Master CORO M1

Master in Control and Robotics

Project Report

June 2018

New labs for the Computer Vision module

Author:
Ashok Muralidharan
Thuc-Long Ha

Supervisor:
Bogdan Khomutenko
Olivier Kermorgant

Contents

1 Introduction

With the advent of high computing power and large amount of images being generated every day, deep learning techniques are starting to play a crucial role in computer vision applications. A number of challenging applications such as image classification, object detection and localization, face recognition, etc.. are now being solved efficiently, thanks to deep learning. The key essence of deep learning are two things - large amount of training data and good computing power. Deep learning algorithms are proven to perform well with more training data, thus has the ability to achieve accuracy in certain problems at par or some times better than humans.

1.1 Objective of the project

The primary objective of our project is to design a new lab material for the computer vision module. The main aim of the lab material was decided to illustrate the importance of deep learning in computer vision applications, different techniques and technologies used for developing such algorithms quickly and efficiently. The task that was chosen was image classification. The task was to build an image classifier that should predict the road signs using a deep learning technique called Convolutional Neural Networks(CNNs). Few example images of the task are shown in the figure 1. The entire code of the project is available in this Github link: here and also in the appendix of the report.

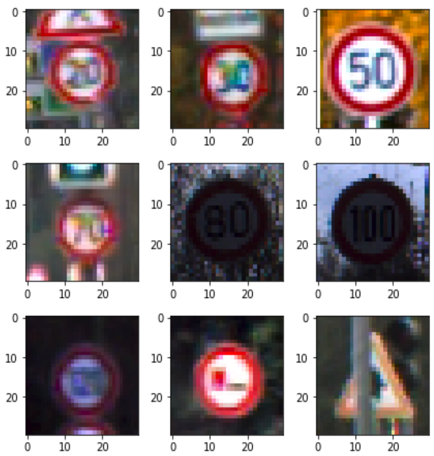

Figure 1: Traffic Signals for classification

2 Background

2.1 Image classification

2.2 Convolutional Neural Network(CNN)

Convolutional Neural Networks (CNNs) are the specialize Neural Networks which have been extremely effective in artificial intelligence tasks such as image, video classification and recognition. CNNs have been used intensively in identifying objects, human faces and traffic signs.

LeNet was one of the CNNs which pioneered the researches of Deep Learning. LeNet5[6], which is developed by Yann LeCun. During the past, LeNet was classically used for reading OCR and character recognition, etc.

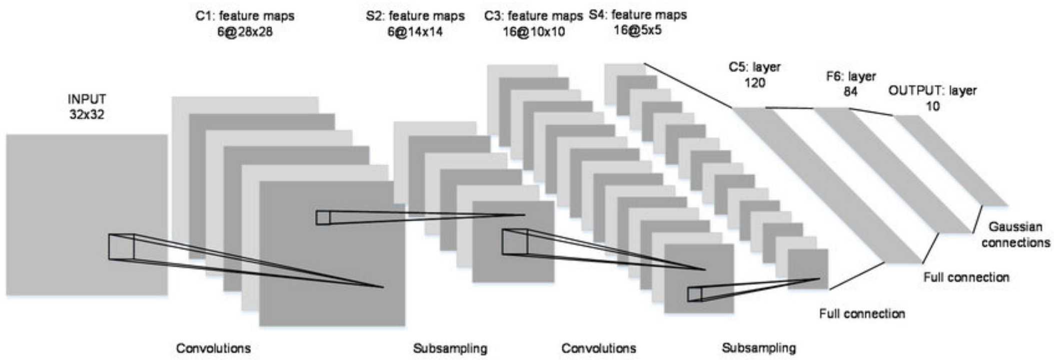

Figure 2: Lenet5 Architecture[6]

The Convolutional Neural Network above is identical to the original LeNet used for character recognition works.

The main operations in the CNN shown in Figure 1 are:

- Convolution (CONV)
- Non Linearity (ReLU)
- Pooling or Sub Sampling (POOL)
- Fully Connected Layer (FC)

Convolution:
We can imagine convolution in deep learning as a dot product of two vectors. The

pixels of the image is convolved with corresponding values in the filter (i.e) the pixels are multiplied with the corresponding filter values are are added to form the output. Thus the convolved output will have reduced dimensions than that of the original input. It is just like applying a filter in image processing. A typical example is shown in the figure 3

$$\begin{bmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -5 & -4 & 0 & 8 \\ -10 & -2 & 2 & 3 \\ 0 & -2 & -4 & -7 \\ -3 & -2 & -3 & -16 \end{bmatrix}$$

Figure 3: Convolution - Vertical edge detection

ReLU - Nonlinear Activation Function

As we can see that neural networks consists of bunch of neurons connected in linear fashion performing multiplication and addition ultimately. But many functions are non-linear and linear model might just not be good enough to represent the data. To add non-linearity to the CNNs we use the ReLU(Rectified Linear Unit) as activation function. ReLU is a function that thresholds the value to zero. The function is can be represented as $f(x) = max(0, x)$. Graphically representation of the function is shown in the figure 4. Apart from ReLU, there are many other activation functions such as sigmoid, tanh, etc.. that are used in deep learning.

