Traffic Sign Recognition

Sanyam Agarwal - April 2018 BATCH

Goal and steps of project

The goal is to create a traffic sign recognition program using convolutional neural network.

The 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

Rubric points:

— Basic Summary of the Data Set

I used the ‘numpy' library to calculate summary statistics of the traffic signs data set:

Number of training examples = 34799

Number of validation examples = 4410

Number of testing examples = 12630

Image data shape = (32, 32, 3)

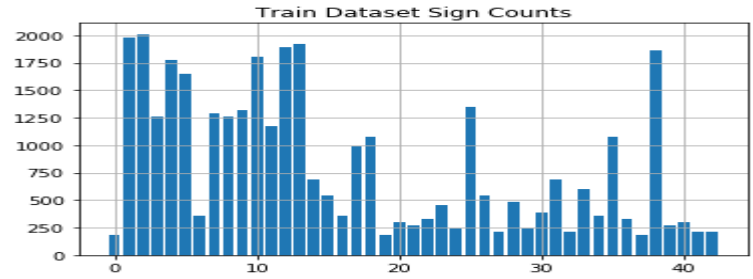
Number of classes = 43

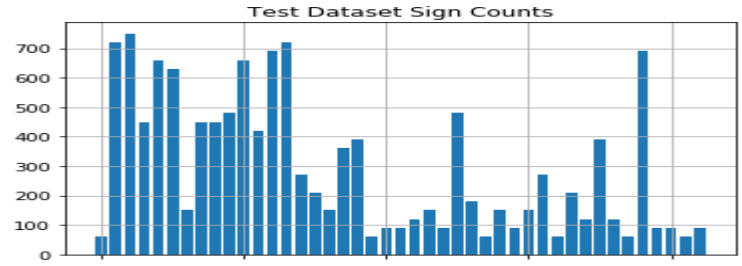
— Include an exploratory visualization of the dataset

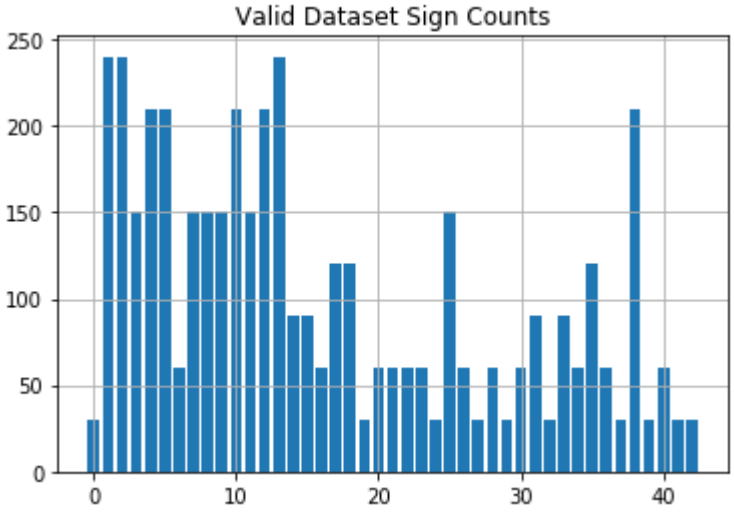
The code for this step is contained in the third code cell of the IPython notebook.

This visualization provides a good quick overview about, quality and visuals of the image and also it becomes clear that, some class ids has more samples than others.

Below is an exploratory visualization of the data set. This visualization will tell how many samples are available per class ID.







— Design and Test a Model Architecture

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

Since the images could be of various intensity and from different angles, it was required to find out a way to make images more understandable for the model.

Below are the steps taken on datasets\images for pre-processing:

1. **Data Normalization:** Normalizing the data to the range (-1,1).

   This was done using the equation:

   X_train_normalized = (X_train - mean_of_X_train)/standard_deviation_of_X_train,

Below shows mean and variance of my Train, Valid and Test data before and after normalization. After normalization they mean very close to zero and variance are equal to each other.

```
== Data before Normalization ==
Train Data  82.6775890369964
Valid Data  83.55642737563775
Test Data  82.14846036120183
Train Variance [0-127]  4913.158880331108
Train Variance [128-255]  4627.149022786359
Train Variance [256-384]  4667.095589377374
```

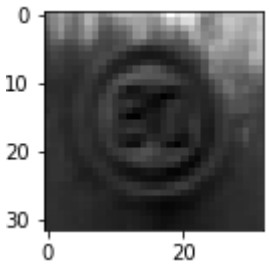
```
== Data after Normalization ==
Train Data  -1.7228093718140882e-18
Valid Data  -2.3496783589950405e-19
Test Data  -4.020126655544246e-18
Train Variance [0-127]  1.073472545797675
Train Variance [128-255]  1.0055238233124735
Train Variance [256-384]  1.0187481772143239
```

As seen above data are very close to zero mean and equal variance which is a good starting point for optimizing the loss to avoid too big or too small.

2. **Converting image into Grayscale**: I use gray scale image in LeNet Architecture using Tensorflow. Training time is less and this will be helpful while training the network on my linux machine.

