# Machine Learning Engineer Nanodegree

## Capstone Project: German Traffic Sign Classifier

**Ali Parsaei**
December 21, 2017

# I. Definition

## Project Overview

One of the classical challenges in self-driving cars has been correctly recognizing road traffic signs using vision. This is crucial to car's functionality since the computer needs to make a decision based on the signs installed on the road. Before the current era of computers, this task was considered infeasible due to the low speed of processors, the absence of proper methods to classify images, and possibly lack of proper data. Within the past two decades, thanks to the amount of data available on the web, the computational capabilities of the current computers, and various architectures introduced in the domain of convolutional neural networks, solving the problem of classifying traffic signs is feasible now. In this project, we will try developing a model to correctly recognize an image of a traffic sign taken from the German Traffic Sign Dataset.

Plenty of research has been done in the field of Convolutional Neural Networks (CNN). An absolutely relevant publication on applying CNN's to classifying traffic signs is Traffic Sign Recognition with Multi-Scale Convolutional Networks by Pierre Sermanet and Yann LeCun, which is the baseline for this project as well.

This project platform and data are obtained from Udaciy CarND Traffic Sign Classifier Project GitHub. The original dataset used for this project is the German Traffic Sign Dataset. Udacity has provided a pickled version of the dataset, in which the images are converted to (32, 32) color images for ease of use and smaller size. I used the same version of the dataset that can be downloaded from here.

Similar to a lot of Image Classification problems, the dataset required for training a model should include a set of images as the input and the corresponding labels (classes) as outputs (target variable). Similarly, this problem's dataset is composed of the images as inputs and labels as outputs. Moreover, the data is divided into training and testing datasets. The Training dataset and the Test dataset have 39209 and 12630 entries respectively. Note that every single input entry has a (32, 32, 3) (width, height, channels) image and every output entry (y) is an integer label between 0 and 42 (inclusive), which corresponds to a unique sign among 43 traffic signs. Another file that is included in the dataset is a CSV file that represents what each class ID/label represents. For example, Class ID = 14 represents "Stop" sign. This is for a better understanding of the data and sanity check of the results.

## Problem Statement

As mentioned in the previous section, one of the important tasks that need to be done on a self-driving car is to identify the traffic signs that the camera on the cart captures while moving. One way to solve this problem is to train a model on images of different traffic signs that have been recorded already, then use the compiled version of the model as a pipeline, which gets an image of a traffic sign as input, and classifies the image as one of the traffic sings that it has been already trained on. This result can be fed into the decision making part of the system to decide what to do based on the sign.

Based on the dataset, a proper way to accomplish the task of predicting the label of an image is to create a pipeline that takes in images as inputs, preprocesses the images, feeds the processed data into a CNN with multiple layers (which needs to be trained and tuned), and finally outputs the class ID of the image. In this project, we follow a similar procedure. More precisely, we perform the following steps:

1. First, we load the data from the files to the script. The inputs (X) are going to be sets of images and the outputs or target variables (y) are going to be sets of labels.

2. Then, we need to divide the data into training (training, validation) and testing datasets. The data used for this project have been already divided into the training and testing datasets. Therefore, we only divide the training dataset into training and validation datasets.

3. After defining the datasets, we need to take a look at the data to have a better intuition of the nature of the data points.

4. The dataset does not have missing entries or other issues that require treating; therefore, we only preprocess the data in order to be appropriate to feed into the algorithm.

5. After preprocessing the images and labels, we create our model architecture, which takes in the preprocessed images as inputs and provides the most likely label for each image. The type of dataset and goal of the problem are well-suited for a Convolutional Neural Net (CNN). We need to define an architecture that takes images into the neural net and outputs Softmax probabilities (set of probabilities of which each entry corresponds to the likelihood of the image belonging to that class ID).

   The CNN architecture used in this project is a variation of the LeNet architecture with two convolutional + max pooling layers as well as two fully connected layers. The inputs are normalized grayscale images and outputs are Softmax probabilities.

6. After defining the Architecture, evaluation metrics are defined.

7. In this step, the model is trained using the training dataset and validated against the validation set. This step is probably the longest section of the problem, where the architecture and its weights need to be tuned. After the algorithm works well on both training and validation datasets, the final model is stored in a local file.

8. After the model is successfully trained, validated, and stored in a local file, the model is restored to test the test dataset. In this step, we need to make sure we can attain the evaluation metric defined in the proposal and if not, tune the model to reach our goal.

   Figure 1 summarizes the data flow in this project:

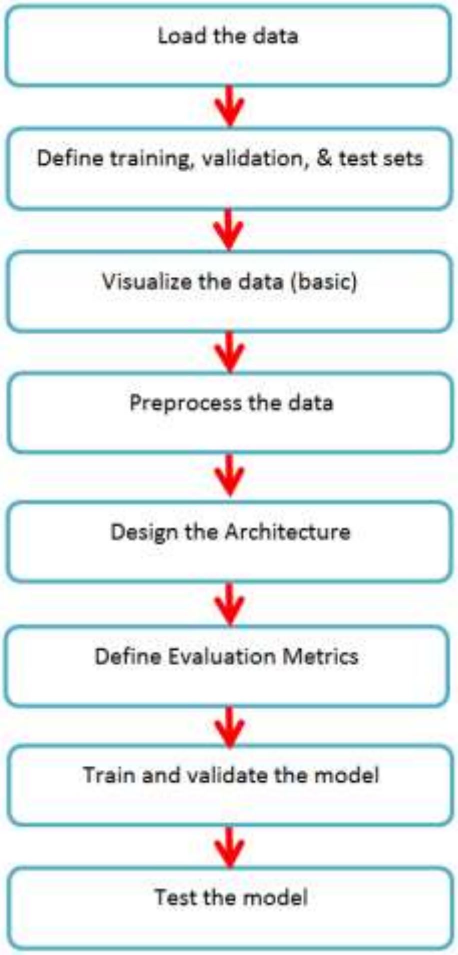
Load the data
Define training, validation, & test sets
Visualize the data (basic)
Preprocess the data
Design the Architecture
Define Evaluation Metrics
Train and validate the model
Test the model

