Traffic sign classifier Project.

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

You're reading it! and here is a link to my project code.

Data Set Summary & Exploration

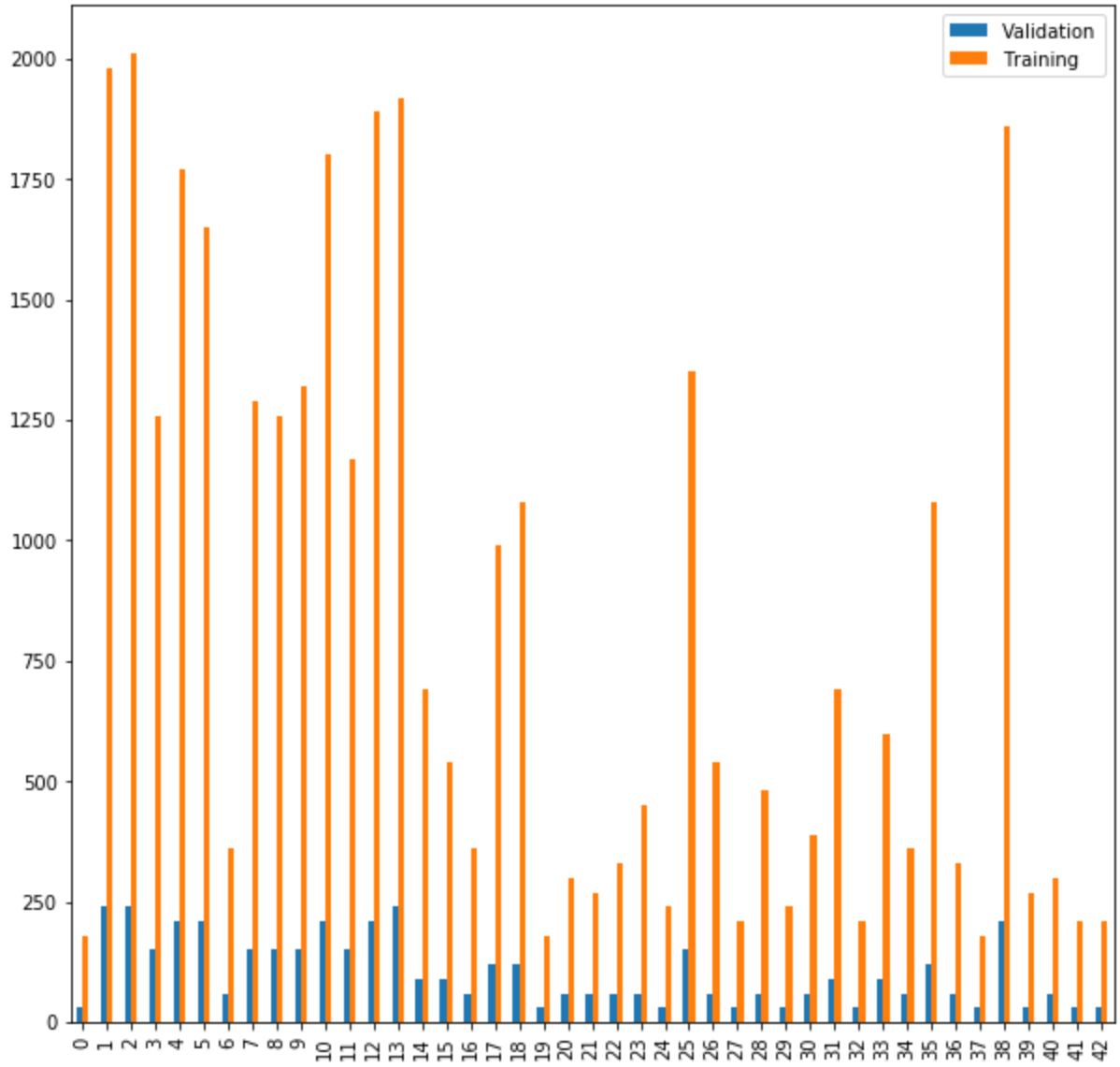
1. Basic summary of the data set

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

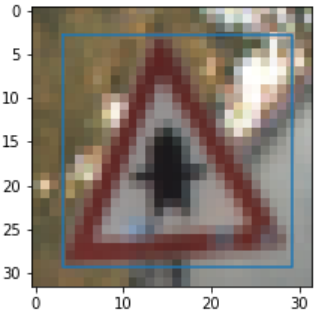
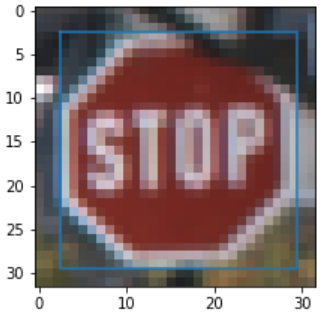
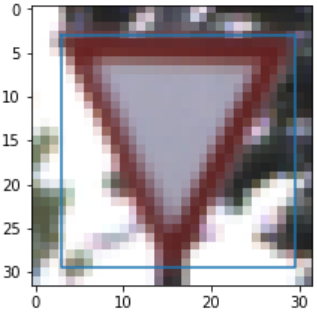
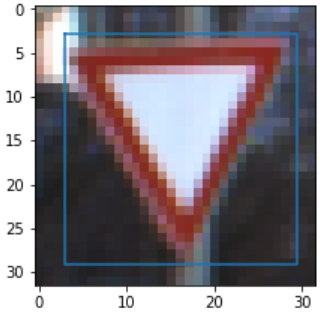
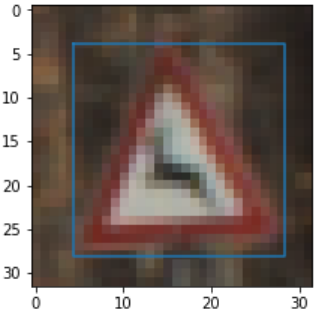
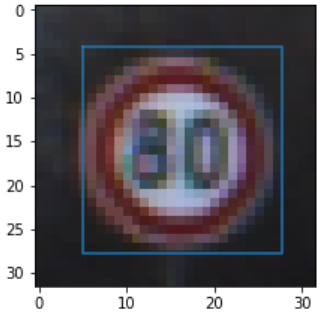
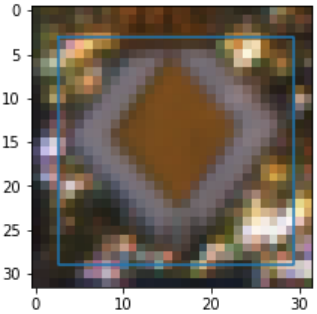
