Machine Learning Engineer Nanodegree

Capstone Project

Traffic Signs Classifier Using Convolutional Neural Network (CNN)

Osama Mosaad

2 December 2017

I. Definition

Project Overview

The problem domain this project is trying to tackle is to recognize and classify the traffic signs in the real-world to help self-driving cars (as a typical use case) to do this task efficiently while driving on the roads. The dataset used in that project is **German Traffic Signs Dataset** (1). This dataset has more than 50,000 images in total classified into 43 classes. Due to the ambitious projects to build self-driving cars, classification of traffic signs became very essential task to achieve this.

Generally, the humans are capable to recognize the traffic signs with very high level of accuracy (humans can do this task with accuracy of **98.81%**) and this could be considered as a real challenge to the implemented solution which also has to achieve a high level of accuracy and correctness near to the humans' benchmark.

Problem Statement

Classification of the Traffic signs is very essential problem that has to be tackled very well in any self-driving car solution. As we expect human drivers to be highly capable to recognize the traffic signs, we expect also that the auto-drivers (self-driving cars) to be the same to achieve the safety aspects while driving on the roads. Traffic signs recognition is a multi-class classification problem with unbalanced class frequencies. There is a wide range of variations among the different classes of the traffic signs in terms of shapes, colors, having text inside the traffic signal or not. On contrary, parts of these classes are very similar to each other such as the speed limit signs.

In reality, there are large variations in the visual appearance of the traffic signs due to the weather conditions, illumination changes, rotations, the time of the day (night time or day time)…etc. All of these factors should be considered when we design a solution for this challenging real-world problem.

The implemented solution here to solve this problem is mainly building a convolutional neural network classifier (CNN classifier) of multi-convolution layers and max-pooling layers, then compile and train that model using the inputs from the training dataset. This CNN classifier has been trained from scratch to able to detect the patterns in the images and to learn the low and high level features of the images given in the training dataset. Finally this trained classifier has been evaluated against the test dataset to assess its accuracy and ensure it generalizes well for the unseen instances in the real-world.

Metrics

The evaluation metrics that I have used to evaluate both the solution model and the benchmark model is **the test accuracy**. Model's accuracy is the number of instances which are correctly classified divided by the total number of test instances in our test dataset. The accuracy is represented by the following formula:

$$\text{Model's Accuracy} = \frac{\text{No. of correctly classified instances}}{\text{Total number of test instances}}$$

Also I have used precision and recall metrics; here are the formulas of Precision and Recall subsequently:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Basically, this is a classification problem that is why I have chosen the above evaluation metrics which are suitable for that type of problems.

Using the final results of the solution model and the predicted class labels of the test dataset samples, I also plotted the confusion matrix to show the predicted labels versus the true or actual class labels.

II. Analysis

Data Exploration

The dataset that I used for this project is the **German Traffic Signs Dataset** [(1)]. This dataset has more than 50,000 images in total classified into 43 classes. The training data set contains 39,209 training images in 43 classes and the test dataset contains 12,630 test images. The original image format is "PPM (Portable Pixmap, P6)" which is basically

RGB format. The images are not necessary squared as image sizes vary between 15x15 to 250x250 pixels. The images in the training dataset are grouped in 43 directories (folders); the name of each folder is the encoding of the corresponding class label in a format of numbers from 0 to 42.

The images contain one traffic sign each and Physical traffic sign instances are unique within the dataset which means each real-world traffic sign only occurs once. The actual traffic sign is not necessarily centered within the image and there is about 10% border around the actual traffic sign. This dataset comes with Annotations provided in CSV files. Fields in these CSV files are separated by ";" (semicolon). Annotations contain the following information:

- Class Id: Assigned class label (For the training dataset)
- Filename: Filename of corresponding image
- Width: Width of the image
- Height: Height of the image
- ROI.x1: X-coordinate of top-left corner of traffic sign bounding box
- ROI.y1: Y-coordinate of top-left corner of traffic sign bounding box
- ROI.x2: X-coordinate of bottom-right corner of traffic sign bounding box
- ROI.y2: Y-coordinate of bottom-right corner of traffic sign bounding box

As initial preprocessing of the original dataset, we iterated over the images of the training and test datasets to resize each image to a fixed width and height (32 x 32) pixels, and then convert each resized RGB image to a numpy 2-D array with numbers range from 0 to 255 which represents the intensity of each pixel. After that initial preprocessing, I dumped this data into two pickle files; one for the training data "Train.pkl" and the other one for test data "Test.pkl", to save the initial transformations that we have done and make the further processing easier.