Figure 1: Data Flow

## Metrics

Different evaluation metrics can be used in classification problems. The common performance metrics used for classification problems are accuracy, precision, and recall. There is always a tradeoff between recall and precision. In some of the problems such as detecting cancerous tumor cells, a high recall model is desired, while for some other problems such as the spam detection, a high precision model is desired. Note that both of the mentioned examples here fall into binary classification, and the reason that recall and precision have different priorities roots in consequences of a correct or incorrect classification for a certain class. On the other hand, in the general form of our problem, which is a multi-class classification, the accuracy of the model seems to be the best criterion to evaluate the performance of the model. The reason behind this statement is the fact that in general, it is equally important to detect all the signs correctly; therefore, neither recall nor precision has higher importance over the other. Note that in reality, correctly detecting some of the signs may have higher importance compared to others; however, in this project, we treat all the signs equally in a general situation.

***Passing Requirement***

Based on the introduction provided, the evaluation metric used for this problem is the **accuracy of the prediction**, which is mentioned in Benchmark as well as the project proposal. My goal is to attain an accuracy of **90%** or better over the test set images to indicate a successful result.

# II. Analysis

## Data Exploration

The ultimate goal of this project is to correctly classify a traffic sign image into one of 43 traffic sign labels. The dataset for this project should include images as inputs and labels as outputs. As expected, the dataset used for this project includes images and corresponding labels. More precisely, the pickled dataset is a dictionary (map) with 4 key/value pairs:

- **'features'** is a 4D array containing raw pixel data of the traffic sign images, (number of examples, image width, image height, channels).
- **'labels'** is a 1D array containing the label/class ID of each traffic sign in the dataset. **'signnames.csv'** contains Class ID → Name mappings for each sign.
- **'sizes'** is a list containing tuples, (width, height) representing the original width and height the image. *This entry is NOT used in this project.*
- **'coords'** is a list containing tuples, (x1, y1, x2, y2) representing coordinates of a bounding box around the sign in the image. Note that these coordinates assume the original image, while the pickled data contains RESIZED versions of these images (32x32). *This entry is NOT used in this project.*

Note that the only entries used for this project are **'features'** and **'labels'**. Also note that the pickled dataset is divided into training and testing datasets. I divided the training dataset into training and validation sets. Table 1 includes important statistics on the pickled datasets used in the project.

**Table 1: Statistics of used datasets for this project**

| Size of training set | Size of validation set | Size of testing set | Image shape | Number of classes (labels) |
|---|---|---|---|---|
| **29406** | **9803** | **12630** | **32x32x3** | **43** |

According to Table 1, the training, validation, and testing datasets have 29406, 9803, and 12630 data points, respectively. Each input image is a 2D colored image with 32x32 pixels and 3 channels (Red, Green, and Blue). For example, the following image is the first entry in the training set, of which corresponding class ID is **2**.

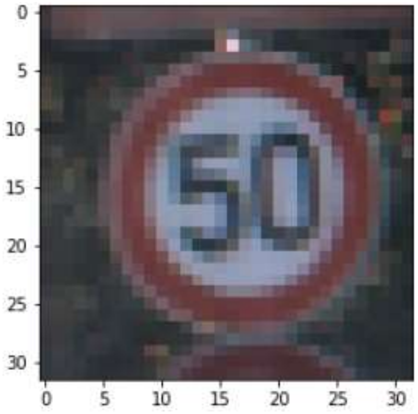

Figure 2: First Image in the pickled training dataset

On the other hand, '**signnames.csv**' has only 43 entries, in which each entry is a class ID and its corresponding label in English. For example, '**0,Speed limit (20km/h)**' and '**42,End of no passing by vehicles over 3.5 metric tons**' are the first and last entry in this dataset, respectively.

Fortunately, these datasets do not have any missing or noisy data points. Therefore, no extra work is required to clean the datasets.

## Exploratory Visualization

Note that a Jupyter IPython Notebook is attached to this project, in which all the steps in the project are included. In this notebook, several sample images are provided for each label (traffic sign class ID) for a better understanding of the data. You can refer to the attached notebook for further insight on the data. In this report, on the other hand, we have provided the histogram of the data for each class ID for training, validation and testing datasets.