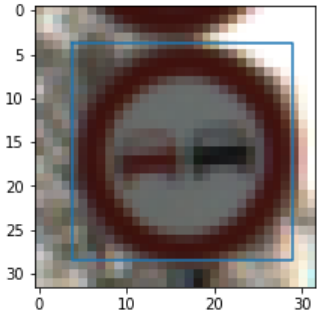
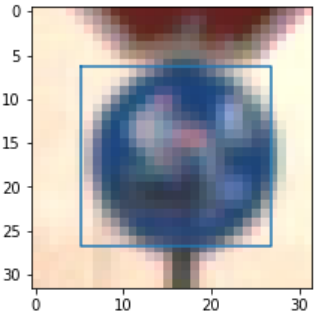
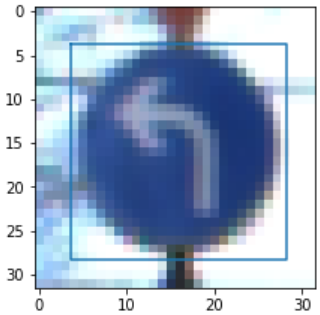
- The size of training set is ?
  - 34799
- The size of the validation set is ?
  - 4410
- The size of test set is ?
  - 12630
- The shape of a traffic sign image is ? Example image 200
  - Size = (53, 55]) , image location is (6,6) - (48,50)
- The number of unique classes/labels in the data set is ?
  - 43

2. Exploratory visualization of the dataset.

This figure show the number of images in the training and validation set per classification.



