Traffic Sign Classification

- **Files Submitted:**
  - constants.py – Contains all constants
  - preprocessing.py – contains Normalization and One Hot Encoding Code
  - fake_data.py – contains Augmentation code
  - load_labels.py – contains code for load all the classes of the dataset
  - NeuralNetwork.py – contains code for implementing the network
  - train_aug.p – contains augmented dataset
  - model-final.index, model-final.data, model-final.data – The trained model saved files
  - test_images – directory containing test images
  - american_images – directory containing other country images

- **Data Set Summary & Exploration:**

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

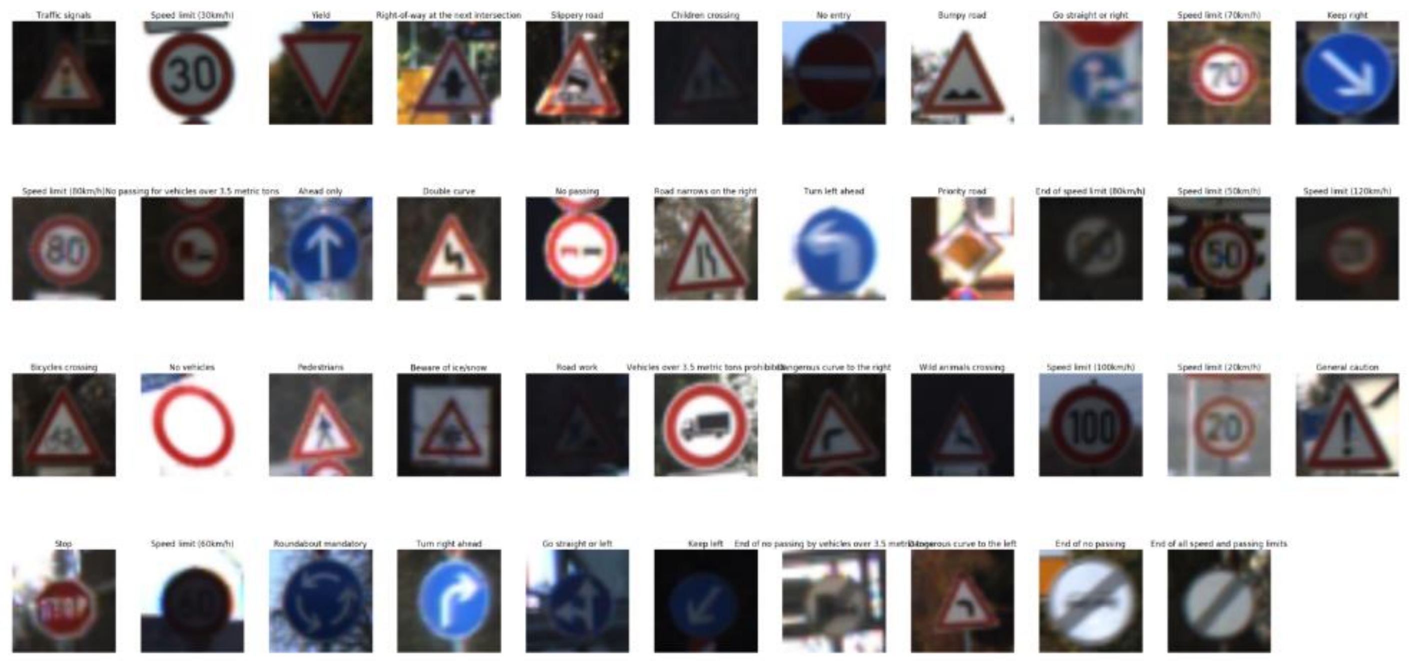
- The size of training set is 34799
- The size of validation set is 4410
- The size of test set is 12630
- The shape of a traffic sign image is (32, 32, 3)
- The number of unique classes/labels in the data set is 43

- **Exploratory Visualization of Dataset:**

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

Here is an exploratory visualization of the data set. I have randomly selected an image from the dataset, repeated 1000 times, just to be sure that I would cover all classes of images. Then I got the labels for different classes and mapped each of the randomly picked image uniquely to the labels. Then I attached the corresponding labels as titles to the images and displayed them.

```
Different Classes of Images:
```