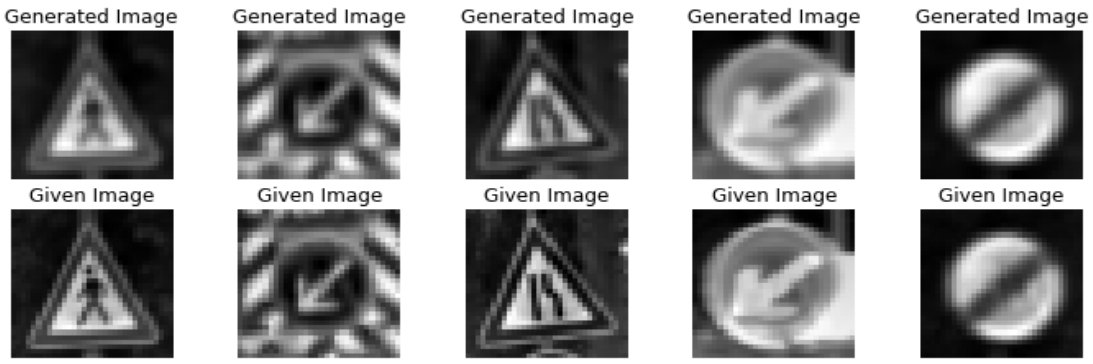
```
Gray Image, Class_id 5

Gray X_train shape  (34799, 32, 32, 1)
```



3. Data Augmentation : Some of the classes were having more data points compare to other classes sometimes 10 fold higher. This is a problem because the lack of balance in the class data will lead into becoming biased toward the classes with more data points. Instead of omitting valid data from the classes with more datapoints I decided to generate more data for the classes those are less than 500 by using a Gaussian Blurring of the real image with a kernel size of 3, 5 etc..

Below are the results of augmented image compare to original image



As you can see the augmented images are blurred compare to original images, this solves the two purpose:

- Less chance of overfitting as classes will have balanced data
- Blurring reduces noise

— Model Architecture

The LeNet architecture is used as the base for this project.

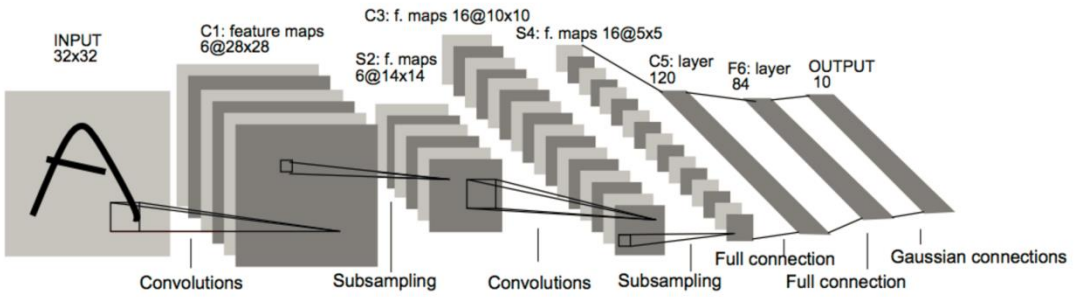


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Below is the final Model of layers used:

Layer	Funtionality	Input	Output
Layer 1	Convolutional(5X5)- relu activation, 1X1 Stride, VALID Padding	32X32X1	28X28X6
	MaxPool- Fiter 2X2	28X28X6	14X14X6
Layer 2	Convolutional(5X5)- relu activation, 1X1 Stride, VALID Padding	14X14X6	10X10X16
	MaxPool- Fiter 2X2	10X10X16	5X5X16
	Flatten	5X5X16	400
Layer 3	Fully connected - relu activation- Dropout	400	120
Layer 4	Fully connected - relu activation- Dropout	120	84
Layer 5	Fully connected	84	43

— Train, Validate and Test the Model

To train the model I used following Hyper parameter \ Configuration:

EPOCHS = 15

BATCH-SIZE = 128

Rate = 0.001

Prob1 = 0.5

Mu = 0

Sigma = 0.1

Activation = relu

MAXPOOL = 2x2

Optimizer = AdamOptimizer

During training the model, the values of training set are shuffled before creating a new batch to provide more robust training. The loss is calculated at the end of each training iteration optimized using “AdamOptimizer” which is most popular for adaptive optimization. The trained model fed with validation samples to calculate the validation accuracy as well.

Final model results were:

```
Training_accuracy   = 0.990
Validation Accuracy = 0.953
Test Accuracy = 0.922
```

Initially this architecture with default learning rate and other hyper parameter gives only approximate 89% of validation accuracy, then there were many iterations and techniques were tried to make the stable parameters to achieve 93%+ accuracy for trained network.

Some techniques / adjustments which were tried and adjusted:

- Increasing the number of EPOCHS which started causing the overfitting at some point, uses Dropout to remove overfitting.
- Multiple dropout values were tried 0.7, 0.2 and then finally set with 0.5. With 0.5 it gives the better result and prevent overfitting
- Tried adding more convolution layers after 1st and 2nd convolution, which did nto help much in increasing the validation accuracy and then were removed.
- Uses mean = 0 and sigma =0.1 , low sigma means small peak. Less opinionistic, can train better

— Test a Model on New Images

To test the trained network well, it was required to test the model with random internet images of the German traffic signs.