This Set of images is a random sampling of the training images. I overlayed a bounding box on the area of the image that contains the sign.



STOP




30





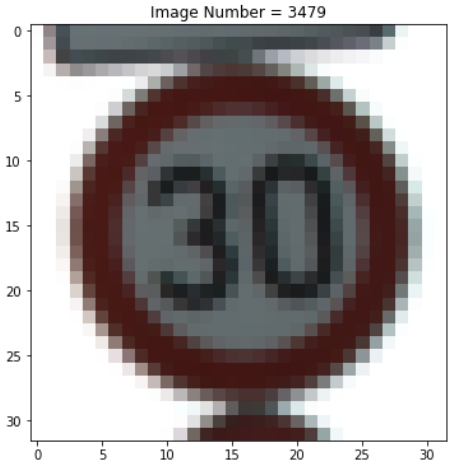
Design and Test a Model Architecture

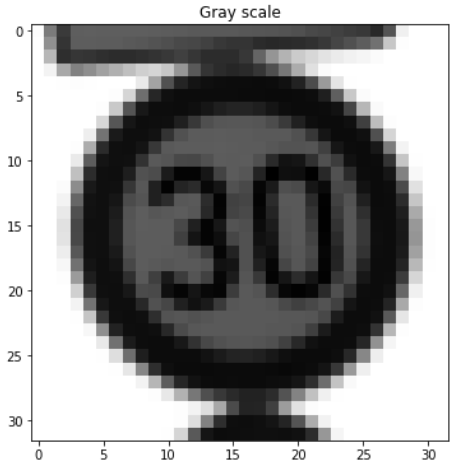
Preprocessing

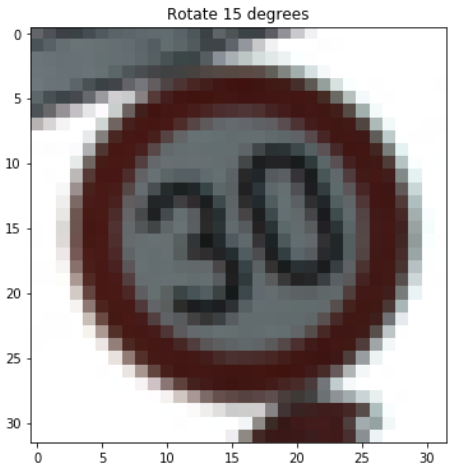
I used a simple normalization procedure as suggested in the notebook, I just converted from the [0-255,0-255,0-255] images to [-1.0-1.0,-1.0-1.0,-1.0-1.0] using the fast val - 128.0 / 128.0

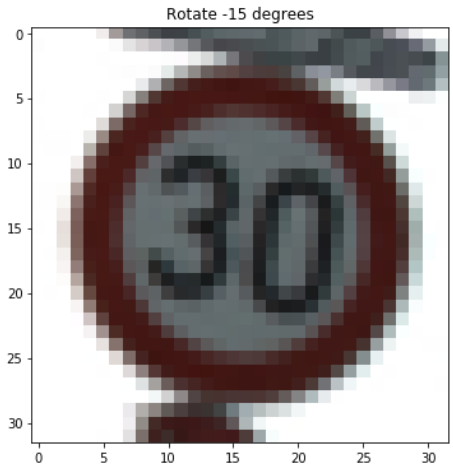
I experimented with gray scaling the images, and I also extended the training image dataset by adding rotations of the image +- 15 degrees. I read this in paper , they also used other techniques to extend the

Here is an example set of changes to a single image (3479) during the preprocessing.









Model architecture.

My final model consisted of the following layers:

Layer 1: Convolutional. Input = 32x32x3. Output = 28x28x9.
Relu Activation.
Pooling. Input = 28x28x9. Output = 14x14x6.

