Traffic Sign Recognition Writeup

Build a Traffic Sign Recognition Project

The goals of this project are the following:

- Load the 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

Data Set Summary & Exploration

I was able to initially load my dataset using the pickle module and extrapolate the size of each of the training, validation, and testing data sets. The training data set contained 34799 images. The validation data set contained 4410 images. The testing data set contained 12630 images. Using the extensive Numpy library, I was able to explore the data set. Shown below is the number of elements along with a histogram visualizing the datasets more closely.

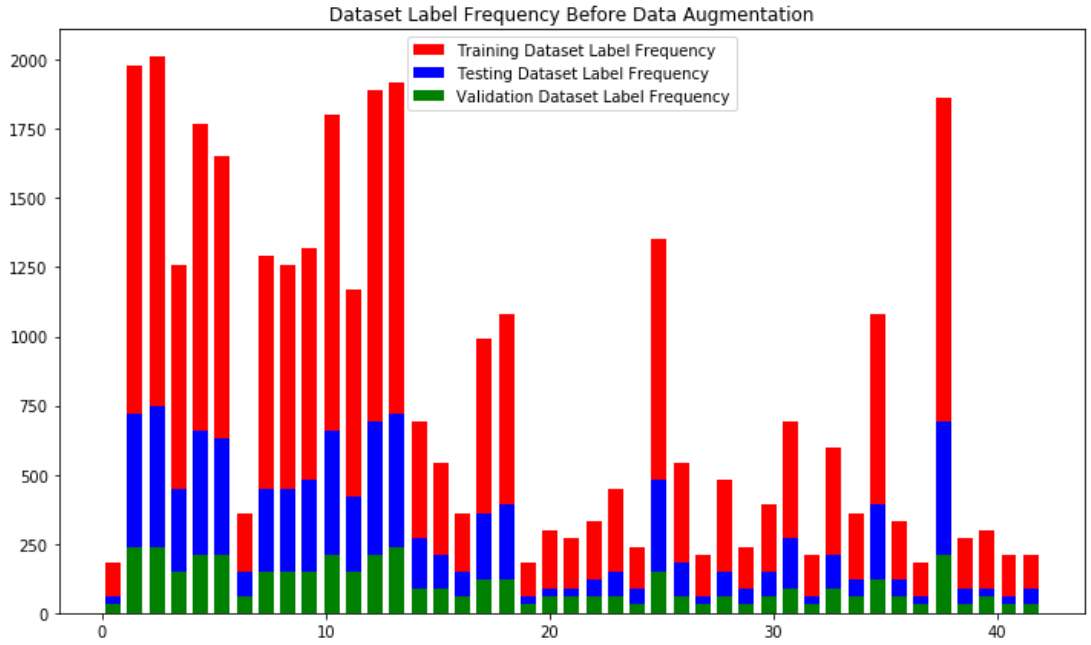

Figure 1. Dataset label frequency before data augmentation with the training dataset displayed in red, the testing dataset displayed in blue, and the validation dataset displayed in green.

```
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
```

As you can see from the histogram above, the data sets are not very well balanced. For example, the second class has over 2000 images while the nineteenth class only has 250 images. Luckily, the training, validation, and testing data sets all follow the same skewed balance. To deal with this challenge, I augmented the data sets, which is analyzed more thoroughly in the next section. Also, to explore the data sets further, I plotted ten images as a visual for what the images will look like; lighting, color, etc. I noticed from this visualization that the images were relatively centered, colored, and varied in lighting as shown below.