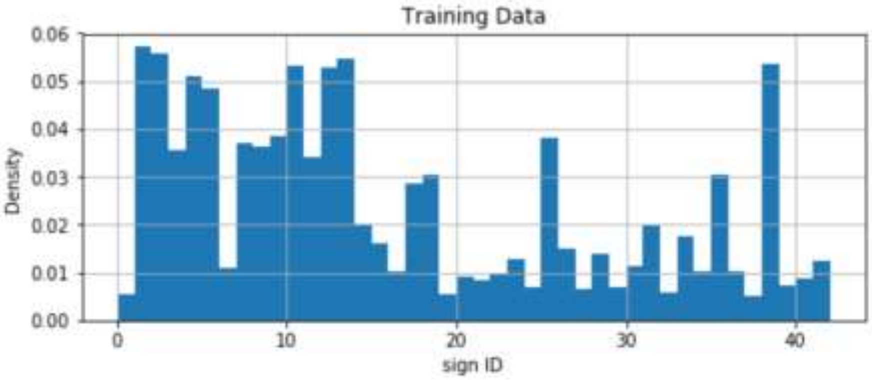

Figure 3: Histogram of the training data

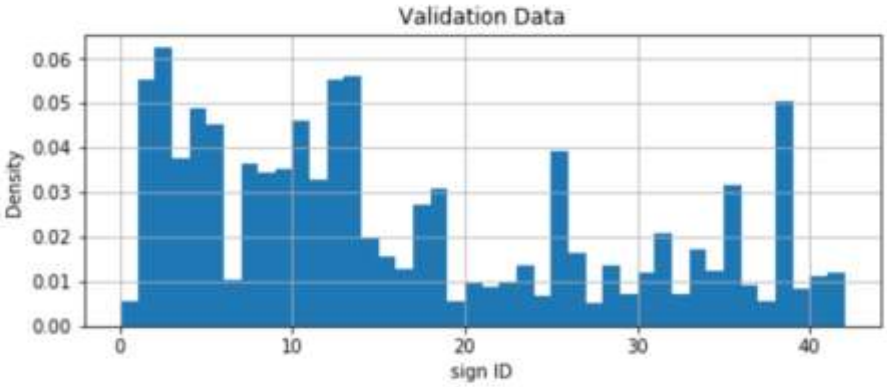

Figure 4: Histogram of the validation data

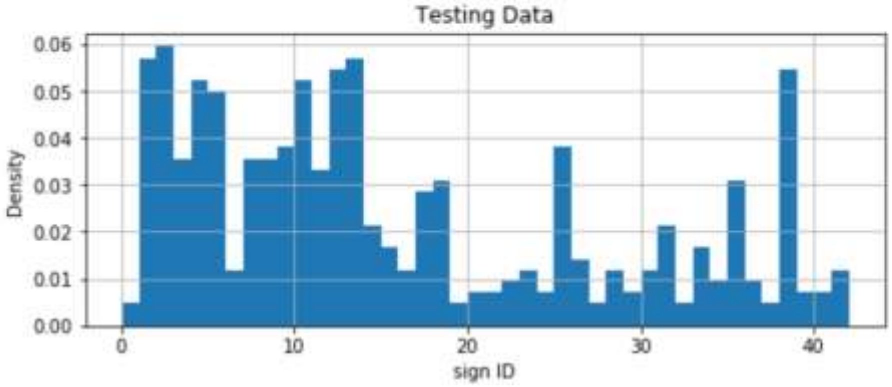

Figure 5: Histogram of the testing data

The first thing to check in these histograms is to see if all the classes are represented in the datasets, i.e. the density of all classes is non-zero, which is the case here. Moreover, we check to see if the data are uniformly distributed, which is desired for machine learning problems. This means that each class is equally represented in the training; however, this is not possible in a great number of real-world problems. One way to deal with this case is to not use all the data for the training and randomly remove some of the data points with labels that are present in higher numbers. However, this will significantly decrease the size of the datasets. Based on the fact that the size of the datasets is not large, this would probably be detrimental to the training performance; therefore, we keep all the data points.

On the other hand, you can observe that the distribution of the data is almost identical in training, validation, and testing datasets, which is desired. This means that the performance of the model is captured more accurately, which is intended for every project.

## Algorithms and Techniques

Convolutional Neural Nets (CNN’s) are a family of neural networks that have shown great performance in classifying 2D images. In classic methods which used deep neural nets for image recognition, the input images would be flattened into a 1D vector and fed into the first layer of the neural net. CNN's, on the other hand, feed the 2D image into the neural net. This is doable by employing a set of convolutional and pooling layers in the initial layers of the net. The output of last convolutional and pooling layers are typically flattened and fed into fully connected layers at the end of the neural net. The number of units in the last fully connected layer of the CNN should be equal to the number of classes in the problem.

The number of layers and units for each problem varies based on the complexity, diversity, and size of the datasets. Usually, the initial architecture is set based on experience and then the model is modified and tuned based on the training results. Note that not every individual has experienced a similar problem to use their personal experience to build the architecture; therefore, studying the literature is always a great way to build a base architecture for image recognition problems.

LeNet Architecture defined in Traffic Sign Recognition with Multi-Scale Convolutional Networks by Pierre Sermanet and Yann LeCun is absolutely relevant to this problem, which forms our baseline for the model used in this project. Figure 6 is the architecture used in this problem, which is slightly different from the architecture used in the mentioned publication. This is a sequential architecture in which the input of each layer is the output of the previous layer.