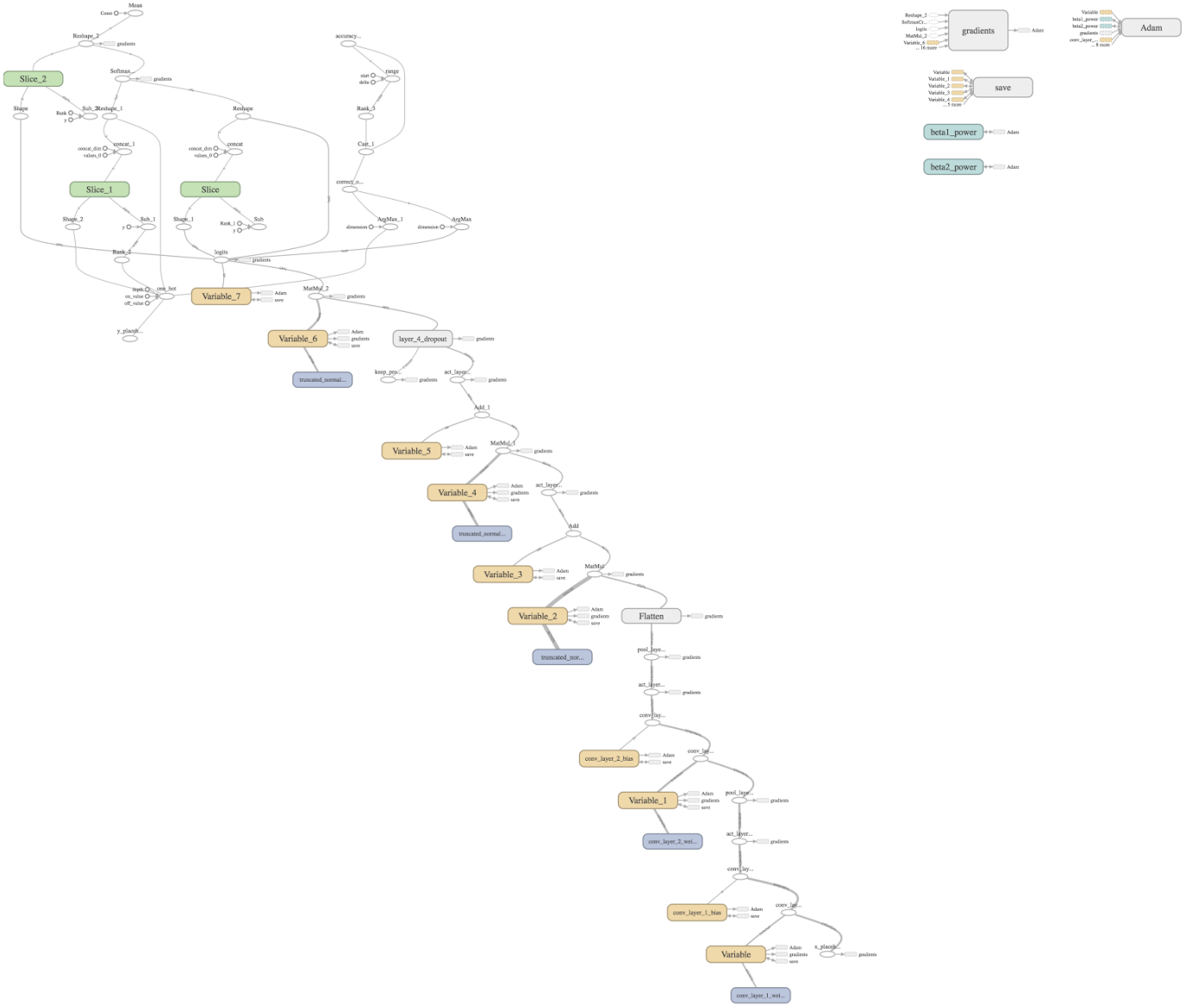
Layer 2: Convolutional. Output = 10x10x21.
Relu Activation.
Pooling. Input = 10x10x21. Output = 5x5x21.

Flatten. Input = 5x5x16. Output = 525.
Layer 3: Fully Connected. Input = 525. Output = 150.
Relu Activation.

Layer 4: Fully Connected. Input = 150. Output = 84.
Relu Activation.
Dropout of 50% connections between Layer 4 and Layer 5

Layer 5: Fully Connected. Input = 84. Output = 43.