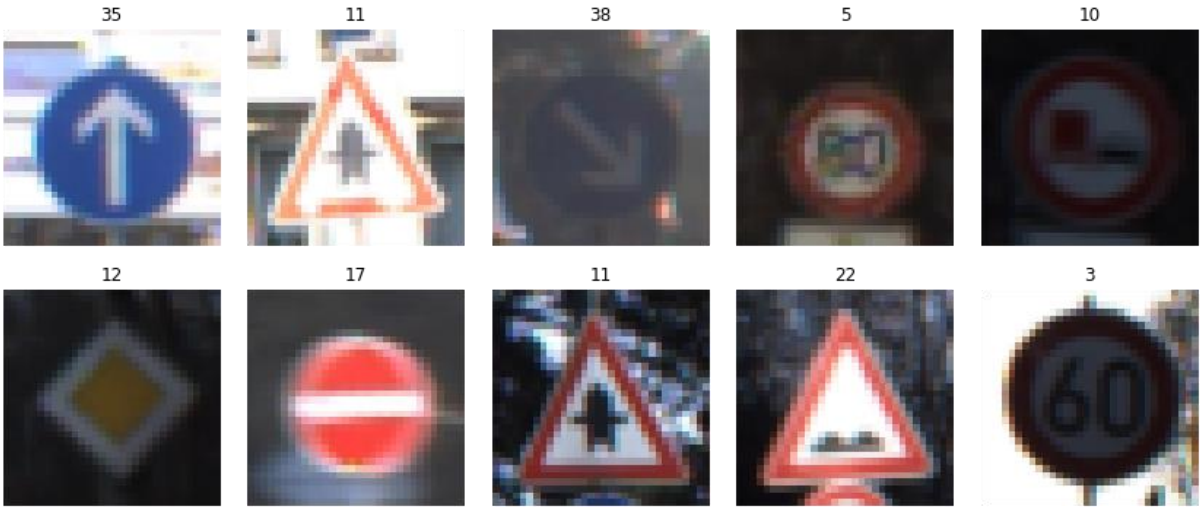

Figure 2. Ten random images to further visualize the data set. Ideally, I would've liked to display the text labels rather than the numbers referring to those text labels, but this works for now.

Pre-Processing & Data Augmentation

I used three main methods for pre-processing the data set and data augmentation. The first process I used was gray-scaling. I performed the task on all three data sets and analyzed the shape change as shown below.

```
X_train_rgb = X_train
X_train_gray = np.sum(X_train/3, axis=3, keepdims=True)
```

```
(34799, 32, 32, 3)
(34799, 32, 32, 1)
```

The main note to make here is the last number that changes from three to one after gray-scaling. This suggests that the color channel was originally three channels (RGB) and is now one channel (Gray).

The second method I used for data augmentation is from the extensive opencv library. I incorporated the warpAffine function and the warpPerspective function to randomly rotate, translate, and scale the images to a "new" image. The methods to produce the skewed images are straightforward enough and implementation was a matter of generating the new images

and appending those new images to an array to later concatenate to the original image data sets. Two main details to point out are:

1. The transformation matrix, M, as seen in the previous project (Advanced Lane Finding) should be made random every time so the new images that are being generated are random.
2. The main reason for data augmentation to generate more data is underfitting during the training of the neural network. After the first run of my model architecture, I was achieving accuracies of less than 5%. Although this ended up being a simple bug in the code, it made me spend a lot of time researching underfitting vs. overfitting and how to adjust data augmentation and the model architecture for both.

Finally, I used the train_test_split method to generate a larger validation data set and get an even more accurate neural network.

Shown below, is the data set label frequency histogram after data augmentation and splitting of the data set.