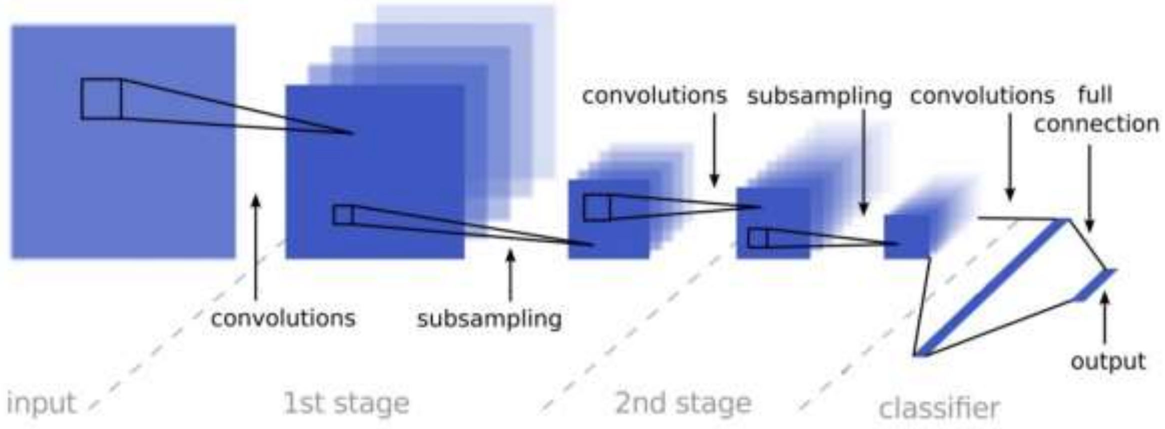

Figure 6: LeNet Architecture used in this Problem

The input to the model is 2D images and the output layer is a Softmax probabilities for 43 class ID’s. In each stage, a window with a fixed size and the desired number of filters to convolve the data with are defined. Then the input of the layer is divided into smaller areas with the window size, and the filter is convolved with the input data in each small area. All the results from each filter are gathered into a convolutional layer. The convolutional layers in each stage go through a ReLU activation layer, which is then fed into a pooling layer. The output of the pooling layer is sent to the next stage as input layers.

The output of last pooling layer is then flattened and fed into a fully connected layer. The output of the fully connected layer goes through a ReLU activation layer and fed into the last fully connected layer which has 43 units, which is equal to the number of sign class ID’s.

Note that the parameters trained at each stage are weights (including the biases) of each linear classifier. These weights and biases are initialized randomly and with a constant array, respectively. The values of the parameters are then tuned by back-propagation in solving an optimization problem.

## Benchmark

As mentioned in the overview section, the accuracy of the classification is the criterion used to evaluate the performance of the model. An impressive publication done on this project is the Traffic Sign Recognition with Multi-Scale Convolutional Networks by Pierre Sermanet and Yann LeCun. Based on the mentioned paper, human's accuracy in correctly classifying the traffic signs using 32x32 colored images is 98.81%. The accuracy attained by Sermanet and LeCun in the mentioned paper is 98.97%, which outperforms human being.

Based on the fact that this paper has used a more advanced version of our model, we do not expect to attain such a performance. As mentioned in the proposal, the target accuracy for a successful performance in this project is **90% or better**. Note that 90% is still a decent performance based on the fact that the some of the images are taken in the dark and are not even recognizable by human beings.

# III. Methodology

## Data Preprocessing

2D images are appropriate inputs for CNN's. Note that a variety of creative data preprocessing could still be performed on the input data to improve the performance of the model. Feature scaling is one of the simplest and most effective image preprocessing methods. Another preprocessing step that can improve the performance of the model is converting the colored images into gray-scale images. This is particularly helpful due to the relatively small size of the data. Based on the curse of dimensionality, the more dimensions we have for the data, the more data points we need to generalize the data. Therefore, by averaging the data over the channel axis, we effectively reduce the number of data points we need to 'learn' well.

In a nutshell, the following steps have been taken in data preprocessing on the images:

1. All the colored image pixels are converted to gray-scale by averaging the values over the channel axis.

$$Pixel_{grayscale} = \frac{1}{3} \sum_{c=\ r,g,b} pixel_c$$

2. The gray-scaled images are then scaled so that each pixel's value falls into [-1, 1] range. Thus, the value for each pixel in gray-scaled images goes through the following transformation:

$$pixel_{preprocessed} = \frac{pixel_{grayscale} - 128}{128}$$

Moreover, the output values are also categorical; therefore, we need to make sure that they are appropriate for the architecture. An easy and effective data transformation used for categorical data is One-Hot Encoding. Note that the value of the labels varies from 0 to 42. Therefore, each label is encoded into 43 values, of which one entry has value 1 and the rest are 0. For example, if the value of class ID for an entry is 2, the corresponding encoded value is [0, 0, 1, 0, 0, … , 0].

## Implementation and Refinement

In the first step of the implementation, the focus is on the data and preparing the data to feed into a CNN. After dividing the data into training, validation, and testing sets, all the images are transformed by gray-scaling and then scaling the gray-scaled pixel values to fall into [-1, 1] range. Note that visualization of sample data points before and after preprocessing are also performed for sanity check as we move forward.

After preparing the datasets and preprocessing the data points, the variation of LeNet architecture used in this project was implemented using TensorFlow. The number of layers was chosen based on the baseline model acquired from the mentioned publication by Sermanet and LeCun. However, the number of units except for the last layer was set randomly.

Now that the architecture is defined and the data are preprocessed to be fed into the mode, we need to frame an optimization problem and solve the problem iteratively. In order to do so, we need to define the loss function and solver first. The loss function here is a categorical cross-entropy and the optimizer used is ADAM, which is a general-purpose adaptive stochastic gradient descent solver.

The next step before launching an iterative optimization session is to implement the evaluation metric. In this problem, the accuracy, which is the ratio of the correct predictions over the total number of predictions made is the criterion used in the training.

After preprocessing the data, implementing the architecture, and implementing the required methods for a training session, we launch a training session with aid of TensorFlow. In this training session, we define the number of ‘epochs’ to train the model, while we output the accuracy of the model after each epoch on the training and validation sets.