Below is an diagram of the CNN model.
using TensorBoard (TensorFlow's built in graph viewer).



3 .Training the model

I started with the same set of hyper parameters as used in the LeNet sample.
Using the AdamOptimizer for the learning rate decay.

I didn't have time to experiment much beyond extending the number of Epochs.

Final hyper parameters chosen

Initial learning rate of 0.001
Drop out of .5
Batch size 128
Number of epochs 20
Initial Weights (mu = 0, sigma = 0.1)

4. Approach

Initially I tried a version of the LeNet model but expanded the first convolution layer to take 32x32x3 as the input. I left all other layers the same until the output layer that needed to be changed to having 43 outputs - one for each image classification type.

With this I achieved approx .90 accuracy.

Next I extended the layers to add more connections. Thinking that classifying into 43 buckets might require the network to have more fine grained features. I increased the depth of the convolution layers and then widened the fully connected layers.

Next I tried playing with coordinates of the actual sign with in the image. First I blanked out the borders with 0s as I didn't think that data was relevant. That actually reduced the accuracy significantly. Then I thought that perhaps if I randomized the borders it would help the model to focus on the more important parts of the image. However this didn't lead to an improvement either, and just slowed the training process. I then tried grayscale image pre processing. As a final step I extended the training dataset by adding a copy of images rotated 15degrees ,and a copy rotated -15 degrees. This help boost the accuracy a few more %points. I also experimented between grayscale and color with rotation. There wasn't much in the difference in terms of final performance ,so I ended up going with the color based model.
In hindsight I think grayscale might have been better , just for the preprocessing performance boost if nothing else.

Training set Accuracy = 0.999
Validation set Accuracy = 0.952
Test set Accuracy = 0.946

My final model results were:

- training set accuracy of ?
    - 99.9%
- validation set accuracy of ?
    - 95.2%
- test set accuracy of ?
    - 94.6%

Iterative Approach

- What was the first architecture that was tried and why was it chosen?
    - Chose the default LeNet architecture modified for 3 channels of color on in the input and 43 outputs in the final layer..

- What were some problems with the initial architecture?
    - Using the same hyper parameters from LeNet achieved approx 90% accuracy on the validation set.
- How was the architecture adjusted and why was it adjusted?
    - First I tried to expand the number of connections (making the mode wider). My intuition was that given we have 43 possible outcomes up from 10 in the original LeNet architecture more connections would allow the model to better differentiate the images.
    - Secondly I added a drop out layer.
    - The combination of these 2 improvements in the architecture boosted the validation performance to approx 95%
- Which parameters were tuned?
    - I trained the model locally on a CPU so time constraints limited the amount of tuning. I researched potentially using Google Cloud Engine's Hyptertuning to help do this, but I didn't have time to experiment , you have to package your tensor flow model in a specific way to do this.
    - I did some basic tuning with the number of Epochs but ended up just using 10.

- How does the final model's accuracy on the training, validation and test set provide evidence that the model is working well?

Comparison of validation results per epoch for 5 different approaches.

Epoch	Basic LeNet with Color input Accuracy	Wider LeNet with Color	Wider LeNet gray scale	Wider LeNet gray scale Rotation +- 15degrees	Wider LeNet color Rotation +- 15degrees
1	0.713	0.789	0.789	0.877	0.88
2	0.792	0.876	0.876	0.914	0.924
3	0.826	0.905	0.905	0.934	0.936
4	0.879	0.922	0.922	0.934	0.943
5	0.881	0.936	0.936	0.938	0.939
6	0.863	0.95	0.95	0.943	0.939
7	0.887	0.941	0.941	0.956	**0.95**
8	0.899	0.944	0.944	0.953	0.944
9	0.884	**0.95**	**0.95**	**0.96**	0.953
10	**0.908**	0.939	0.939	0.958	0.952

Test a Model on New Images

I choose a random sampling of images, which I cropped to be similar to the input images in terms of being mostly the sign, then I ran them through the model.

Discussion per image.