<u>Note:</u> I have uploaded the two pickle files on OneDrive and they can be reached and downloaded through this link https://1drv.ms/f/s!Apt9CJrW-9NSghXkbEbbY50ZAe-H

The training dataset has been split into two splits; one split for the actual training dataset which represents 88% of the original training data and the other split for the validation dataset which represents the other 12%.

I already did analysis over the data and here a list that summarizes the statistics about the samples count per each class label in the actual training dataset after splitting original data into training and validation datasets.

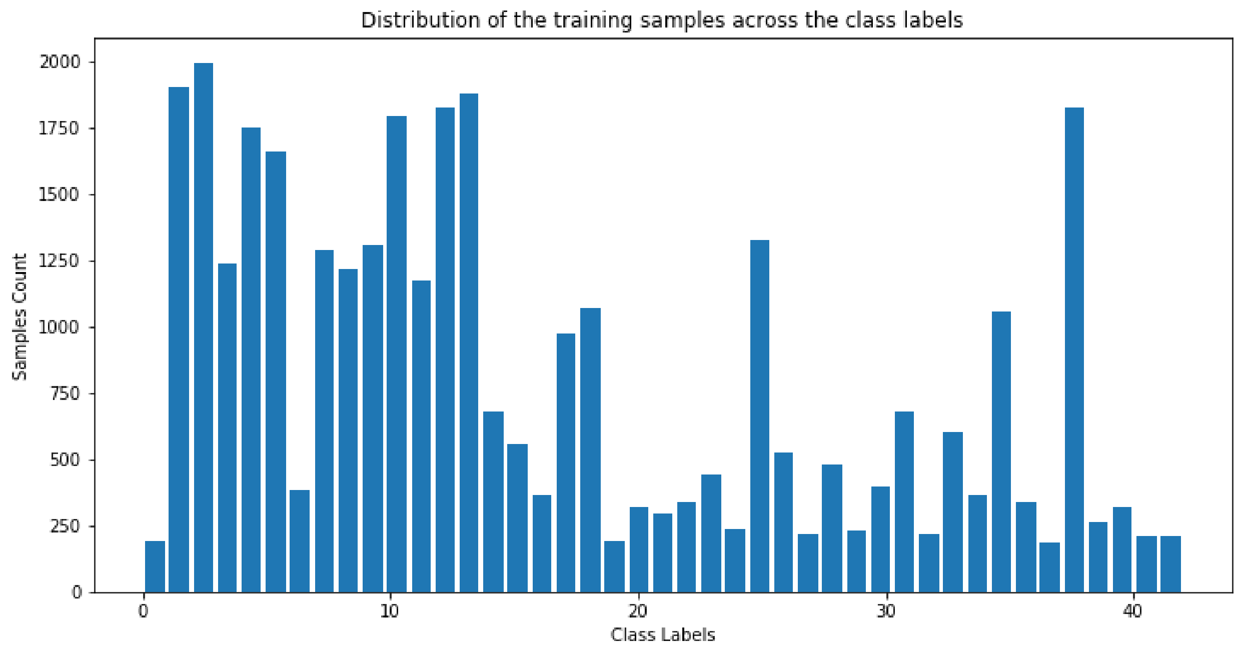
ClassId	**class Label**	**Samples Count**
0	Speed limit (20km/h)	188
1	Speed limit (30km/h)	1905
2	Speed limit (50km/h)	1991
3	Speed limit (60km/h)	1235
4	Speed limit (70km/h)	1750