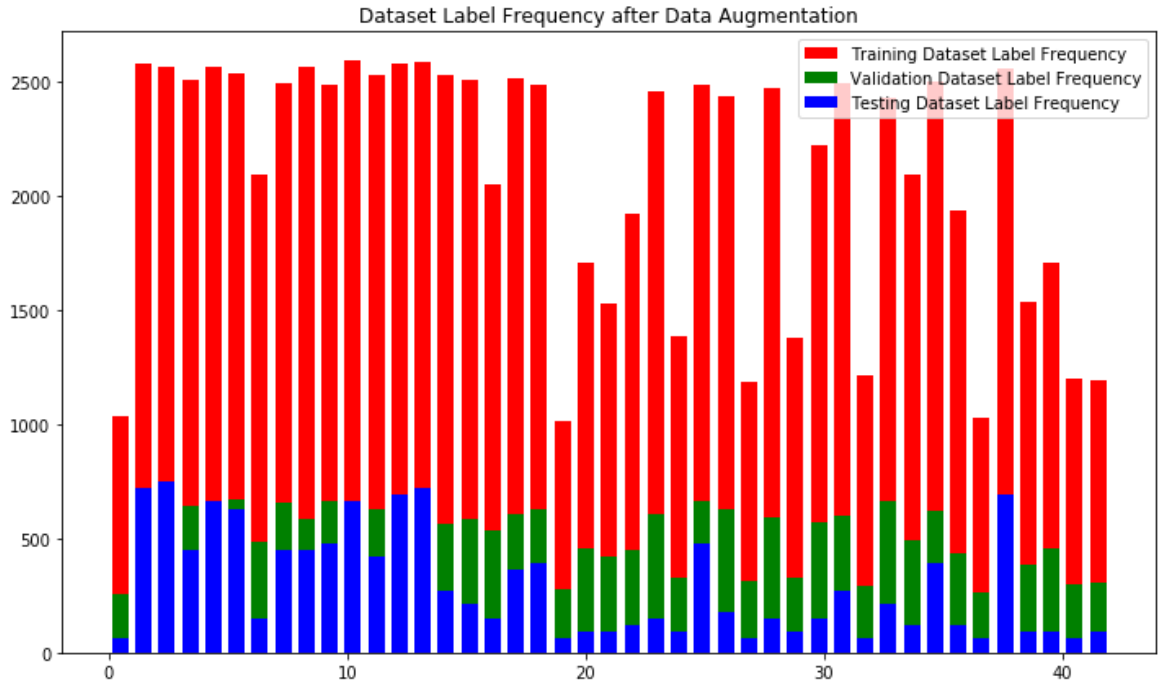

Figure 3. Dataset label frequency after data augmentation with the training dataset displayed in red, the testing dataset displayed in blue, and the validation dataset displayed in green.

Also, below is a visualization of the data augmentation for a few of the images in the original data set.

