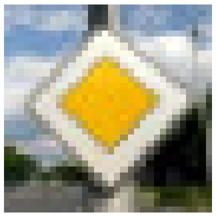
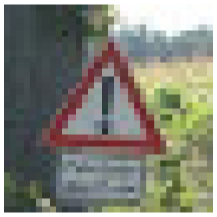
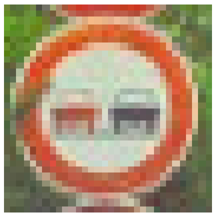
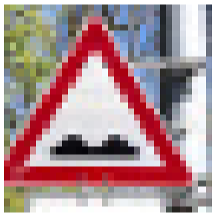
Test a Model on New Images

1. Choose five German traffic signs found on the web and provide them in the report. For each image, discuss what quality or qualities might be difficult to classify.

Here are eight German traffic signs that I found on the web:



The sixth image (general caution) might be difficult to classify because there is a fair amount of background surrounding the sign, and thus less portion of the image contains sign information. Also, the eighth image (bumpy road) might be difficult to classify because a small portion of the sign is cropped off, although I think this should still not be a major problem for the classifier.

2. Discuss the model's predictions on these new traffic signs and compare the results to predicting on the test set. Identify where in your code predictions were made. At a minimum, discuss what the predictions were, the accuracy on these new predictions, and compare the accuracy to the accuracy on the test set.

The code for making predictions on my final model is located in the sixteenth cell of the Ipython notebook.

Here are the results of the prediction:

Image	**Prediction**
80 km/h	50 km/h
Yield	Yield
No Passing	No Passing
30 km/h	30 km/h
Priority Road	Priority Road
General Caution	General Caution
No Passing	No Passing
Bumpy Road	Bumpy Road

The model was able to correctly guess 7 of the 8 traffic signs, which gives an accuracy of 87.5%. This compares favorably to the accuracy on the test set of 75% for the web images.

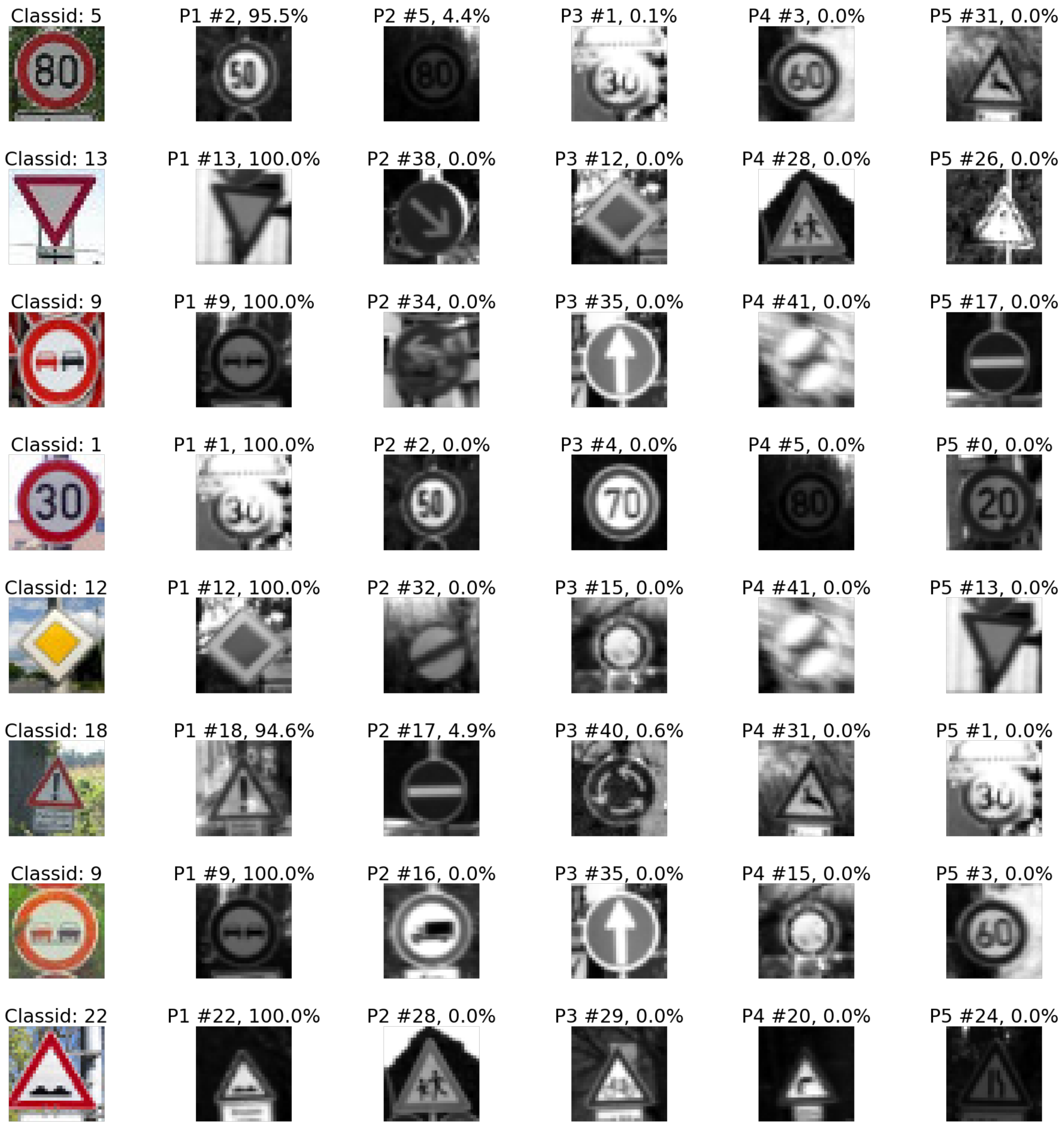
3. Describe how certain the model is when predicting on each of the five new images by looking at the softmax probabilities for each prediction and identify where in your code softmax probabilities were outputted. Provide the top 5 softmax probabilities for each image along with the sign type of each probability.