5	Speed limit (80km/h)	1659
6	End of speed limit (80km/h)	382
7	Speed limit (100km/h)	1289
8	Speed limit (120km/h)	1216
9	No passing	1309
10	No passing for vehicles over 3.5 metric tons	1792
11	Right-of-way at the next intersection	1172
12	Priority road	1827
13	Yield	1879
14	Stop	678
15	No vehicles	556
16	Vehicles over 3.5 metric tons prohibited	363
17	No entry	972
18	General caution	1072
19	Dangerous curve to the left	188
20	Dangerous curve to the right	318
21	Double curve	291
22	Bumpy road	341
23	Slippery road	440
24	Road narrows on the right	239
25	Road work	1327
26	Traffic signals	523
27	Pedestrians	214
28	Children crossing	479
29	Bicycles crossing	232
30	Beware of ice/snow	395
31	Wild animals crossing	680
32	End of all speed and passing limits	218
33	Turn right ahead	599
34	Turn left ahead	367
35	Ahead only	1058
36	Go straight or right	339
37	Go straight or left	187
38	Keep right	1828
39	Keep left	262
40	Roundabout mandatory	322
41	End of no passing	209
42	End of no passing by vehicles over 3.5 metric tons	212

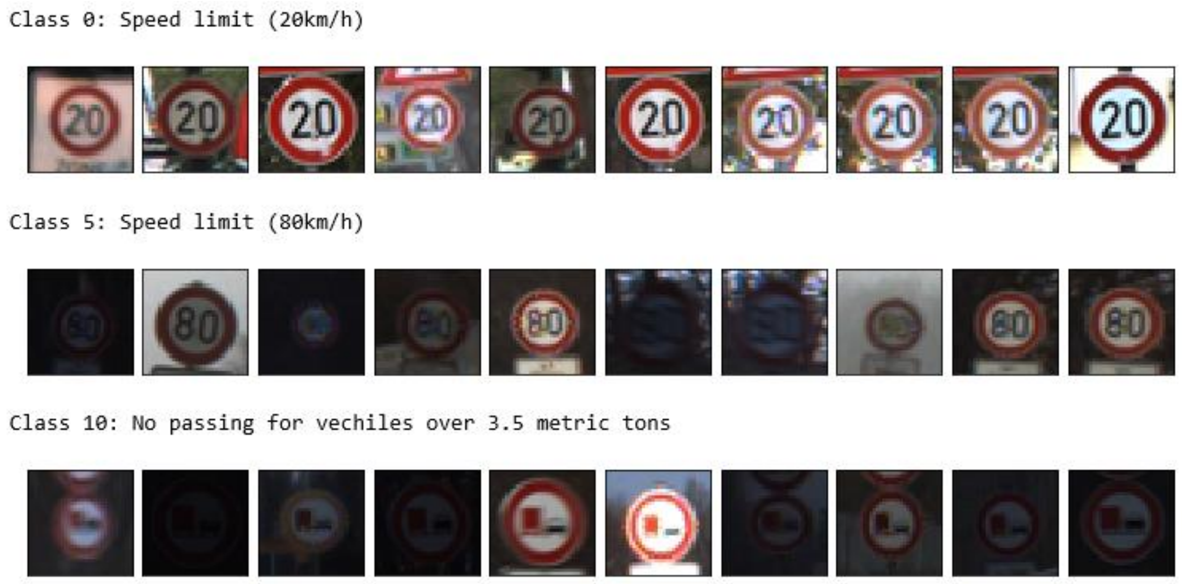
From the above list and the next exploratory visualization, we can conclude that some few classes in the training dataset have about 200 instances or less assigned to them, but most of the class labels have adequate number of training instances (more than 400 instances).

Exploratory Visualization

After loading the original training dataset and splitting that dataset into training and validation datasets with a percentage of (88% to 12%) subsequently, I have plotted the following histogram to show the frequency and the distribution of the training samples across the different 43 class labels. As usual, the X-axis of the histogram shows the encoding numbers of the class labels (from 0 to 42) and the Y-axis shows the number of instances belong to each class label.