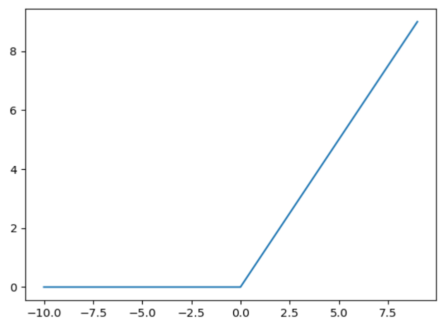

Figure 4: ReLU

Pooling or Sub Sampling:

The next major part of CNNs are pooling layers. The commonly used pooling functions are max-pooling and average-pooling. But max-pooling is the most adopted one. We can imagine max-pooling as something like taking the max value among a region that we see in the input. This way we try to reduce the dimension of the input, by generalizing based on the max information considering the adjacent values. The figure 5 should give a clear idea of max pooling.

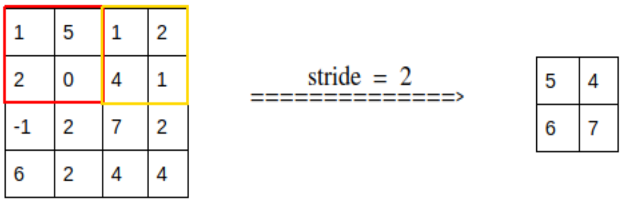

Figure 5: Max Pooling - 2X2 with stride 2

Fully connected layers:
The last major component in CNNs are fully connected networks. Fully connected networks are just the feed forward deep neural networks. Generally, in CNNs, one or two layers of fully connected neural network layers are added. The feed forward networks consists of number of neurons which are connected fully with each other, which performs multiplication(with weights) and addition(with biases) operation. Again a non-linearity is necessary here, which is called activation function. The figure 6 should give a brief idea of deep neural networks.

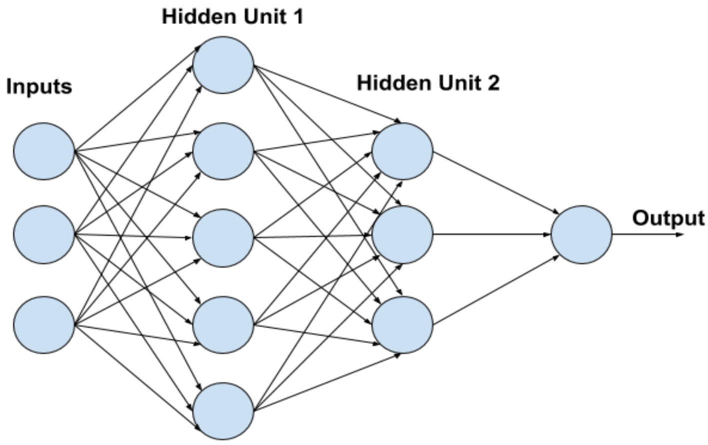

Figure 6: Fully Connected Networks

Apart from full connections, we also have biases which is basically a number that is added. Biases are used to shift our activation function to left or right. For brevity, we are not getting into the details of deep neural networks working, but here is what a single neuron(say in hidden layer 1) does

$$Inputs = X1, X2, X3$$
$$ConnectionWeights = W1, W2, W3$$
$$Bias = b1$$
$$Output = ActivationFunction(W1 * X1 + W2 * X2 + w3 * X3 + b1)$$

Similarly, every neuron will multiply the inputs and weights that it receives, add a bias and apply activation function such as ReLU it. This forward computation in neural networks is called as the forward propagation.

Back Propagation:
The learning algorithm that is generally used for the neural networks is back propagation. As the name suggests, back propagation involves propagation of error backwards in the network by computing the gradients at each step and updating the weights and biases which are the learning parameters. The main intuition here is to consider the CNN as one giant optimization problem, have all weights and biases as the system variables and error to decrease or optimize.
This process happens during the training phase. Initially we randomly initialize weights and biases. We pass the input through the network(forward propagation) and compute our prediction. We then find the error(prediction value - actual value). We then compute the gradient with respect to weights in the previous layer that affects the error.

$$\frac{\partial error}{\partial W1}$$

We can use the chain rule for updating the weights deep in the layer. An example is shown below.

$$\frac{\partial error}{\partial W2} = \frac{\partial W3}{\partial W2} * \frac{\partial error}{\partial W3}$$

Thus by updating the weights towards the direction of gradient, and doing this iteratively, we will minimize our error, thus move close to the actual output.