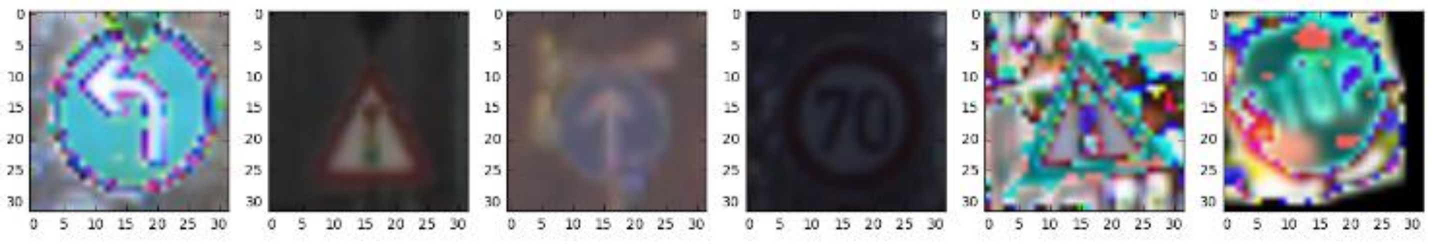
- **<u>Design and Test a Model Architecture:</u>**
-

1. **<u>Preprocessing</u>**

   For preprocessing the dataset, I have first Augmented the data set by adding additional images by taking the original images and rotating and translating them. I have augmented the data this way because, I felt the more data the network had the better it would learn. Also, adding rotations and translation would help the network recognize images better in various conditions rather than only in clean images.

   Then I have normalized the data to optimize the classifier. Normalizing the data would help the classifier to quickly learn by reducing the search time during gradient descent.

   Here are some examples of the images after the preprocessing stage.



The code for preprocessing is in code cells 5,6 and 7. It can be found in files **fake_data.py** and **preprocess.py**

2. Architecture

Layer	Output Dimensions
Input	32x32x3
Convolution, Filter Width = 3, Stride = 1, Padding = SAME, Depth = 16	32x32x16
Max Pool, Filter Width = 3, Stride = 1, Padding = SAME	32x32x16
Convolution, Filter Width = 5, Stride = 3, Padding = VALID, Depth = 64	10x10x64
Max Pool, Filter Width = 3, Stride = 1, Padding = VALID	8x8x64
Convolution, Filter Width = 3, Stride = 1, Padding = SAME, Depth = 128	8x8x128
Convolution, Filter Width = 3, Stride = 1, Padding = SAME, Depth = 128	8x8x64
Max Pool, Filter Width = 3, Stride = 1, Padding = VALID	6x6x64
Flatten	2304
Fully Connected Layer, 1024 Hidden Nodes	1024
Dropout Layer	1024
Fully Connected Layer, 1024 Hidden Nodes	1024
Dropout Layer	1024
Output Layer	43

3. Model Training

The code for Networks class is in the file **NeuralNetwork.py**, and the NeuralNetwork object is created and layers are added code cell 10

I have used Gradient Descent Optimizer for training my model.

Some of the parameters I have used after a lot of trial and error:

EPOCHS = 200

BATCH SIZE DURING TRAINING = 128

BATCH SIZE DURING EVALUATION = 2048

LEARNING RATE = 0.005

All these parameters are set in constants file which is named as **constants.py**

4. Solution Approach

I have tried various approaches for training my model. I have tried LeNet, AlexNet and GoogleNet Inception architectures. LeNet architecture was doing good but the results were better using the AlexNet architecture. I have also tried to implement GoogleNet's Inception Layer with parallel layers but I was not able to run it due to insufficient memory in the GPU.

The code for the accuracy results are in the second and fourth code cells under the heading "Train, Validate and Test the Model"

My final model results were:

- Training set accuracy of 0.997
- Validation set accuracy of 0.957
- Test set accuracy of 0.941

My model has been inspired from the AlexNet Architecture and is pretty like it. I have tried to modify the architecture by adding parallel inception layers, but was not able to run it due to memory problems. However, I have presented the code for inception layer in **NeuralNetwork.py** and will try it in the future.

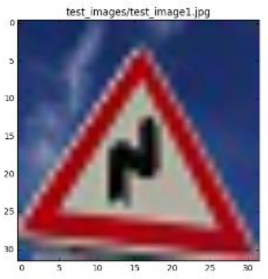
I have added two Dropout Layers to regularize the model and avoid overfitting. However, there is still slight overfitting in my model.

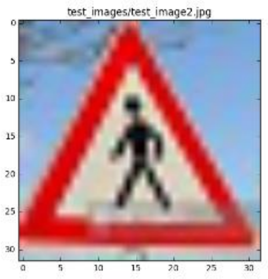
The final score on Training, Validation and Testing sets prove the model is very good. It works great on new images as well.