Additionally, I plotted samples of the images in the training dataset from different class labels as shown in below (the details available in the Jupyter Notebook).


```
Class 0: Speed limit (20km/h)
Class 5: Speed limit (80km/h)
Class 10: No passing for vechiles over 3.5 metric tons
```

Algorithms and Techniques

During the analysis phase, initially we resized the raw images that have different sizes to have a fixed square size of (32 x 32) pixels, and this is actually a best practice to make all input images in a square fixed size shape. We also loaded the training and test datasets as numpy arrays with numbers range from 0 to 255, because this is the way that the computers can understand images. This technique to convert the raw images to array of numbers is essential when working with computer vision problems. The shape of the input data after using that technique will be (32 x 32 x 3) which represents the height, width and the number of channels; here the number of channels is "3" because they are RGB images.

Then we used normalization technique to rescale the input values to be in the range from 0 to 1, and this technique is used to make the mean of the data around 0 and the variance ranges from 0 to 1. We did that simply by dividing each number by 255 (The highest value in the range).

We also used data split technique to split the original training dataset to training dataset with a percentage of 88% and the validation dataset with the other 12%. This validation dataset is used during the training of the CNN model to calculate the validation loss and validation accuracy of the model and measures how the model improves and learns from the training data.

During the implementation phase of the solution model, we have used the feedforward and backpropagation algorithms during the training of our CNN model. Simply these algorithms are essential to enable the model detecting and learning the patterns in the data by updating the weights or the parameters of the different nodes in the network until getting the optimal weights that minimize the error function that represents the difference between the predicted classes and the ground truth labels.

In the CNN implementation part, we also used the dropout technique to protect against overfitting, and the data augmentation technique to increase the number of samples in the training dataset by generating different variations of the training images through applying different types of image transformations such as shifting in x-axis and y- axis, rotation with specific angles, zooming…etc.

Benchmark

We have considered two benchmark models as a reference to compare our final solution model against, these two benchmark models as follow:

- **The humans' model:** Human performance in recognizing the traffic signs is about **98.81%**.

- **The multi-scale convolutional network model [2]:** This model was developed by Yann LeCun and Pierre Sermanet and it had achieved accuracy of **98.97%** using convolutional neural networks. This model achieved state-of-the-art results in solving this classification problem by exceeding the humans' performance.

III. Methodology

Data Preprocessing

In the data preprocessing step, we loaded the data from the pickled version of the dataset that we prepared from the previous steps. I split the training dataset into training (88%) and validation dataset (12%) as mentioned earlier.

I rescaled or normalized the numpy arrays of the images in the training, validation and test datasets to make the range of numbers from 0 to 1. This normalization process is to make the mean around 0 and the variance ranges from 0 to 1 in the data distribution. We will do that simply by divide each number by 255 (The highest value in the range).

Finally I expanded the dimensions to have 4-D tensors with shape of (sample no., height, width, channels no.), these 4-D tensors in that new shape will be the input later to our CNN model. As part of my implementation, there is a function that converts 3D tensor to 4D tensor with shape (1, 32, 32, 3) and return 4D tensor which will be fed later into the CNN model.

Implementation

The steps that I took to get my final CNN architecture were as follows:

- Generally in my CNN model architecture, I used convolution window of size (3 x 3) and stride is 1 to move the convolution window one step at a time. I used these settings for kernel size and stride, because practically they give better results.
- Our CNN model starts from 32 filters in the first convolution layer (the input layer) and the number of filters increases slowly in a sequence as we go deeper in the network till reach 128 filters at the final convolution layer in my CNN model.
- I started with two convolution layers; each one of them has 32 convolution filters, the kernel size of the filter is (3 x 3) and stride of 1. Also there is a batch normalization layer right after each one of those convolution layers.
- The input shape parameter has been set to (image_height, image_width, channels) ➔ (32, 32, 3)
- The input for this model for training is a 4D tensor of (training_samples_number, image_width, image_height, channels) = (34503 , 32, 32, 3)
- I use "relu" activation function / layer after each convolution layer. It introduces non-linearity to the network and helps alleviate the vanishing gradient problem.
- After each convolution layer in the network, there is a batch normalization layer which makes the learning process faster, improves the higher overall accuracy and protect against overfitting as well. Also we used batch normalization to avoid "internal covariate shift" as batch normalization minimizes the effect of weights and parameters in successive forward and back pass on the initial data normalization done to make the data comparable across features.
- Then max pooling which is basically a down-sampling layer. Applying this layer after the convolution layers drastically minimizes the spatial dimension of the input volume (The width and the height change, but the depth remains the same).