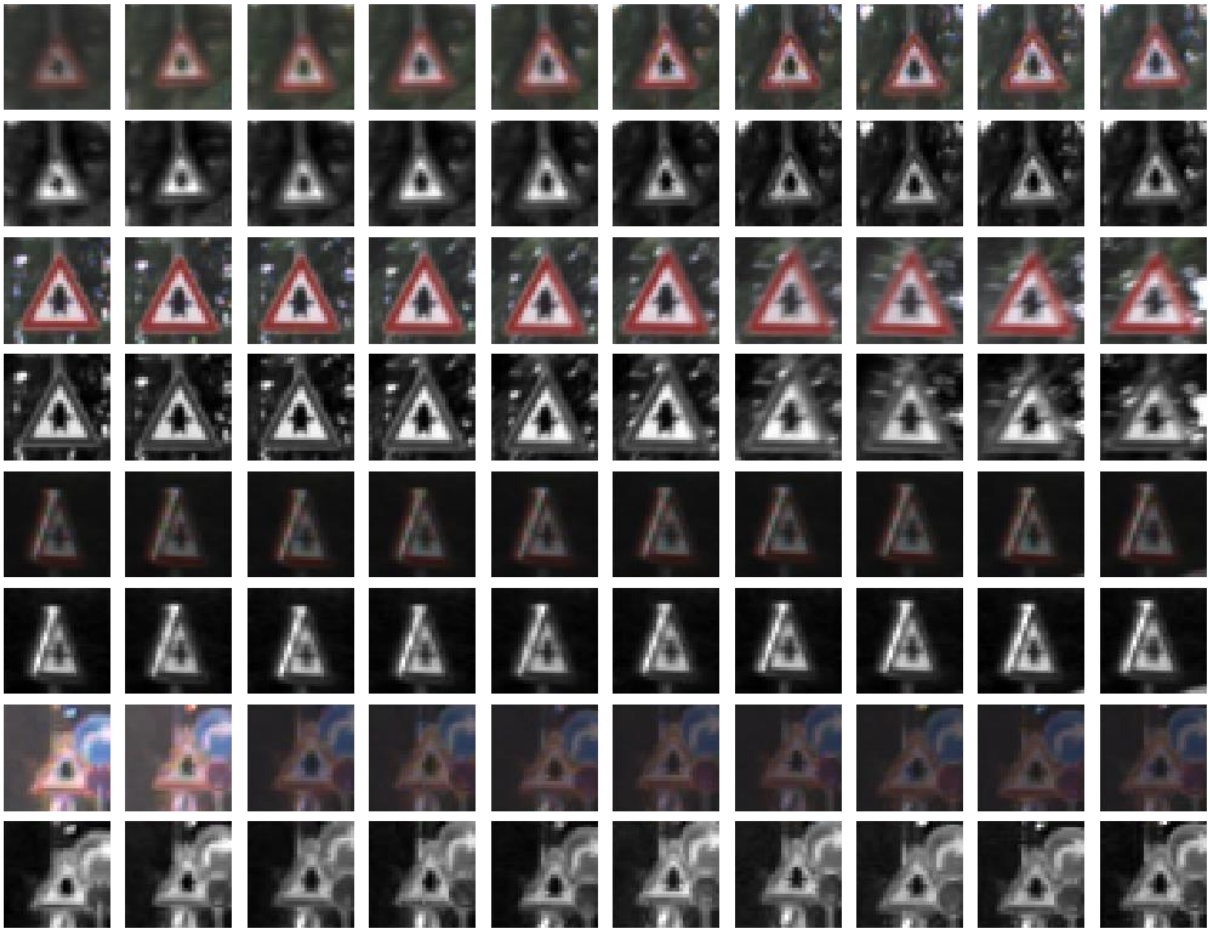

Figure 4. Visualization of the data augmentation for a few of the images in the original data set.

The third method I used for pre-processing, was a simple normalization using the simple equation:

$$(pixel - 128)/128$$

As you can see, the justified mean for all three data sets lies between (-1, 1) after normalization.

Mean before normalization	Mean after normalization
80.2823185241	-0.37279438653
80.0328173977	-0.374743614081
82.1484603612	-0.358215153428

Finally, just before implementing the model architecture, I used the scikit library's shuffle function to shuffle the data set, reduce variance, and reduce the likeliness to overfit.

Model Architecture

I decided to train my model architecture with 30 epochs and a batch size of 32. This proved to return good results and consistently generate a high validation accuracy. The model architecture I used is the Lenet model type discussed in the lessons with three convolution layers, two fully connected layers, and a variation of activation, pooling, and dropout layers in between. The table below further describes the model architecture.

Layer	Description
Input	32x32x1 Gray Image
Convolution 5x5	1x1 stride, padding, outputs 28x28x6
RELU	
Max Pooling	2x2 stride, outputs 14x14x6
Convolution 5x5	1x1 stride, padding, outputs 10x10x16
RELU	
Max Pooling	2x2 stride, outputs 5x5x16
Flattening	Outputs size 400
Fully Connected	Output size 120
Dropout	Reduces overfitting data
RELU	
Fully Connected	Outputs size 84
Dropout	Reduces overfitting data
RELU	
Fully Connected	Outputs size 43

Table 1. Describes each layer of the Lenet architecture used in the project.

Along with the Lenet architecture with 30 epochs and a batch size of 32 described above, I used an AdamOptimizer with a learning rate of 0.001 that is popular with this neural network.

Training, Validating, and Testing

The iterative approach I took to train the model didn't last very long. As mentioned previously, my initial training set and validation set accuracy were below 5%. In order to fix this issue, I dove deep into the statistics of underfitting versus overfitting, produced more data through data augmentation, and adapted my model architecture to have two dropout layers. Again, I ran the code and the training set and validation set accuracy were below 5%. I spent some more time reading through the code looking for any issues I could find. Low and behold, I made an error in the training session basically providing no training data for the model architecture to even be trained on. After fixing the issue, I ran the model architecture twice with the results shown below.

1. The first run of 10 epochs and a batch size of 32 gained a test accuracy of 99.1% and a validation accuracy of 98.8%!
2. Eager to reach the validation accuracy of 99%, my second run of 30 epochs and a batch size of 32 gained a test accuracy of 99.4% and a validation accuracy 99.0%!

A validation accuracy of 99.0% was good enough for me to continue on with testing. By simply running the test dataset in the session that started with my second run of training and validation, I gained a testing accuracy of 94.0%!

A couple of notes to make about my training technique:

- If I wouldn't have been getting accuracies of less than 5%, I probably wouldn't have spent so much time on data augmentation and dropout layers in my model architecture which would've made me have to fine tune the parameters.
- I decided to use the Lenet architecture because it was the model architecture discussed in the course lessons. Therefore, I was most familiar with it.
- Most of the adjustments I made to the model architecture were related to fine-tuning underfitting and overfitting rather than to fine tuning the hyperparameters that help create a more accurate model architecture.

Testing on New Images

Below are the five German traffic signs that I found on the web.