After the optimization session is launched and the training and evaluation are finished, the final result of the session, which is a neural net with known weights, is stored within a local file. This model can be called in a program to perform a prediction on a preprocessed (set of) image(s).

After the model is successfully saved, the model is restored to be tested on the test dataset to find the performance of the model on data points never seen by the model. The accuracy attained on the test dataset is the real-life accuracy of the model.

Note that this problem has a large number of parameters that can be tuned to acquire better results; however, this will be too time-consuming. Therefore, following, I have listed the crucial (hyper-) parameters that can be tuned for obtaining better results:

- Architecture: Number of units in each layer
- Optimization:
    - Learning rate
    - Batch size
    - Number of epochs

Note that there is a reasonable range of values that make sense for each parameter. The criteria that we observer by changing the values of the parameters are the accuracy (training, validation) of the model as well as the time it takes to train the model. Following, every parameter is briefly discussed:

- Number of units: we started with (64, 128, 256) number of units respectively for 1st, 2nd, and 3rd layer. This took a long time to train by CPU. Therefore, decreased the number of nodes up to (32, 64, 128) units in multiple steps. The accuracy of the model did not decline, while the model became way faster to train for each epoch. We could continue this path to find the minimum number of the nodes required to train the model well. However, we stopped here since the model is already relatively simple. Following table is results for the initial and final number of nodes.

**Table 2: Comparison of Training Duration V.s. Number of Units in the CNN**

| Number of Nodes | (64, 128, 256, 43) | (32, 64, 128, 43) |
|---|---|---|
| Training Time (10 Epocs) | 4000 secs | 1500 secs |

- Learning rate: we started the model with a high learning rate of 0.01; however, this did not provide a very good accuracy. Then we decreased the learning rate and captured the model performance. Learning rate of 0.001 provided a good performance while keeping the speed high enough. Following table is results for the initial and final learning rates.

**Table 3: Comparison of Performance V.s. Learning Rates**

| Learning Rate | 0.01 | 0.001 |
|---|---|---|
| Training Time (10 Epocs) | 1380 secs | 1500 secs |
| Validation Accuracy | 10 – 80 % | 99.2% |

Note that when a high learning rate is used, the results can vary a lot in every trial, which is not a good characteristic of a model. The worst accuracy I attained with 0.01 learning rate is 10% and the best is 80%. However, the results with 0.001 learning rate are consistent. Moreover, note that the time spent for training is not significantly different in the two cases.

- Batch Size: I started with larger batch sizes to find the performance with faster models. The initial batch size was 256; then I used 128, 64, 32, and 10. The accuracy of the model improved as the batch size became smaller up to batch size of 32, while the learning duration slightly grew longer as the batch size grew smaller. Following table displays the results for 3 batch sizes.

**Table 4: Comparison of Performance V.s. Batch Size**

| Batch Size | 256 | 32 | 10 |
|---|---|---|---|
| Training Time (10 Epocs) | 1240 secs | 1500 secs | 1550 secs |
| Validation Accuracy | 94.0 % | 99.2% | 98.7% |

- Number of epochs: for each of the variations mentioned above, I started from a low number of epochs such as 5 and then changed the value of epochs to reach a point where the training accuracy does not change significantly. Ideally, this could be automated by using early-stopping tools defined in TensorFlow. However, I did this step manually as well.

  In another effort, I ran the model for 14 epochs. The results are shown in the following figure.

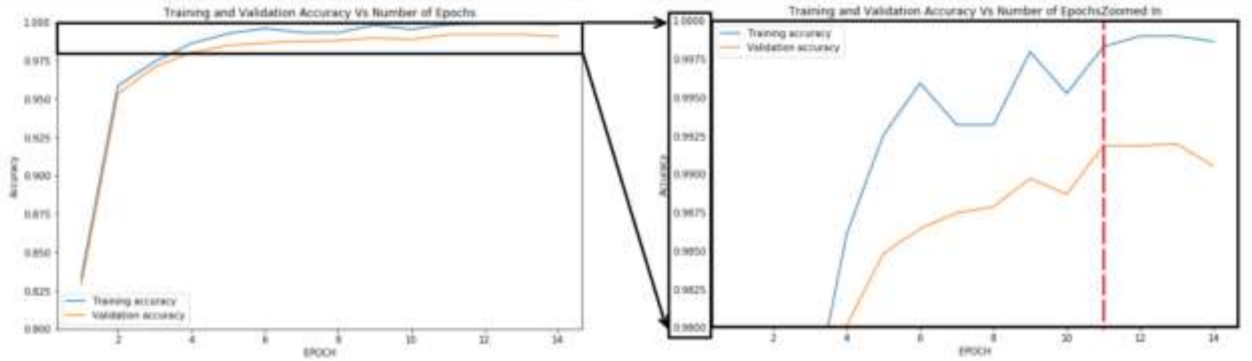

Figure 7: Training and Validation Accuracy of Model V.s. the Number of Epochs

In the figure you can observer that the validation accuracy of the model does not improve from Epoch 11. Therefore, it makes sense to stop the training at/near epoch 11.

Note that during the optimization sessions, I outputted both training and validation accuracies. The reason behind this is to spot any overfitting in the model. Overfitting occurs when the performance of the training set improves BUT the performance of the validation set deteriorates.