- The max pooling layer in this model has a pool filter with size (2 x 2) and a stride of the same length (2). Practically these numbers for pool filter size and the stride length give better results.
- The max pool layer applies to each activation map from the previous convolution layer and it produces the maximum number in every sub-region that the pool filter convolves around.
- The main two benefits of max pooling here is to reduce the number of parameters which in turn reduce the computational cost of training the model and it also controls the overfitting.
- The first top convolution layers in the network try to find the patterns in the images data and figure out the low level features. As we go deeper in the network, the number of filters increases in the next convolution layers starting from 32 to be 64, 128, then 256 in a sequence to find more patterns within the patterns and so on till find the high level features in the training data at the deepest layers of the network.
- Then I flatten the final output of all the convolutional layers by using "Flatten" layer to flatten the output of the convolution Layers into a vector, to be fed into the next dense hidden layer of 1024 nodes in our model architecture.
- Then we have a fully connected hidden layer "Dense" of 1024 nodes with "relu" activation function applied.
- Then Dropout layer with a probability of dropping nodes = 0.5. This layer helps protecting against overfitting.
- Finally, we have the final output layer; it is a fully connected layer of 43 nodes to produce the final classification of 43 traffic signs, with a "softmax" activation function to convert logits or margin scores into probability for each class label.

After building that CNN model as described above, we compiled the model using **"adamax"** as an optimizer, "categorical cross entropy" as loss function and specifying "accuracy" as a metric for model evaluation. It's worthy to mention that I tried many optimizers such sgd, adam, and adagrad, but after doing cross validation by performing a randomized grid search across many optimizers, **"adamax"** was the best as shown in my final result in the code.

After that, we trained the model by fitting it to the training dataset. I set the batch size to **64**, and the number of epochs to **100** (I increased the epochs number to 150 after the data augmentation). I used two callbacks; one for early stopping with patience of 20 epochs and the other callback is "Model Checkpoint" to save the best weights of the trained model.

The test accuracy of this trained model before data augmentation step is **97.76%**

Finally, I converted that model to Flask App to be production-ready solution, this Flask App has a back-end part which includes REST APIs that encapsulates the prediction logic based on our final trained model, and the front-end part which includes a web page that enables the end-user to pick a traffic sign image and upload to the server, then click a button called "Predict Traffic Sign" to invoke the prediction end-point and shows the prediction result of that traffic sign image.

Refinement

I tried different architectures until I produced the final model described above. To fine-tune and optimize my solution model, I used the following techniques:

- Tune and optimize the hyperparameters of the model using grid search with cross validation technique. I used randomized grid search because it is more efficient and faster. I applied the cross validation technique to figure out the best settings possible for the optimizer, the number of the nodes in the hidden dense layer, and the dropout probability. We used the same batch size (64) and we run the training during the cross validation for 20 epochs. In that context, the final results show that the best optimizer is **"adamax"**, and the best number of the hidden nodes is **"1024"** with dropout probability of **"0.5"**.
- I did data augmentation for the training and validation data as shown in the code cells below. Data augmentation basically applies different kinds of transformations on the images such as shifting, rotating, translating, zooming, shearing...etc. It augments the number of the instances in the training dataset by generating different variations of the images and it improved the model performance a little. For the data augmentation, I tried different transformations until I got the ones I have finally used in my Jupyter Notebook. I avoided using vertical and horizontal flipping because they will change some traffic signs to another traffic signs, so the class label assigned to the traffic sign won't be correct in that case.
- Trying different architectures and different layers in my model. Initially I started with a simple model of three convolution layers; the first layer has 16 filters, the second has 32 filters, and the third one has 64 filters. Each convolution layer followed by a batch normalization layer and max pooling layer, then I optimized my model to get the final model described above by adding more convolution layers and fine-tune the optimizer, the number of the nodes in the dense hidden layer, and the probability value of dropout layer by applying cross validation technique.

IV. Results

Model Evaluation and Validation

The final solution model after applying fine-tuning and data augmentation techniques has been evaluated based on the test score or test accuracy using the test dataset. This test dataset has 12630 instances which is considerable amount of test instances that the model didn't see before. After evaluating the model against this test dataset, we got a test accuracy of **98.37%** which is very good results. This test score of the final model indicates that the model generalizes well to the unseen data.