1. 30 KP/H Limit

This test image is clear, well lit , sharp from a front on position and with no rotation. The model should be able to classify this image well. There are some background images with this image which maybe could interfere with the classification.

2. Children crossing

Again this is a very clear image, with a single color background , I would expect a good result.

3. Do Not Enter


On the left was the original image that I ran through the model. It was incorrectly classified, so I tried it again by cropping the image further so more of the image was filled with the sign data (results below) . It was correctly classified only with the second image. So it's important that any higher level processing to identify possible traffic signs in a larger image would correctly bound the image sent to this model.

4. Right Turn

Very clear and uncluttered background. Perhaps the blue background of the image and blue background of the sign might cause some issues with the model , which would need to focus on the arrow as the most important feature.

5 Slippery Road with Snow


I tried 2 versions of this image. I thought it would be interesting to use a more real world example where a human can understand the fact that there is snow on the sign , and yet still make out the symbol underneath. I can imagine this image would be difficult to classify in general. Ideally the training set would have obscured images in order to be able to “filter” out the snow above. In fact for a project like this, it could be possible to add this type of distortion to the training set - either as just pure random noise through the image , or in a structured way to simulate what happened in the above case.
For this image , my final model didn’t classify it at all, nor did it even have the correct classification in the top 5.

6. Slippery Road

I tried a regular slippery road image also to compare performance with the obscured version. This image is very clear and has no rotation so I was more confident this would work successfully.

7. Straight ahead and right

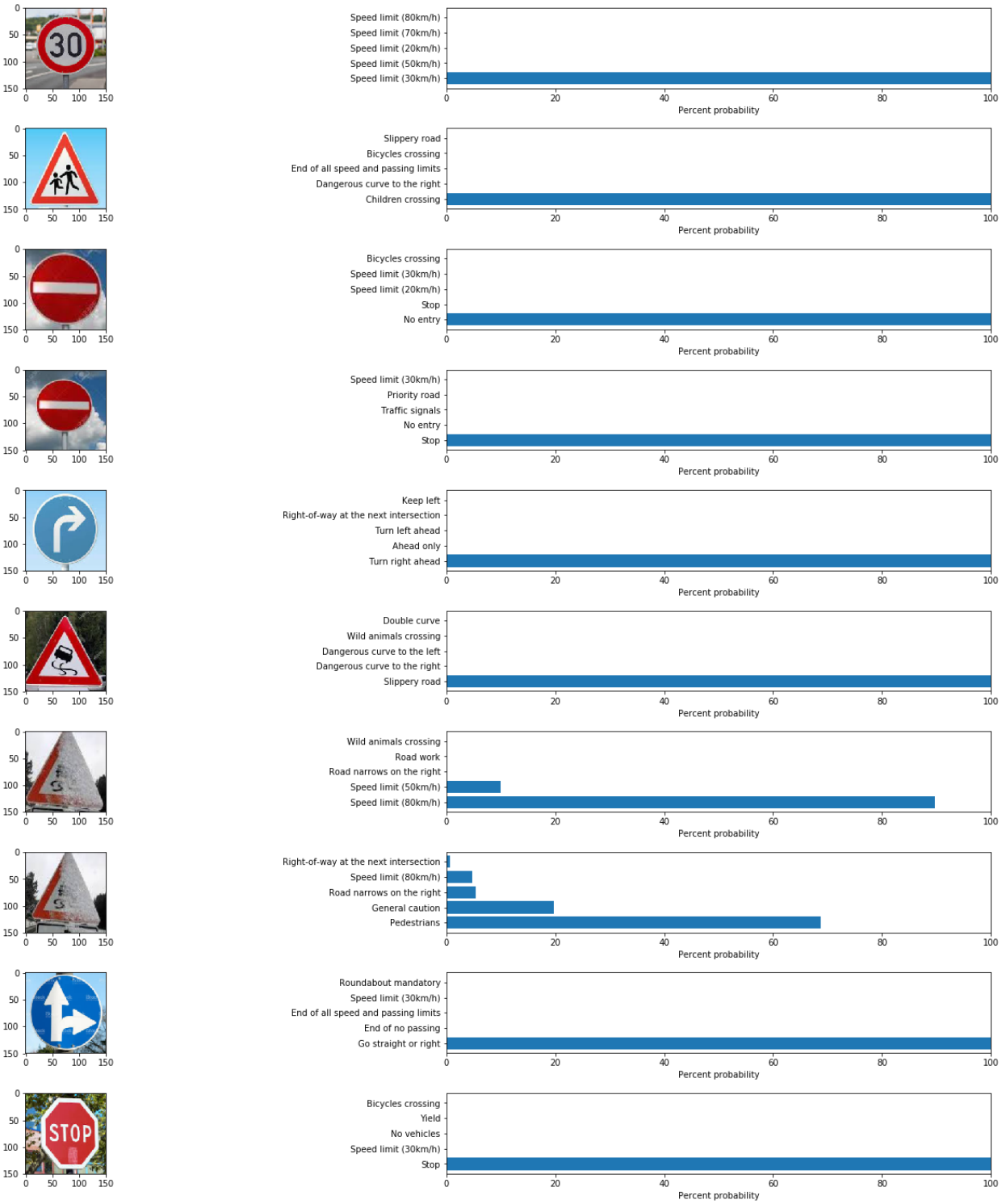
This image should classify well, it’s clear and not obscured.