3 Dataset

3.1 Structure

The data set that we used in the lab were obtained from German Traffic sign dataset [9]. The images are in ppm format. Training and test images are divided into two different folders for facilitating easier retrieval of images during the process.

Training Images Location: /GTRSB/Final_Training/Images
Testing Images Location: /GTRSB/Final_Test/Images

Each Image has size that varies between 15x15 to 250x250 pixels. So we can observe that preprocessing of images is a necessary step here to make the dimensions of the images similar. Also the actual traffic sign is not always centered within the image.

3.2 Training set

The training folder contains 43 folder named from 0 to 42 indicating the classes to which the images belong. There is a 'csv' file in every folder which can be used to read the image and the corresponding class of the image. There are a total of 39209 images as a part of the training dataset. Some of the sample images are shown in figure 7. As we can see, the dataset has lots of variance in lighting, orientation, scale, etc.

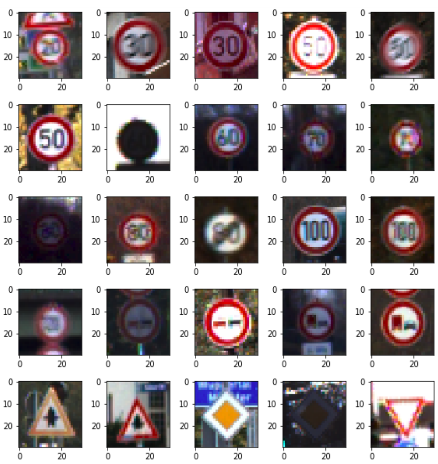

Figure 7: Training Images

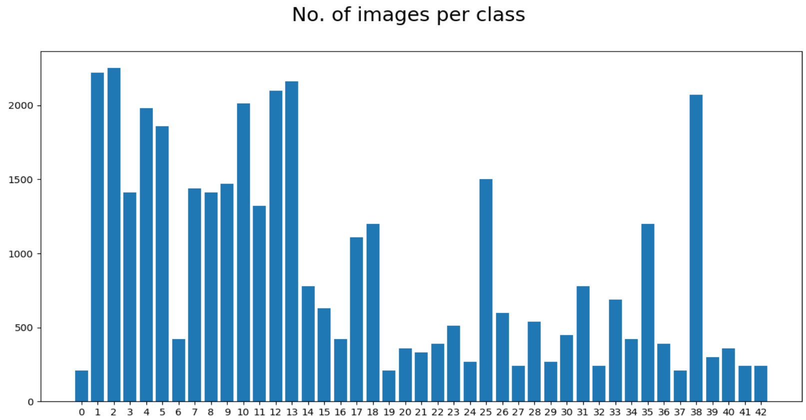

Figure 8: No. of images per class

As we can see from 8, the number of images per class is definitely not uniform. But the distribution is not very bad either. One way to over come this uneven distribution is to find more data to add to the training samples. This is always possible when we take our training data on our own. But here since we are working on a provided dataset, we are not adding additional images. We can also perform image augmentation from the original image to get duplicate augmented images and add them to the dataset. But adding a lot of augmented images might affect the network generality in real world. We have not augmented images here, but have used augmentation during the run time which will be discussed later.

Validation set:
In deep learning, there is always a validation set, which is genrally an extract from the training set. This set will be used during the training process for validating our model for every epoch(one complete pass through the training set). This becomes essential for early stopping of the training if our training process doesnot improve after few iterations. This validation set will not get involved in the training procedure.

3.3 Testing set

The test folder contains 12630 images. The file GT-final_test.csv labels the class and properties of each image. Some sample images are shown in figure 9. These images will not get involved in the training procedure and will be purely used for testing purpose.

Figure 9: Test Images

4 Implementation

4.1 Software and technologies used

The following are the programming languages and major technologies used for our purpose

- Python - programming language
- Numpy - Scientic computation library
- TensorFlow - open source machine learning framework from Google
- OpenCV - Open Source Computer Vision Library
- Git - Version Control System(VCS)

Python is the preferred language for deep learning for computer vision applications since it facilitates fast prototyping and has excellent community support. TensorFlow is a popular framework for machine learning applications which was developed and has support from Google. TensorFlow plays the major part of CNN implementation. This software stack is very popular and prevalent in the computer vision community. We also use GIT for the purpose of version control of the code.

4.2 Architecture of CNN

The architecture which we used is shown in the figure 10. We did not go with industry standard, state of art architectures like inception [10], VGG-16 [8], etc. because the network in these architectures are quite big and training will take a lot of time which might not be feasible to do in the labs. So we followed a pattern in which the network has CONV, CONV, POOL, CONV, CONV, POOL layers stacked in order. This architecture was chosen after some trial experiments with layers and with some inspirations from the internet. This architecture is fairly simple, so that this can quickly run in the labs as well as produce decent results. You can see the architecture in the figure 10.