As part of the model evaluation, I also calculated the precision and recall of the final model for each class label as well as the total recall and total precision. I already reported the details of these final results in my Jupyter Notebook. The following table summarizes the evaluation results of the final model:

Evaluation Metric	**Results**
Test Accuracy	98.3690%
Total Precision (On Average)	98%
Total Recall (On Average)	98%
F1-Score	98%

The formula of the F1-score as follows:

$$\mathrm{F1-Score} = 2 * \frac{Precision * Recall}{Precision + \ Recall}$$

Additionally, I plotted the confusion matrix that visualizes and shows the true positive probabilities for each predicted class label against the ground truth of the test dataset instances. The details of the confusion matrix and its plot are included in the Jupyter Notebook to keep this report shorter.

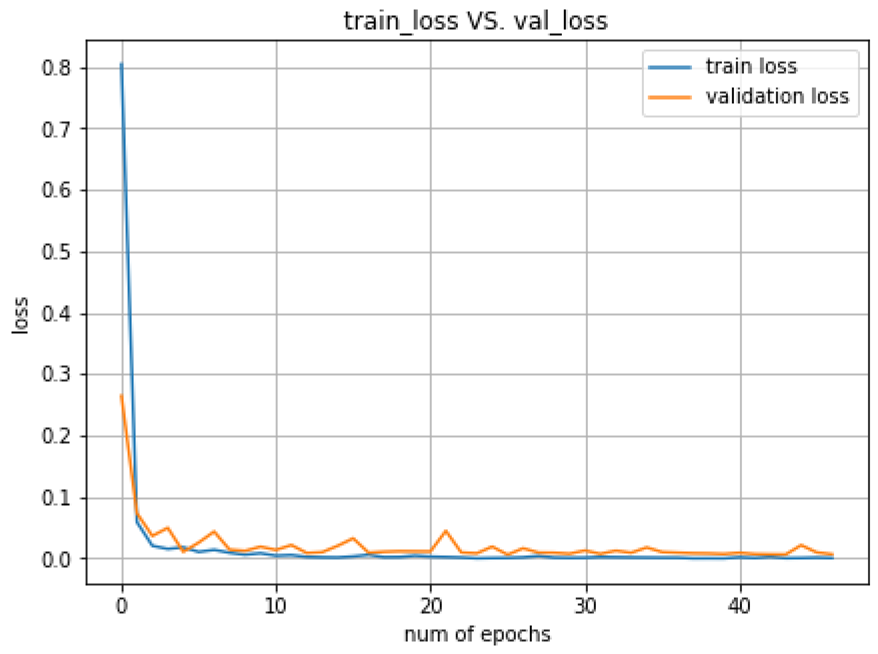
Justification

We set two benchmarks before as a reference models to compare our final solution model with. The first benchmark is the human beings who achieve 98.81% accuracy in recognizing and classifying the traffic signs, and the second one is the **multi-scale convolutional network model** which achieved 98.97% accuracy. Compared to these benchmarks, the test accuracy of our model is 98.37% which is really very near to the benchmarks' accuracy.