As expected, I observed some overfitting in the results, which needs be addressed. In order to do so, first, we need to determine the factors that can lead to overfitting. The major factors that can lead to overfitting in such a problem are as follows:

- Number of layers
- Magnitude of the weights and biases
- Number of the nodes (units)
- Number of epochs (with is positively correlated to the number of training steps).

Following, we explain what has been done regarding each factor to avoid overfitting.

- Number of layers: Note that based on the nature of the problem, we have used the minimal number of layers that can work out for this problem; therefore, we should not worry about the number of the layers.
- Magnitude of the weights and biases: We have not explicitly controlled the value of the weights and biases; however we initialized the weights and biases with tiny values; this will be helpful in not having very large weights and biases.
- Number of Nodes: As discussed before, we tried to use few numbers of nodes in the first place; however, we did not make sure to use the minimum number of nodes required to train the model. It is common in training CNN's to use more number of nodes that the minimum required, but use

‘Dropouts’ to avoid overfitting. This means that in each epoch, we randomly exclude a certain portion of the nodes in training.  We used the dropout layer before the final dense layer, in which we dropped 40% of the data in training.

- Number of epochs: In order to address this factor, we start from a small number of epochs and increase the number of epochs. The technique used to control the number of epochs is ‘Early Stopping', which could be automated by TensorFlow and Keras modules in Python; however, this can be done manually as well, which is what I did in this project. Early stopping means we continue training as long as the training accuracy improves significantly in each epoch. By comparing the accuracy of this project model (Figure 7) with the model complexity graph (Figure 8), you can see that we have stopped the training right before overfitting occurs.

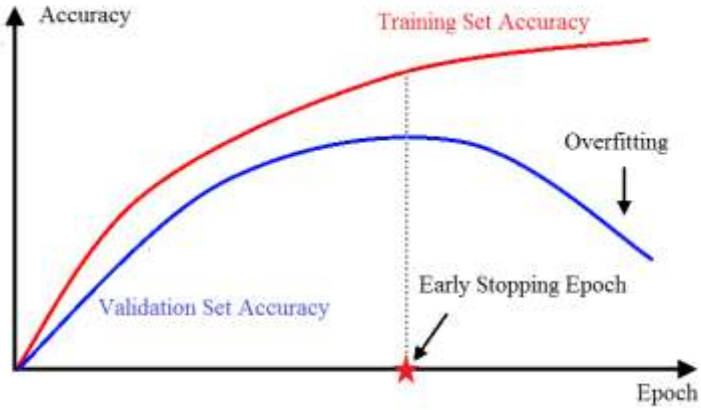

**Figure 8: Model Complexity Graph**

In a nutshell, I attenuated overfitting by using few number of nodes and using a dropout layer, initializing the weights and biases with small values, and finally, stopped the training when the model training accuracy did not improve significantly. At the same time, I checked the accuracy of the model on the validation data set to make sure overfitting does not occur.

# IV. Results

## Model Evaluation and Justification

After tuning the initial model, we observed a better performance in the model on the training and validation sets. After saving the final model in a local file on the computer, we restored the model using TensorFlow built-in methods and tested the model on the testing dataset. Following table summarizes the performance of the model on training, validation, and testing datasets.

Table 5: Performance of the final model on different datasets

| Dataset | Training | Validation | Testing |
|---|---|---|---|
| Accuracy | 99.97% | 99.18% | 95.58% |

If the model perfectly generalizes a dataset, and the datasets are similar, the accuracy of the model on all the sets should be almost identical; however, overfitting is a very common issue in CNN's trained on small datasets. As explained before, we tried to remove or mitigate the overfitting factors from the model to get a better generalization, which resulted in accuracies that are close to each other. The difference between the performance of the model on the training and testing datasets is less than 5%, which is a good quality for this project.

Based on the fact that the size of the testing set is larger than 40% of the size of the training set, and also the fact that the testing dataset is composed of a variety of images with different angles, translations, and lightning, performance attained on the testing dataset is absolutely reliable.

As you can see in Table 5 the model's performance on the testing dataset is about 95.6%, which is above the target accuracy we defined in section II. Note that the difference between the testing performance of the model and the paper on which this project is done (~ human's accuracy) is less than 4%, which is good due to the fact that we only used a single laptop CPU for a limited time and the fact that the model used in this project only has 1 layer as input and 1 layer as output for hidden layers.

# V. Conclusion

## Free-Form Visualization

In section IV, we discussed the results obtained from this project and the accuracy of the prediction on the test images; in this section, we downloaded a few images from the web to perform the analysis on.

In order to do so, first I wrote a pipeline which takes the file path of an image on the local machine and predicts the label of the image. The model predicted the labels of all images correctly, which is a good sign of the model's performance.

Moreover, I created another pipeline that takes in an image path and provides the top 5 likeliest predictions on the image label. This is done to get a better intuition of the confidence of the model's prediction on different images. Following image displays the 32x32 colored images used in this step, the prediction, and the probability distribution of the predictions for each image.