The first two pictures may be hard to classify because the first is highly pixelated and the second contains numbers that could be interpreted as other numbers. However, I felt confident in my model architecture and proceeded. I ran the new images through the pre-processing steps and tested them in my model architecture. Viola! My model architecture was 100% accurate when given the new test images. On the following pages, I display the top guess for each of the new test images, as well as, output the softmax probabilities for each image.

Figure 5. Top three guesses for each of the new traffic signs analyzed. Note: The input images are unfortunately shown in BGR color space rather than RGB from above. An unfortunate visual, but the image is converted to grayscale before being tested in the model architecture.

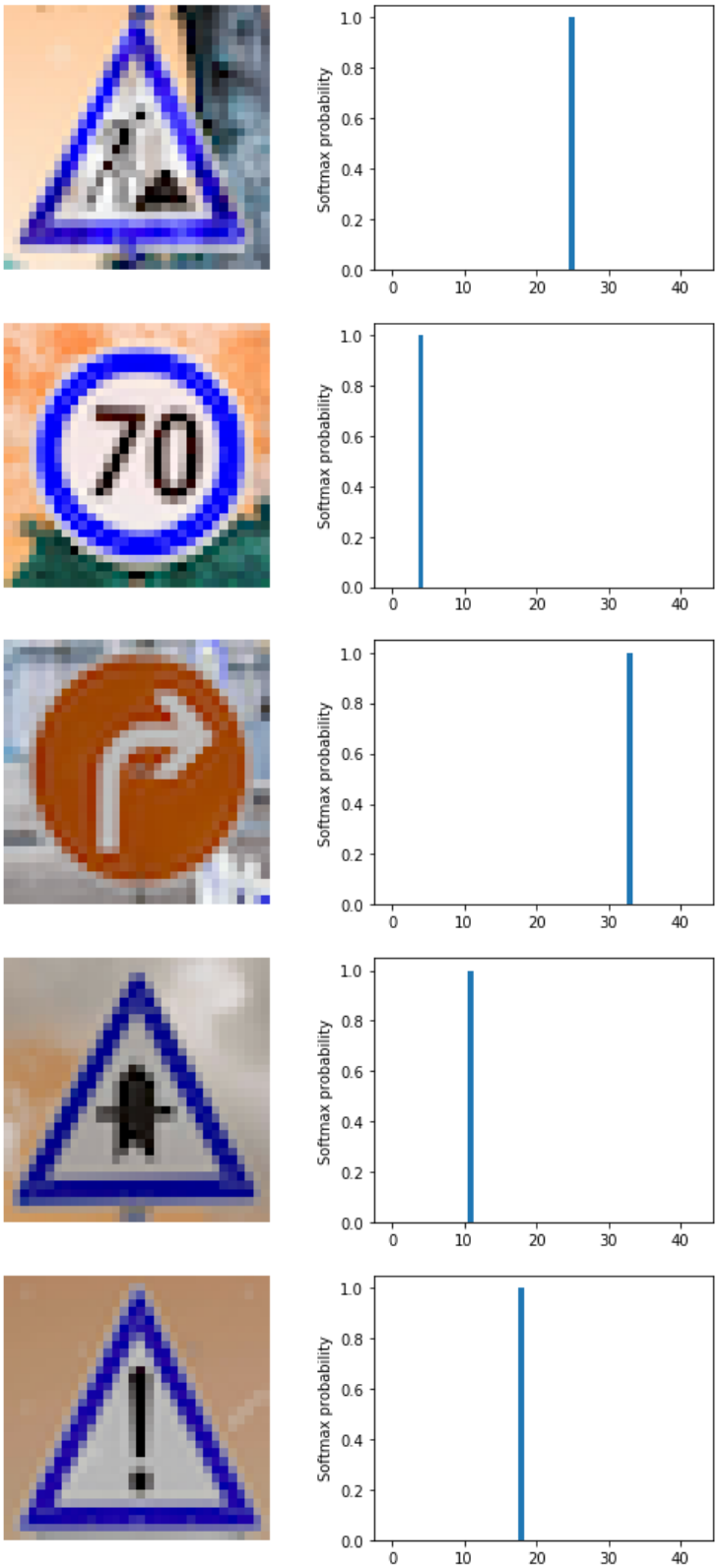

Figure 6. Softmax probabilities for each of the new images. Note: Again, the input images are unfortunately shown in BGR color space rather than RGB from above. An unfortunate visual, but the image is converted to grayscale before being tested in the model architecture.