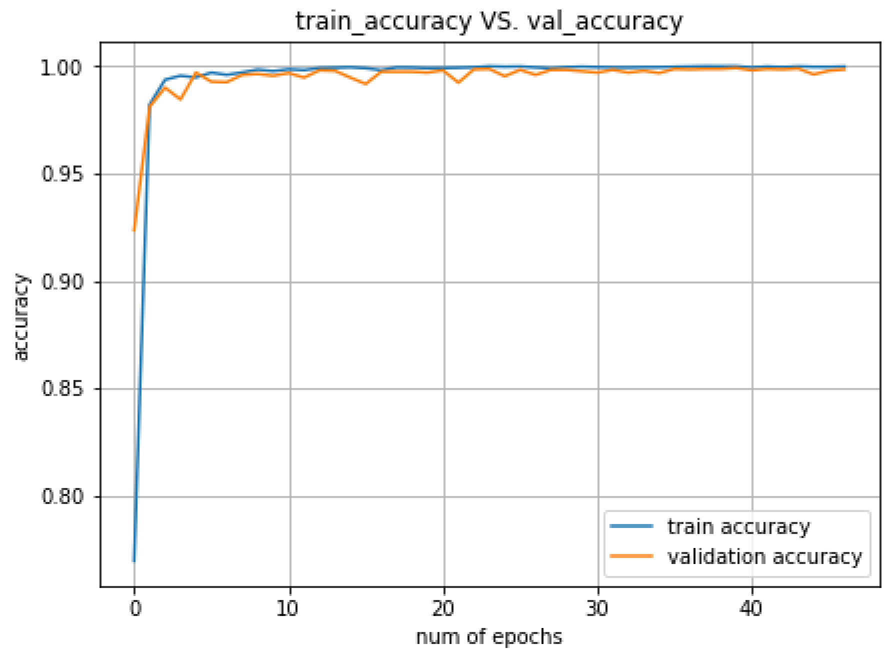
V. Conclusion

Free-Form Visualization

The following plot shows the training loss versus the validation loss of the trained model. The x-axis shows the number of epochs and the y-axis shows the loss value.



Alternatively, the following plot shows the training accuracy versus the validation accuracy of the trained model. The x-axis shows the number of epochs and the y-axis shows the accuracy.