The code for making predictions on my final model is located in the 17th cell of the Ipython notebook.

Interestingly the one image the classifier did not predict accurately was the 80 km/h sign, which was surprising, since the overall image of the sign was clear and centered. The only thing noticeable is the image appears a little bit grainy, which may have caused the classifier problems. The classifier predicted the 50 km/h sign at 95.5%, and the 80 km/h sign came in second at a distant 4.4% probability. In fact, the top four predictions were speed limit signs.

Another interesting observation is that with the exception of the 80km/h and general caution signs, the classifier predicted the correct sign class with 100.00% certainty. Not sure if this is an indication of over-fitting, but the result seems almost too confident (although correct in these cases).

Here is a detailed graphic of the soft max probabilities:


Classid: 5
P1 #2, 95.5%
P2 #5, 4.4%
P3 #1, 0.1%
P4 #3, 0.0%
P5 #31, 0.0%
Classid: 13
P1 #13, 100.0%
P2 #38, 0.0%
P3 #12, 0.0%
P4 #28, 0.0%
P5 #26, 0.0%
Classid: 9
P1 #9, 100.0%
P2 #34, 0.0%
P3 #35, 0.0%
P4 #41, 0.0%
P5 #17, 0.0%
Classid: 1
P1 #1, 100.0%
P2 #2, 0.0%
P3 #4, 0.0%
P4 #5, 0.0%
P5 #0, 0.0%
Classid: 12
P1 #12, 100.0%
P2 #32, 0.0%
P3 #15, 0.0%
P4 #41, 0.0%
P5 #13, 0.0%
Classid: 18
P1 #18, 94.6%
P2 #17, 4.9%
P3 #40, 0.6%
P4 #31, 0.0%
P5 #1, 0.0%
Classid: 9
P1 #9, 100.0%
P2 #16, 0.0%
P3 #35, 0.0%
P4 #15, 0.0%
P5 #3, 0.0%
Classid: 22
P1 #22, 100.0%
P2 #28, 0.0%
P3 #29, 0.0%
P4 #20, 0.0%
P5 #24, 0.0%