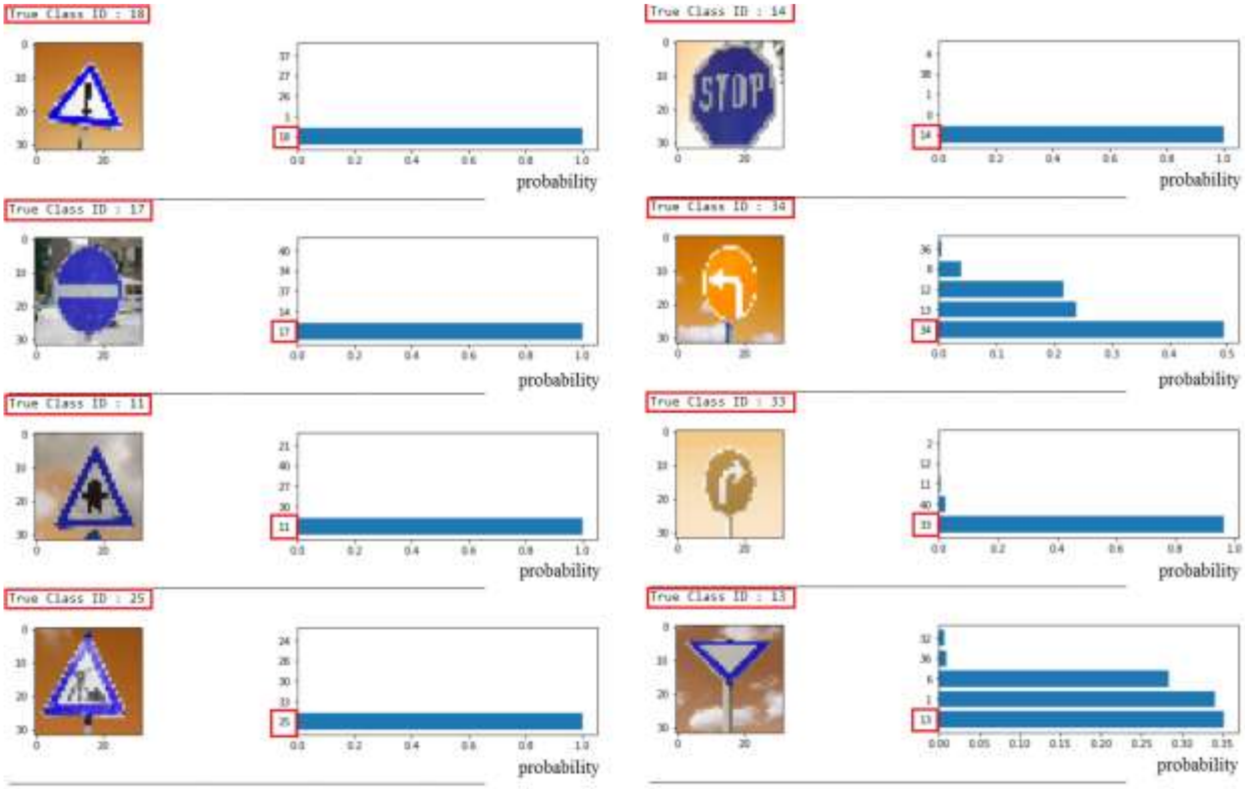

**Figure 9: Likeliest predictions on images from the web**

As you can see in the image, the algorithm is 100% confident on the prediction of 5 images, almost confident of 1 image, and not very sure about 2 images. Note that analysis of this phenomenon needs more time and energy; however, by looking at these sings, it seems that the model is confident on images with distinct features, such as 'Stop' and 'Right-of-way at the next intersection' signs with sign ID's 14 and 11 respectively; however, the prediction of 'Yield' sign with class ID 13, which does not have any specific features rather than its triangular shape, seems to have a lot of uncertainty.  Note that this is expected based on how CNN's work in image classification.

If you look at the last image in figure 9, you can see that the sign in this image is not centered in the image. One important note in this project is the implied assumption of the algorithm of receiving images with a sign almost in the center of the image. Therefore, in order to get a good performance for any image, another step should be added to the preprocessing step that detects the boundary of signs, sets a certain area near the sign as the region of interest, and feeds only the region of interest to the model rather than the whole image.

Another characteristic of the last image in figure 9 is the angle from which the picture is taken. Note that most of the images in the dataset are taken from a straight point of view, which means that the weights of the graph are optimized to predict images with the straight point of view. Thus, the point of view in this image affects the performance of a CNN algorithm in image recognition. One way to address this issue is in the image acquisition step, where the camera on the autonomous car can be set in a way that detects images with a proper angle.

One way to address both the translation and point of view problems in image recognition is ‘Data Augmentation’. This means that (a portion of) images in the training dataset are randomly translated horizontally and vertically as well as tilted up to a certain range. Due to the relatively small size of the dataset, the original images should be kept in the training dataset too. This has been proven to improve the performance of the CNN's that are trained with a small number of training samples.

## Reflection

In this project, we aimed to design, train, and test a model to correctly predict the label of an image taken from a traffic sign.

The main dataset provided for this project is composed of 32x32 colored images of 43 traffic signs as well as their corresponding labels. After loading the datasets into the IPython Notebook and keeping them in local variables, the datasets were divided into training, validation, and testing datasets. After creating the training, validation, and testing datasets, multiple samples of each traffic signs were visualized to get a better understanding of the dataset. After visualizing the datasets, the images in the datasets were preprocessed by gray-scaling and normalizing the images to be appropriate for feeding into our model. Besides, the labels of the traffic signs were One-Hot encoded to be used as output targets variables in our model.

After preprocessing the data, the CNN architecture of the model was designed based on LeNet architecture. The first and second stages have one convolutional and one max-pooling layer each. The output of the second stage is fed into a fully connected layer whose output will be fed as input into the output layer – a fully connected layer with 43 units. Note that both convolutional layers and the fully connected hidden layer go through ReLU activation layers as well.