8. Stop sign

Initially I thought the background on this image might cause it some problems, but it seems that the STOP letters are a pretty strong feature.

Final model performance and softmax probabilities for web images.

The model performed at 87.5% over the 8 images.



Deep dive on model performance for “No Entry”

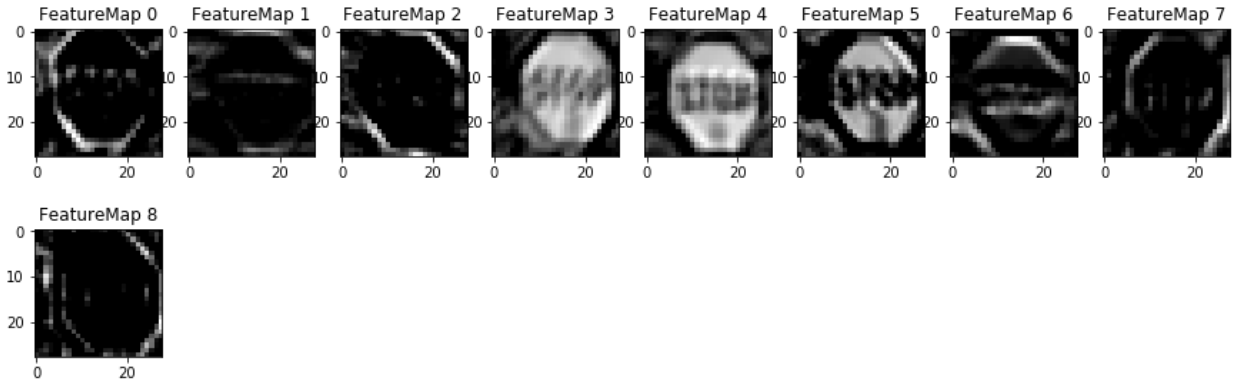
When I initially ran the No Entry image through the model, it classified it as a Stop sign with a very high probability. I wanted to try to debug this and better understand what was happening. So I used the optional part of the project to extract the feature maps at the 2 convolutional layers to see if I could spot anything. I compared these with the stop sign image. In the first layer you can see that the images are somewhat similar , with the STOP and the “------” in a similar place, It’s possible to see why it might be seen as similar. However the 2nd layer seems quite different so I don’t really have a great insight into why it failed. The images are displayed below.

It’s possible that modifying the network further might be able to a better job, but I’m not sure exactly how. This is an area I would be interested in exploring further in the future.