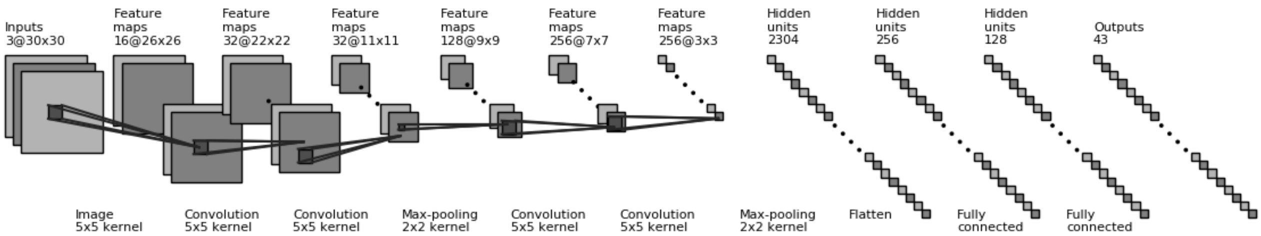

Figure 10: CNN Architecture [3]

As we can see the image that we use is of dimension (30X30X3). Thus it is an RGB image with width and height of 30 pixels. The image is passed through a series of CONV, CONV, POOL, CONV, CONV, POOL layers followed by two fully connected layers. The dimensions after each operation performed in each layer is shown in the figure 10. The operation performed is shown in the bottom of the layer and the resultant dimension is shown in the top of the layer.

4.3 Major steps in implementation

- Image Preprocessing and Augmentation
- Data Preparation
- Forward Propagation

- Back Propagation
- Model Evaluation, Logging and Saving

Image Preprocessing and Augmentation:
Since our model expects a (30X30X3) image, we need to resize the image to that dimension. We use OpenCV for this purpose. Initially, we tried using gray scale image for the purpose, but since traffic signs have colored content in them with specific meaning, we decided to go with RGB image later. Also with RGB image, the accuracy seem to slightly increase. We actually got a very good inspiration from an online article regarding about the augmentation of images [11] where the images are augmented during the computation process. During initial phases of training, we add more augmentation but at later stages of training we reduce the augmentation so that the model fits to the training data set well. This had a small impact on our accuracy improvement. Some of the augmented images are shown in figure 11.

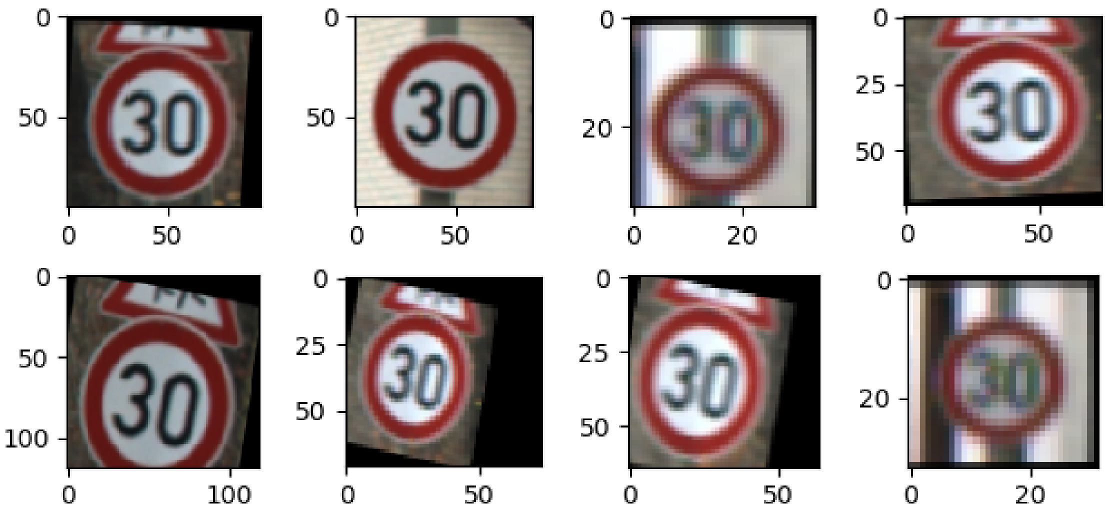

Figure 11: Augmented Images

Data Preparation:
Data preparation is an important step in the deep learning applications. Since deep learning works with data, proper structuring of data is necessary for getting desired results. Here we need to collect the image and split them to training set, validation set and testing set. These three sets are standards that are followed in deep learning community to enable proper evaluation and testing of the model. In order to predict the class, we design the outputs of the network as a onehot encoding. A onehot encoding is a typical structure used for classification problems. A onehot encoding has only 1s and 0s in it indicating the presence and the absence of the particular class(in our case vector of size 43).

