Traffic Sign Classifier

Overview

The goals/steps 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

Dataset Exploration

Dataset Summary

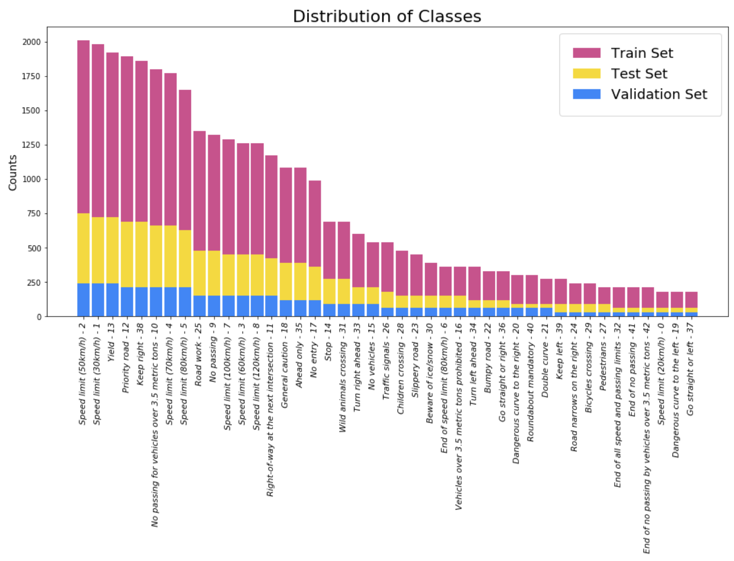
After loading the German Traffic Signs Dataset based on where I saved it on my computer, I extracted basic data summary information by using `numpy` methods. Specifically, I used `numpy.ndarray.shape` and `numpy.unique` to calculate the numbers of examples in the train, validation and test dataset, the dimensions of the images in my dataset, as well as the number of the unique classes.

Exploratory Visualization

In order to extract more information about the dataset and its structure, I did the following:

- An indicative image is shown, chosen randomly from the dataset. The corresponding ClassId of the image is printed as well.

- I created a figure explaining the distribution of the 43 different classes of the data. The figure consists of bars showing the number of images for each class across the training, validation and test sets. By creating this plot, I concluded that the different signs are not represented equally in the data. Some signs have way more samples than others.



The above were created, by using `matplotlib`.

Design and Test a Model Architecture

Preprocessing

Before implementing our model, we need to normalize the inputs, so that they are at a comparable range. My inputs are images, so I proceeded with this suggestion `X_train_norm=(X_train_grey-128)/128` (for all sets of the dataset), making the means of the data approximately zero. Using `numpy`, I confirmed that the means of our inputs are close to zero after normalization.

At first, I proceeded only with this normalization technique. As I was training though my model, I couldn't see improvement. These are some indicative logs, when experimenting with training my model.

```
Epochs: 10, Rate: 0.001 → Accuracy: 0.748
Epochs: 20, Rate: 0.001 → Accuracy: 0.749
Epochs: 20, Rate: 0.01 → Accuracy: 0.650
Epochs: 100, Rate: 0.001 → Accuracy: 0.780
Epochs: 10, Rate: 0.0001 → Accuracy: 0.340
Epochs: 100, Rate: 0.0001 → Accuracy: 0.727
Epochs: 100, Rate: 0.00001 → Accuracy: 0.347
Epochs: 150, Rate: 0.001 → Accuracy: 0.754
```

Then I decided to go back to the preprocessing stage and add one more normalization technique, this of converting my color images to grayscale. Converting the images to grayscale helps with the calculations time, as it reduces significantly the amount of data. I confirmed that my inputs turned grayscale, by plotting a random image from the dataset. My calculations of accuracy improved, as I will show later.

At last, I shuffled the training data, so that I do not obtain batches of highly correlated examples. I did this by importing `shuffle` function from `sklearn.utils` module.

Model Architecture

I implemented the LeNet-5 neural network architecture. The layers of the model are described in the table below.

Layer	**Description**
Input	32x32x1 image
Convolution	28x28 filter, input depth = 1, output depth = 6, 1x1 stride
Activation Function	RELU

Pooling	Output = 14x14x6, 2x2 kernel, 2x2 stride
2nd Convolution	10x10 filter, input depth = 6, output depth = 16, 1x1 stride
Activation Function	RELU
Pooling	Output = 5x5x16, 2x2 kernel, 2x2 stride
Flattening	Length of Vector = 400
Fully Connected Layer	Width = 120
Activation Function	RELU
Fully Connected	Width = 84
Activation Function	RELU
Fully Connected	Width = 43 = Logits, same as our labels of our dataset

Model Training & Solution Approach

For training the model, first I set up the tensorflow variables `x` and `y`, and then mostly concentrated on tuning the `EPOCHS` parameter and the learning `rate` hyperparameter.

Specifically, I set `BATCH_SIZE` variable to be 128. This is the number of how many training images run through the network at a time. As I mentioned earlier, at first I trained the model without converting the images to grayscale. I didn't see any improvement in my accuracy when tuning the training parameters. After converting to grayscale, and tuned again my parameters, I achieved accuracy on the validation set greater than 0.93 (during the lasts epochs, accuracy of . You can see below part of the logs, when trying to tune `EPOCHS` and `rate`:

```
Epochs: 10, Rate: 0.001 → Accuracy 0.897
Epochs: 100, Rate: 0.0005 → Accuracy 0.919
Epochs: 100, Rate: 0.0007 → Accuracy 0.922
Epochs: 100, Rate: 0.0008 → Accuracy 0.930
Epochs: 90, Rate: 0.0008 → Accuracy 0.942
Epochs: 80, Rate: 0.0008 → Accuracy 0.939
Epochs: 90, Rate: 0.0008 → Accuracy 0.941
```

As far as the optimizer is concerned, the `AdamOptimizer` was used, which relies on the Adam algorithm to minimize the loss function.

Test a Model on New Images

Acquiring New Images

After searching the web, I found five German Traffic Signs pictures. Below you can find the corresponding urls of the pictures:

- http://bicyclegermany.com/Images/Laws/100_1607.jpg
- http://image1.masterfile.com/getImage/NjAwLTAzMTUyODU0ZW4uMDAwMDAwMDA=ABwNB4/600-03152854en_Masterfile.jpg
- http://bicyclegermany.com/Images/Laws/Arterial.jpg
- https://thumbs.dreamstime.com/t/roadworks-german-road-sign-ahead-81618639.jpg
- http://bicyclegermany.com/Images/Laws/Do-Not-Enter.jpg

I converted the urls to images, then resized the images (using `opencv` library) to match the size of my model's input. Afterwards I grayscaled the images, and finally output the images with their corresponding ClassId using `matplotlib`.

The signs in the pictures are well lighted. Some of them though show the signs skewed, something that could make the classification more difficult.

Performance on New Images

After normalizing and shuffling the data, I proceeded with training my model on the images I found on the web. The performance on the new images is compared to the accuracy results of the test set and the accuracy is found 0.6. So, my model predicted correctly 3 out of 5 images.

Using `tf.nn.top_k` I found the three most possible guesses, and plotted the corresponding images and labels. This is a nice way visualizing the predictions of the model. I found this visualization code searching online.

Model Certainty - Softmax Probabilities

In order to find out the certainty of our model's predictions, I printed its five largest softmax probabilities for each image using `tf.nn.top_k`.