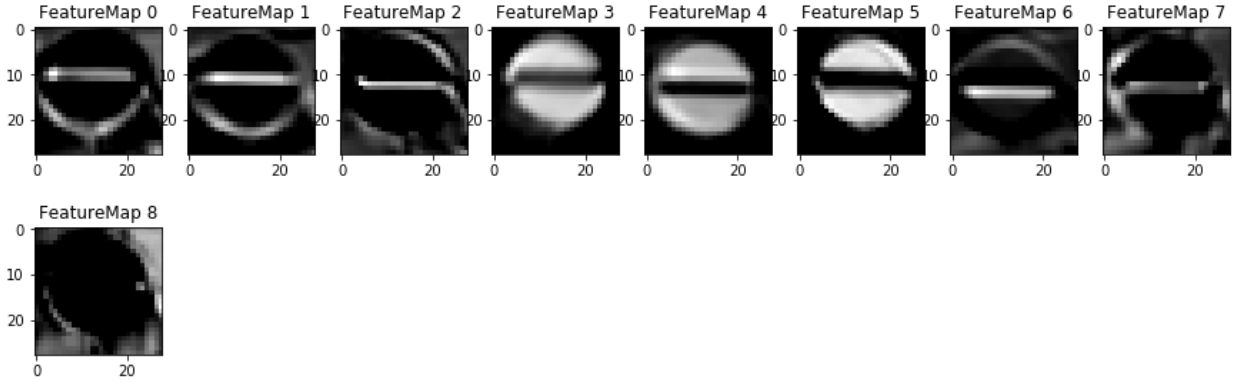
For this particular image, I discovered the issue was the cropping of the original image. By filling more of the input image with the sign, the image was correctly classified. This highlights to me how important it would be to have the sign detection algorithm for say a camera image from a car , be able to consistently crop out the sign before feeding it to the classifier.

First layer

These are for the Stop sign which is correctly classified.

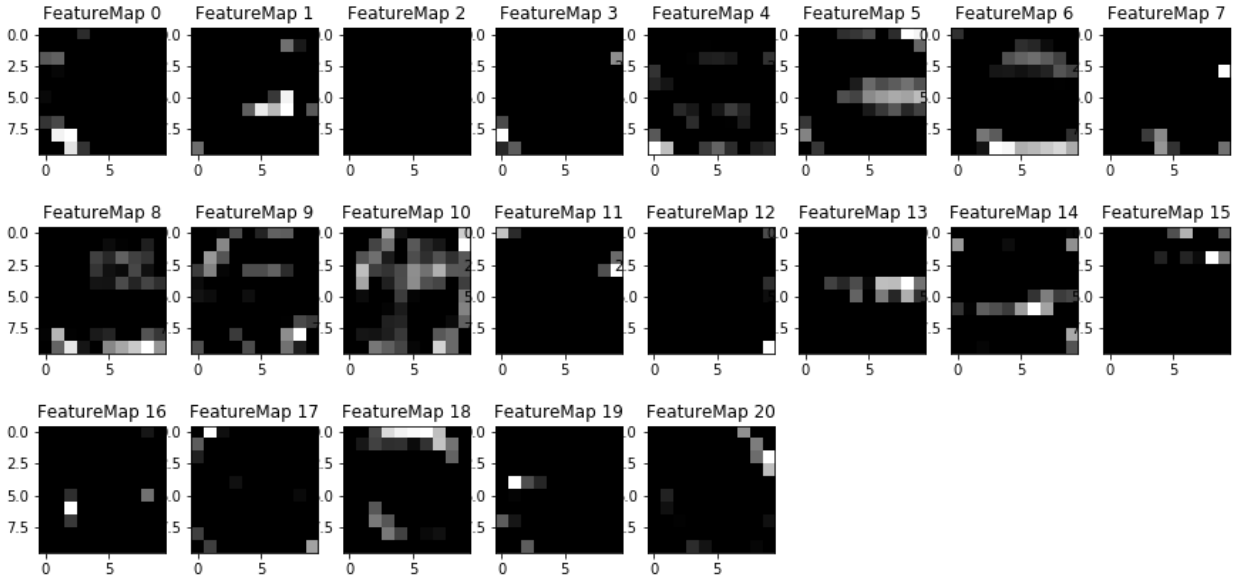


These are for the Do Not Enter sign that is incorrectly classified.



Second layer

Stop Sign



Do not enter sign.