Forward Propagation:
As explained earlier, the forward propagation of input through the network is done to get the predicted output. The forward propagation in general consists of series of multiplications and additions. The image is passed through a series of CONV, CONV, POOL, CONV, CONV, POOL layers followed by two fully connected layer in our architecture. We have already seen about the functions of each layer. We can imagine CONV layers as number of filters and these information are merged to get our final prediction. The visualization of activation output after passing through CONV layers is shown in figure 12. The activation layers 2,3 and 4 in figure 12 has more filters but we have only visualized few of them here. Our final output here is a vector of dimension 42(0-43) which indicates the presence or absence of particular class. TensorFlow lets us build these steps in a very easy and elegant fashion. The codes are available in the appendix section for reference. Thus the output of our forward propagation is one pass through the entire network producing output.

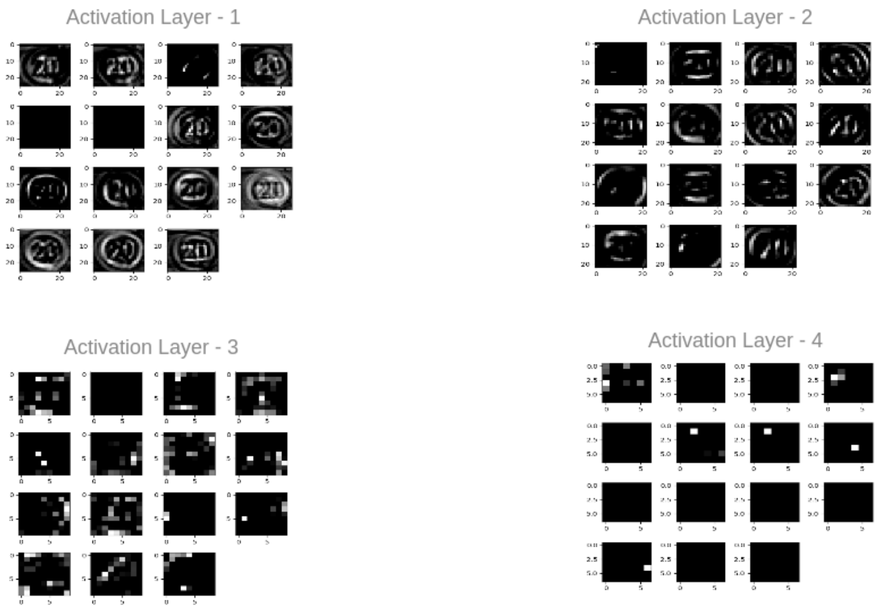

Figure 12: Activation Layers Visualization

Back Propagation:
The learning algorithm in neural networks is back propagation. The algorithm was explained earlier where we compute the gradients of the error with respect to every parameter of the model. It is during the back propagation where our model generalizes to the given training inputs and outputs.

Model Evaluation, Logging and Saving:
In deep learning applications, proper evaluation of the model at every steps during the training process becomes crucial as it provides us with a metric of knowing how well our model is performing or improving. Since CNNs involves lots of calculations with lots of images, the training process is painfully slow in nature. Not having a evaluation of model might result in spending enormous amount of time and resource on a bad model. With the evaluation in place, it helps us in early stopping of the training process, when we do not see any improvement in our learning. A model evaluation is something like determining the accuracy of the batch, loss function in the batch etc.. This is a very good practise that is commonly followed in the deep learning community. Also logging results such as accuracy, loss, etc.. onto the console like shown in figure 13 during the training helps us get a glimpse of what is happening with each epoch. We can also write the results to a file, but it is not implemented as a part of the project. Also saving the model after every few iterations is a nice way to have different checkpoints which can be used to retrieve the model later. TensorFlow provides elegant ways for performing all these operations. We saved a copy of the model for every five epochs. This way we can use the best performing model from the list of models.

```
############ EPOCH 5 SUMMARY: ############
Copy of model saved...
Cost after epoch 5: 0.041192
Validation Data Accuracy: 99.7448980808 %
############################################
```