After implementing the graph with aid of TensorFlow, the parameters are initialized using random values. Moreover, the required parameters for training the model such as the loss function, and the optimizer were defined before the next step. The loss function and the optimizer for this project are ‘Categorical Cross-Entropy’ and ‘ADAM’ (an Adaptive Stochastic Gradient Descent Optimizer). Moreover, the number of epochs to train the model is also defined at this stage.

After constructing the graph, initializing the parameters, and defining the parameters and methods required for training the graph, a TensorFlow session is launched in which the model weights are trained using the training dataset, and the model is validated against the validation dataset. When the performance is good, the resulting graph is saved into a local file for future applications. After the model is successfully stored, the model is restored in another session to be tested on the testing dataset, of which result will portray the real performance of the model.

Finally, an pipeline is designed which takes in the file path of a 32x32 colored image and predicts the label of the image. This is particularly useful in a self-driving car application, where a colored picture of a traffic sign is taken, converted to a 32x32 image, and fed into the pipeline. Moreover, another pipeline with similar structure was developed which intakes a colored 32x32 image and outputs the top 5 predictions for the image label as well as their likelihood. This pipeline is designed to analyze the confidence of the model in its prediction. Several new images were then fed into these pipelines and the results were analyzed.

After all, it is fascinating to see how a simple LeNet architecture can provide about 95% accuracy in classifying traffic sign images. Such a task was deemed near impossible a couple of decades ago, while this and more sophisticated image recognition tasks can be done in a short time with the current technology.

The main challenges in this problem are the finding the proper CNN architecture as well as tuning the parameters of the architecture. Luckily, a proper architecture was already developed for this problem which set the baseline for our model; however, the parameters still had to be tuned. Tuning the parameters was mostly done by educated trial and error. This step was the most time-consuming part of the project, were proper values need to be found by changing a parameter and observing the impact on the performance. One point to keep in mind is that the best result of the model is acquired with a COMBINATION of parameter values. This means in order to assure we obtain the best possible result with a certain architecture, we need to create a grid search for all hyper-parameters and parameters and measure the performance of the model with various combinations. Then refine the model in the range of interest by narrowing down the search. This is not done in this project since this would require a great amount of processing power to complete in a reasonable period of time. This is the reason tuning was indeed the most challenging part of the project.

## Improvement

We observed that the testing accuracy of the model, which is about 95%, exceeds the criterion we set in the benchmark, 90%. Therefore, the algorithm is good enough for the purpose of this project; however, there are steps that can be taken to improve the performance of the algorithm to reach (or exceed) human's performance. Following are some of the notable actions that could help improve the results of this project:

- Collecting more data: If more images of various traffic signs are gathered, especially images of signs which are under-represented in the problem's dataset, the performance of the model can be improved. This point is helpful for almost all machine learning projects whose dataset size is not large. More data can help in generalization as well as enabling us to use more complicated models without overfitting the data.
- Data augmentation: By randomly selecting a subset of the training data, and then shifting and/or rotating the images by random values in a small range, we could help the algorithm in its generalization. Moreover, by keeping the original images while adding the new augmented images to the dataset, we would provide more data points, which would be helpful in generalizing the model according to the first point.
- Batch normalization: Based on the previous sections, we scaled the pixel values to fall in between -1 and 1 before feeding the data into the first layer. However, we did not scale the value of the inputs to next layers. Batch normalization is a technique which allows us to scale the data before

feeding into each layer, in a way that the dataset has zero mean and unit standard deviation. This usually furthers our performance using the same architecture.

- Early stopping using Callbacks: We tried to stop the training by modifying the number of epochs manually. Note that the best epoch to stop the training differs due to random nature of the stochastic learned. By using Keras Callbacks we could systematically stop the training at the best point.
- More complicated architecture: We could use a more complicated data flow in the architecture to acquire better results. This would be slightly challenging since it could lead to overfitting; therefore, this point would probably require a good amount of time to improve the results.
- Transfer Learning: We could use pre-trained architectures such as VGG16, VGG19, ResNet50, etc. as a base architecture and add a simple architecture on top of the pre-trained model to train the model. This would require GPU to complete within a reasonable period of time and a good number of trials to reach an exceptional performance.
- Regularization: We used a dropout layer in our architecture to decrease the chance of overfitting; however overfitting still occurred up to a certain level. By penalizing the model for large weights, we could probably avoid overfitting up to a good extent. Note that L1 regularization could be used to limit the number of non-zero weights, which could be helpful for feature selection. On the other hand, L2 regularization could be used for the more general purpose of limiting the magnitude of the weights.
- Grid search: A systematic way of finding the combination of parameter values which provides the best performance is to create a grid search for all the (hyper-)parameters we want to vary. One can start the grid search by searching a broad range and narrow down the range step by step to systematically get the best results.

# VI. References

- https://www.slideshare.net/butest/an-introduction-to-machine-learning
- https://www.tensorflow.org/tutorials/wide_and_deep
- https://www.tensorflow.org/get_started/mnist/beginners
- https://www.tensorflow.org/get_started/mnist/pros
- http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf
- https://www.tensorflow.org/get_started/mnist/pros#build_a_multilayer_convolutional_network
- https://github.com/udacity/CarND-Traffic-Sign-Classifier-Project
- http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset#Citation
- https://d17h27t6h515a5.cloudfront.net/topher/2016/November/581faac4_traffic-signs-data/traffic-signs-data.zip
- https://machinelearningmastery.com/improve-deep-learning-performance/
- https://deeplearning4j.org/earlystopping
- J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In Proceedings of the IEEE International Joint Conference on Neural Networks, pages 1453–1460. 2011.