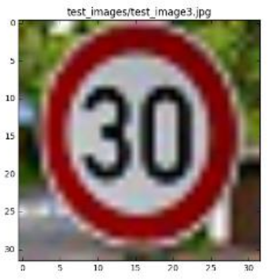
- <u>**Test Model On New Images:**</u>

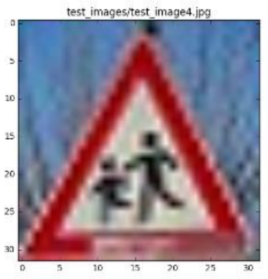
1. Acquiring New Images

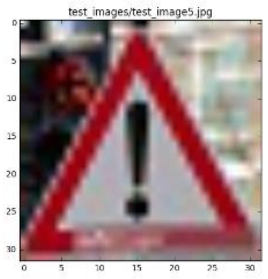
Here are the five German Traffic Signs I have chosen:











2. Performance on New Images

The code for making predictions in the first code cell under the heading "Predict the Sign Type for Each Image"

Here are the results of the prediction:

Sign	**Prediction**
Double Curve	Right-of-way at the next intersection

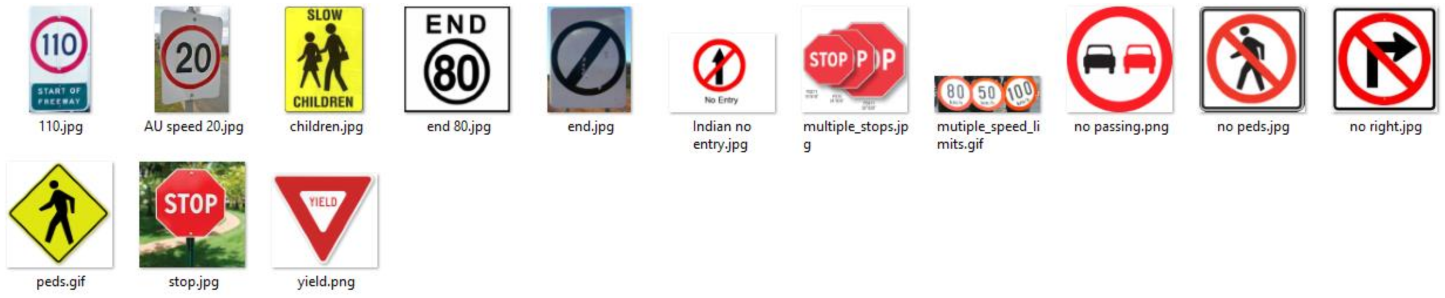
Pedestrians	Pedestrians
Speed limit (30km/h)	Speed limit (30km/h)
Children crossing	Children crossing
General Caution	General Caution

The Network Performed Quite Well.

The model predicted 4 out 5 signs correctly, giving an accuracy of 80%.

I have also run the model on Non-German Traffic Sign Images including American, Indian, Australian Traffic Signs.

Here are the images and results:



Sign	**Prediction**
Speed Limit 110	Speed limit (30km/h)
Speed Limit 20	Speed limit (20km/h)
American Children	Keep right
End 80	Speed limit (50km/h)
End	End of all speed and passing limits
No Entry	Vehicles over 3.5 metric tons prohibited
Multiple Stop Signs	Stop
Multiple Speed Limits – 80, 50, 100	Speed limit (50km/h)
No Passing	No Passing
No Pedestrians	Speed limit (20km/h)
No Right	Keep Right
Pedestrians	End of all speed and passing limits
Stop	Stop
Yield	Yield

The Network seems to have performed well for other images as well, especially considering the fact that, it was not trained on these images.

3. Model Certainty – Softmax Probabilities

The code for making predictions on my final model is located in the first code cell after the heading “Output Top 5 Softmax Probabilities For Each Image Found on the Web”.

For the first image (Double Curve) these are the Top 5 Probabilities:

- Right-of-way at the next intersection: 99.97%
- Pedestrians: 0.03%
- Beware of ice/snow: 0.00%
- Wild animals crossing: 0.00%
- Dangerous curve to the right: 0.00%

For the second image (Pedestrains) these are the Top 5 Probabilities:

- Pedestrians: 100.00%

- Road narrows on the right: 0.00%
- General caution: 0.00%
- Right-of-way at the next intersection: 0.00%
- Double curve: 0.00%

<u>For the third image (Speed Limit 30) these are the Top 5 Probabilities:</u>

- Speed limit (30km/h): 100.00%
- Speed limit (50km/h): 0.00%
- Speed limit (20km/h): 0.00%
- Stop: 0.00%
- Road narrows on the right: 0.00%

<u>For the fourth image (Children Crossing) these are the Top 5 Probabilities:</u>

- Children crossing: 99.86%
- Bicycles crossing: 0.13%
- Wild animals crossing: 0.01%
- Beware of ice/snow: 0.01%
- Road narrows on the right: 0.00%

<u>For the fifth image (General Caution) these are the Top 5 Probabilities:</u>

- General caution: 99.98%
- Pedestrians: 0.01%
- Dangerous curve to the left: 0.00%
- Double curve: 0.00%
- Traffic signals: 0.00%