Figure 13: Logs in console

5 Results

5.1 Result from our developed architecture

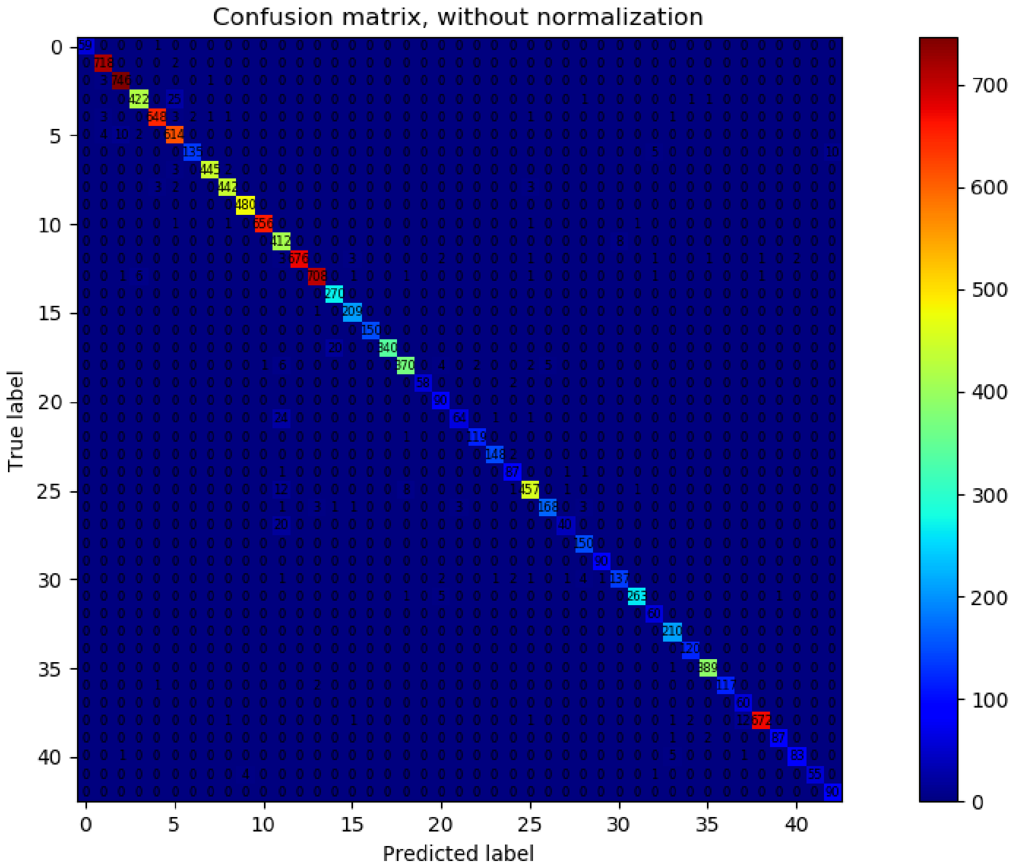

Figure 14: Confusion Matrix [4]

The confusion matrix in the figure 14 gives us a glimpse of overall classification performance. From the confusion matrix, we could see that the most prediction result concentrate in the diagonal line which is true positive. However, test data are uneven distributed in some classes. It is a matrix of actual values and the predicated values with count as the each cell of matrix. This lets us study class to class misclassification(i.e) how our model gets confused with classes. Hence the name confusion matrix.

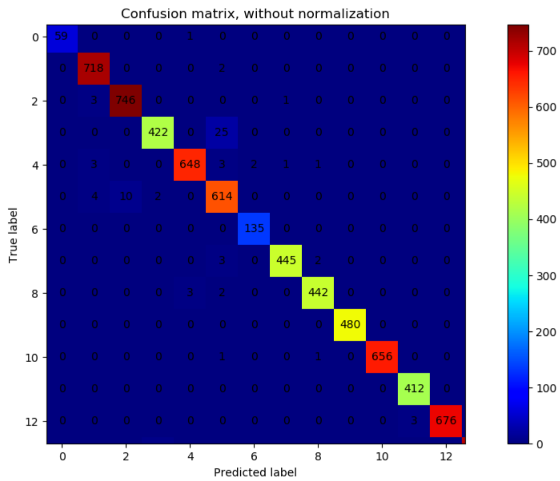

Figure 15: Zoomed - Confusion Matrix

The zoomed version of the confusion matrix is shown in the figure 15. As we can clearly see that the class 3 was wrongly predicted with class 5, 25 times.(i.e) 25 images that belong to class 3 was predicted class 5 by the network. Now let us examine what classes 3 and 5 are. The figure 16 shows the classes that were confused. It is evident that these two classes have similar features in them(like circle with red border, number 8 and 6 having similar curves, etc.).These classifications can be tricky.
Thus we should do some study on our analysis and not just settle on our accuracy alone as the metric.

Figure 16: Class 3 and 5

The figure 17 shows some failed predictions of our model. We could see that there are still some wrong predictions among the classes with similar features such as shape, color, etc. We did see that images with similar features gets wrongly classified from the confusion matrix. This is definitely a drawback of deep learning applications as debugging the reason for failure in these cases is quite difficult task. The best solution for improving accuracy is collection of more training data. Further tuning of hyper parameters such as learning rate, type of optimizer, etc., modifying the model architecture, might result in improved accuracy. We did not find time to play around with our architecture for improving accuracy but we managed to achieve model generalization and decent test accuracy of about 97.2%