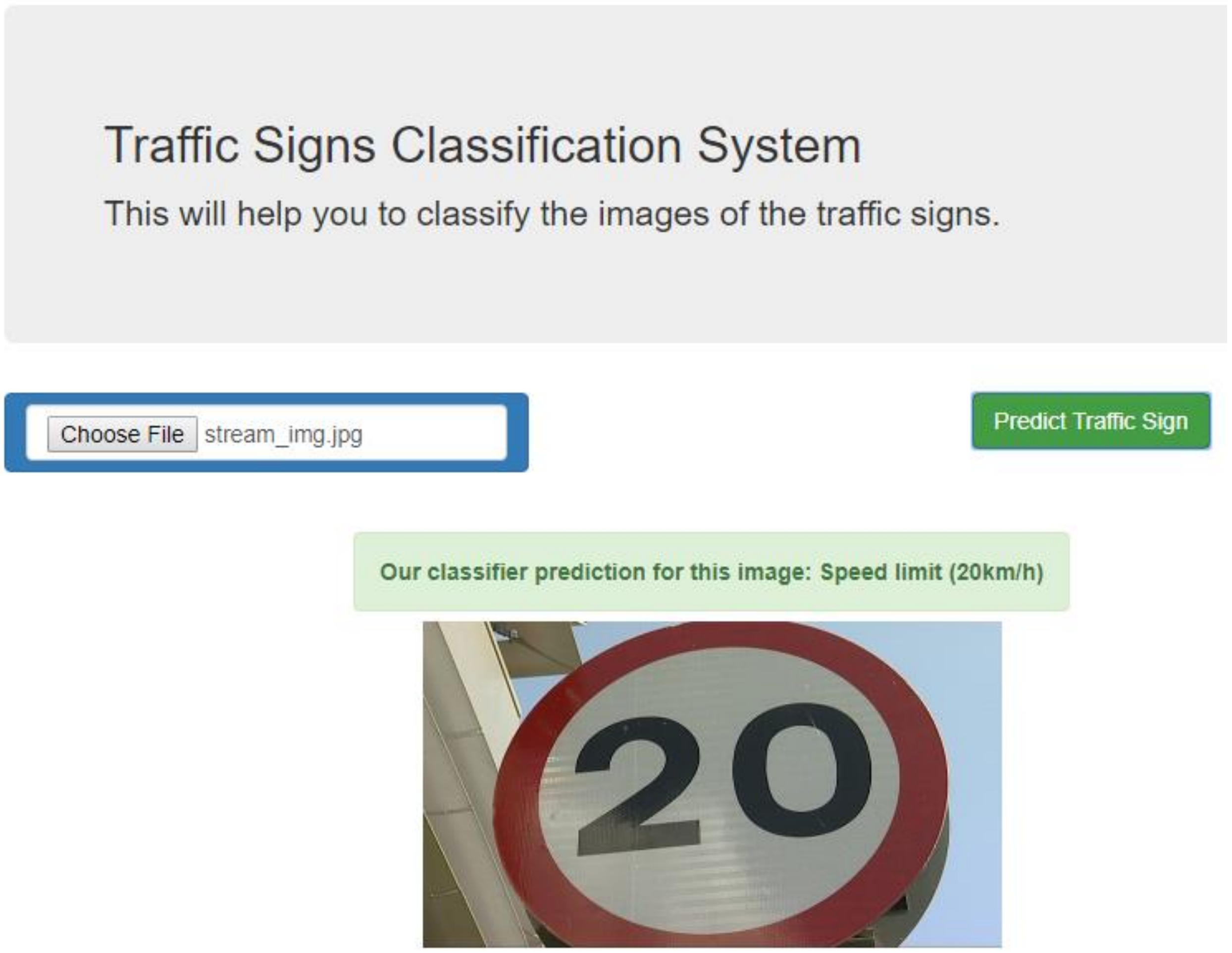


Reflection

The end-to-end implementation process went through the following steps:

- **Initial Preprocessing of the datasets:** This step was to convert the original dataset of the PPM files to pickle files that contain the numpy arrays to make loading the data and the further processing easier. PPM format is not so popular, so doing that step was really helpful for my work.

- **Loading the dataset & Data Exploration Phase:** In this step, I loaded the training and the test datasets from the pickle files, and then I split this original training dataset into training dataset (88%) and validation dataset (12%). After that, we hot-encoded the target labels which is a best practice in this kind of classification problems.
  As part of the data exploration, I plotted a histogram that shows the training samples distribution across the class labels. Moreover, I displayed samples of the traffic sign images for some classes in that training dataset (It's already available in the Jupyter Notebook).
- **Further Preprocessing of the data:** I did rescaling or normalization for the datasets to make the values' range from 0 to 1. We did this to make the mean at zero and the variance of the data ranges from 0 to 1. We also expanded the dimensions of the 3-D tensors to 4-D tensors in the shape of (sample no., height, width, channels no.) to make these inputs ready to be fed later into the CNN model.
- **Build the architecture of the CNN model & Train the model:** In this step, I built the architecture of the CNN model, and then I complied and trained the model using the inputs prepared from the previous step. I started with initial model with less convolution layers, then I expanded the model by adding more convolution layers of the model and changing the number of the filters till I reached the final architecture of the solution model presented in the Jupyter Notebook and explained above (in the implementation section).
- **Fine Tuning the Model:** In this step, I did cross validation using the randomized grid search to figure out the hyperparameters' values that produce better results. I applied that on the optimization algorithm (the optimizer), the dropout probability and the number of nodes in the hidden dense layer. After that, I did data augmentation on the training and the validation datasets.
- **Evaluating & Testing the Final Model:** In this step, I evaluate the final model after tuning and data augmentation using the test dataset. The test score or accuracy of this model was **98.37%** which is very good results compared to the benchmarks that have been set before.
- **Convert the Final Model to Flask App:** To make the final model available for end-users and to be production ready, I converted the final model to Flask App. Here is a screenshot of the flask app while predicting a traffic sign as shown below.


Traffic Signs Classification System

This will help you to classify the images of the traffic signs.

Our classifier prediction for this image: Speed limit (20km/h)

Improvement

There are some ideas to improve the solution model for that problem such as the following:

- Convert the RGB images in the original dataset to gray-scale images and use them as input to the CNN model, because I believe that the gray-scale images will work well for that problem domain and we can avoid dealing with the RGB images, which in turn will make the problem simpler and the dimensions of the input data less. Gray-scale images have only one channel, instead of 3 for the RGB images.
- Fine-tune more hyper-parameters of the model and try more different architectures by adding more convolutional layers or changing the numbers of kernels or filters of the convolution layer. Part of the fine-tuning is to apply other regularization techniques such as L1 Regularization (weight or kernel regularizer in Keras) and L2 Regularization (activity regularizer in Keras).+

References:

(1) http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset

(2) http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf

(3) The pickled version of the training and test datasets that I have created.