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

These images are not in 32x32x3 size. I Resize them to 32x32x3 in order to feed to LeNet model. The model was able to correctly guess all 5 traffic signs, which gives an accuracy of 100%.

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

Here is the Probability and Prediction of the model

Probability	Prediction
1	Yield
1	Slippery Road
1	Keep Left
1	60 km\hr
1	100 km\hr

Network was able to predict the new signs with 100% accuracy.

For understanding this, I print the softmax probabilities for top 3 predictions and also the top 5 probable class ids:

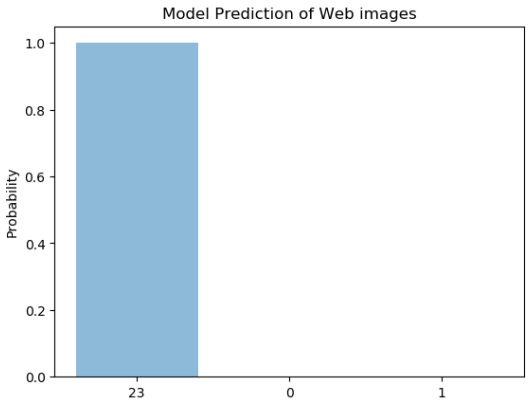
Top 5 Probable predicted classes :

```
13 -->  [13,  0,  1],
23 -->  [23,  0,  1],
35 -->  [35,  0,  1],
3  -->  [ 3,  0,  1],
7  -->  [ 7,  0,  1]
```

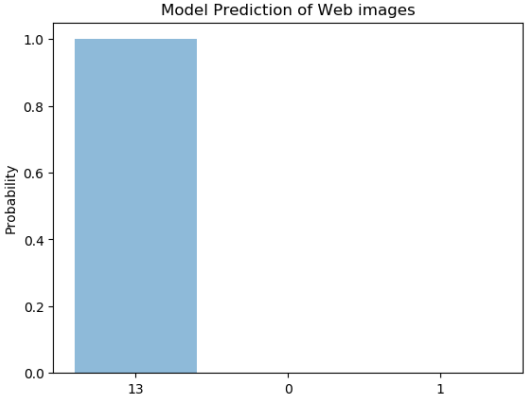
We will now plot the probability of each class to get predicted.

By printing the softmax probabilities for these predictions we get following top 5 probabilities, and possible predictions from the trained network.

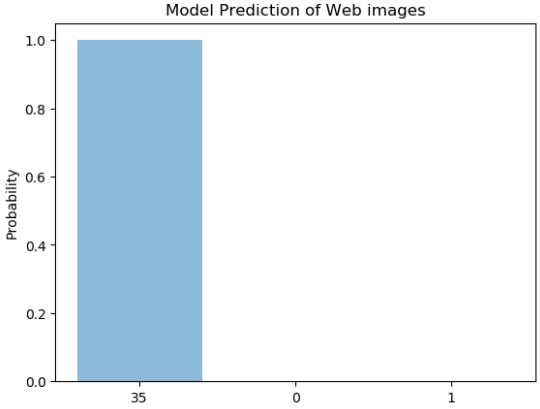
Expected - 23



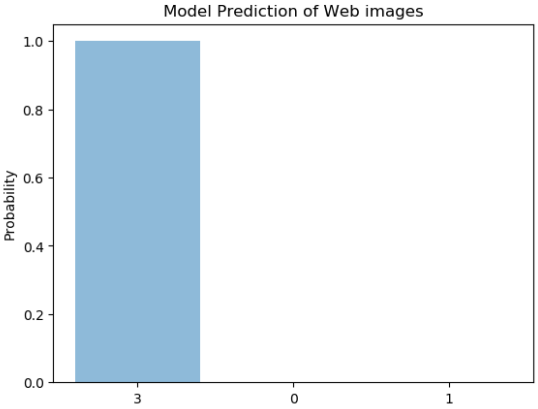
Expected - 13



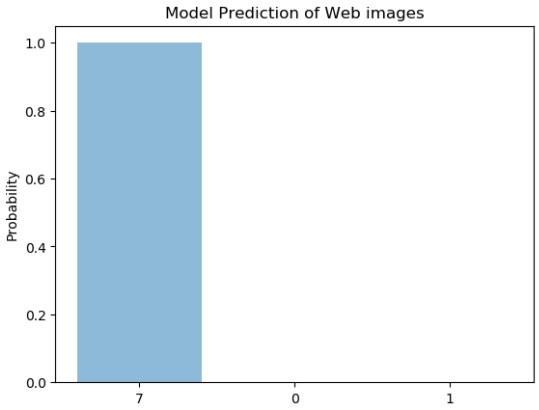
Expected -35



Expected - 3



Expected - 7



Analysis:

By carefully checking the probability plots, its evident that the trained network is quiet sure about the predictions.

The model works well with new web images\data which is good but in real world with not so clear images might need more careful training with better performance.