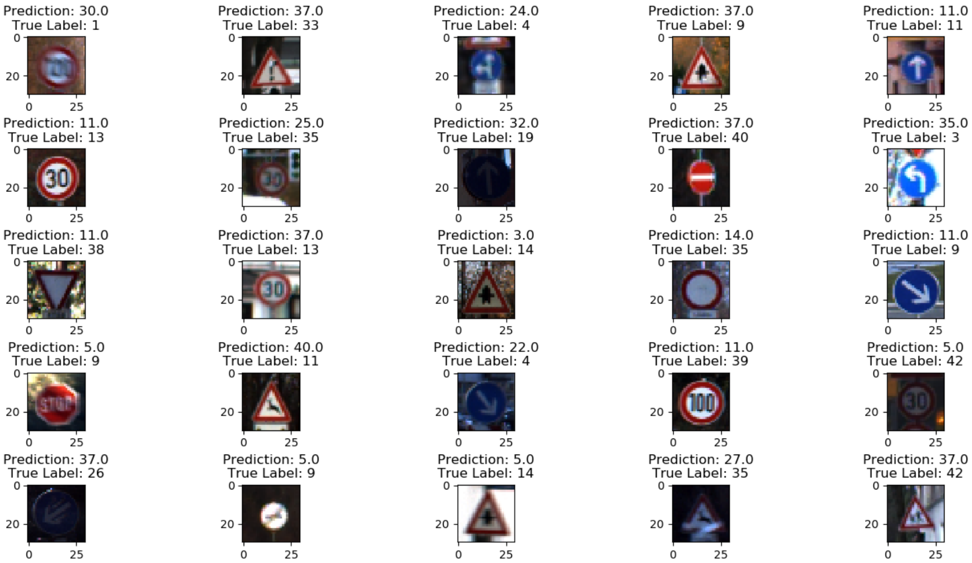

Figure 17: Failed Predictions

5.2 Furthur evaluation:

		Predicted Value	
		TRUE	FALSE
Actual Value	TRUE	True Positive	False Negative
	FALSE	False Positive	True Negative

Figure 18: True, False - Positives and Negatives

True Positives (TP)- These are the predicted positive values which are correct. So the value of real class is 'yes' and so is the value of predicted class. In our case, if actual class value indicates the sign and the sign you actual see the same thing.

True Negatives (TN) - These are the predicted negative values which are correct.It means the value of actual class is 'no' and so is the value of predicted class. In our case, if actual class value indicates the sign does not belong to one specific class and the sign you actual see is not that sign class also.

False positives and false negatives, these values happen when your actual class is in conflict with with the predicted class.

False Positives (FP) – If the actual class is 'no' and predicted class is 'yes'. For example, if actual class says that the sign is not 20km/h speed limit but the predicted class is 20km/h speed limit

False Negatives (FN) – If the actual class is 'yes' but predicted class is 'no'. For example, if actual class value indicates the sign is speed limit 20km/h but the predicted class classified otherwise.

Accuracy - Accuracy is the the ratio between the correctly predicted observation to the total of observations. It is intuitive indicator but is helpful only for the symmetric datasets where values of false positive and false negatives are almost same. Therefore, we have to observe other parameters to define the performance of the model. For our model, we have got 0.97 which means our model is approx. 97% accurate.

Accuracy = TP+TN/TP+FP+FN+TN

Precision - Precision is the ratio between predicted positive observations which are correct to the total predicted positive observations. This number tells how many signs are actual belong to one specific class among those is classified into that class. We got 0.97 precision on average which is good.

Precision = TP/TP+FP

Recall - Recall is the ratio between predicted positive observations which are correct to all of the observations in the actual class. In our case, among the actual 20km/h limit sign, how many we have predicted correctly.

Recall = TP/TP+FN

F1 score - F1 Score can be interpred as harmonic average number of the precision and recall. This number compose of both false negatives and false positives. Although it is not as easy to sess as accuracy, but F1 is more helpful than accuracy in some case, especially when the class distribution is unbalance. Accuracy is meaningful if false positives and false negatives have similar cost. If the cost of false positives and false negatives are very different, it's time to consider both Precision and Recall. In our model, F1 score is 0.97.

F1 Score = 2*(Recall * Precision) / (Recall + Precision)

Below are our statistics of the scores in each class.

```
              precision    recall  f1-score   support

```

```
          0       1.00      0.98      0.99        61
          1       1.00      0.99      0.99       727
          2       1.00      0.97      0.98       774
          3       0.97      0.98      0.97       446
          4       0.99      0.99      0.99       659
          5       0.97      0.98      0.98       623
          6       0.81      0.99      0.89       123
          7       1.00      0.99      0.99       453
          8       0.98      0.96      0.97       459
          9       1.00      1.00      1.00       478
         10       0.98      0.99      0.99       653
         11       0.95      0.98      0.96       404
         12       0.96      0.99      0.98       667
         13       1.00      0.99      0.99       726
         14       1.00      0.99      0.99       273
         15       1.00      0.93      0.96       224
         16       0.99      0.99      0.99       150
         17       0.98      0.99      0.99       355
         18       0.93      0.94      0.94       387
         19       1.00      0.97      0.98        62
         20       1.00      0.86      0.92       105
         21       0.96      0.71      0.82       121
         22       0.88      0.99      0.93       107
         23       0.99      0.86      0.92       174
         24       0.89      0.99      0.94        81
         25       0.99      0.98      0.98       485
         26       0.99      0.94      0.96       191
         27       0.50      0.88      0.64        34
         28       0.97      0.94      0.95       155
         29       1.00      0.95      0.97        95
         30       0.74      0.87      0.80       128
         31       0.99      0.99      0.99       271
         32       1.00      0.91      0.95        66
         33       1.00      0.96      0.98       219
         34       0.99      0.98      0.99       121
         35       0.97      0.99      0.98       382
         36       0.93      1.00      0.97       112
         37       1.00      0.98      0.99        61
         38       0.97      0.99      0.98       672
         39       0.96      1.00      0.98        86
         40       0.97      0.90      0.93        97
         41       1.00      0.94      0.97        64
         42       1.00      0.91      0.95        99

avg / total       0.97      0.97      0.97     12630
```

For final result on test set, our architecture got 97.2% accuracy

5.3 Learning curve & training cost

Training cost and learning curve can be seen below. They decrease with high slop after twenty epoch. At the end of the training, they converge and stabilized to the point the accuracy cannot be improved after 50 epochs.

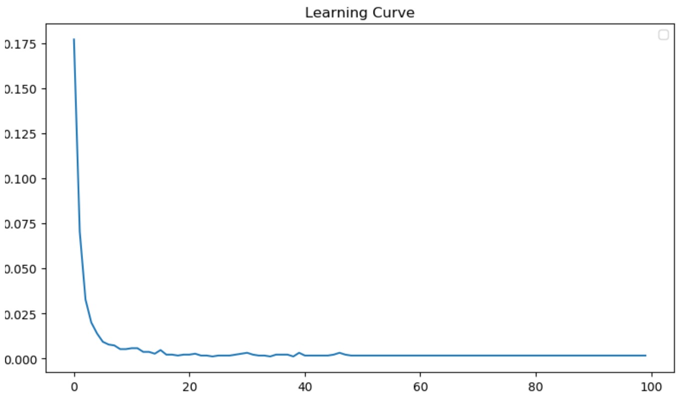

Figure 19: Training learning curve at 10e-4 rate

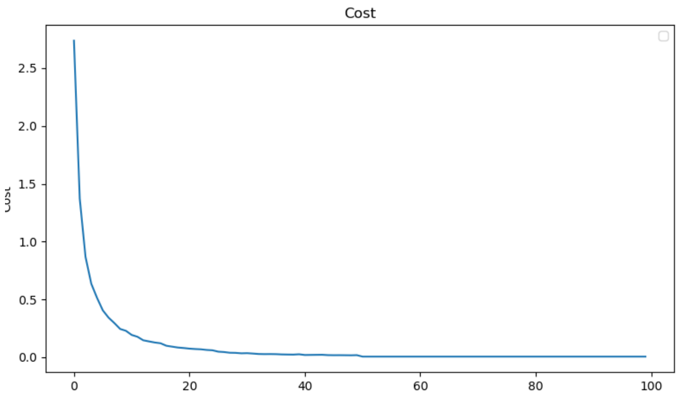

Figure 20: Training cost at 10e-4 rate

By tuning the learning rate to 0.0001, we have obtained the smoother and more stablilized learning curve but the total accuracy remains around 97.2%

Having tried different parameters, we saw that the accuracy was good starting from 0.0001 for learning curve. The rate at 0.001 obtained the same accuracy but only stablilized after 50 epochs. Between 20th and 50th one, the errors oscillate between 0.01 and 0.1. The higher rate above 0.001 have resulted in undesired inaccuracy.

5.4 Comparing with other models

TEAM	**METHOD**	TOTAL
DeepKnowledgeSeville[1]	CNN with 3 Spatial Transformers	**99.71%**
IDSIA[2]	Committee of CNNs	**99.46%**
COSFIRE[5]	Color-blob-based COSFIRE filters for object recogn	**99.87%**
INI-RTCV[9]	Human Performance	**99.84%**
Semanet[7]	Multi-Scale CNNs	**98.31%**
Our	Our Architecture	**97.23%**
CAOR[12]	Random Forests	**96.14%**
INI-RTCV [9]	LDA on HOG2	**95.68%**

Table 1: Our architecture result comparing of others using the same dataset

Comparing with other teams' techniques on the same problem, our model surpass the feature based methods, while less perform than other CNN. The reason is that we lacked the preprocessing steps like illumination. With more time for fine tuning the model architecture and with better preprocessing of images, we are sure that the accuracy can be increased as it is proven with other models.

5.5 Our takeaways:

The following are the key takeaways that we learned as a part of this project

- More the training samples, the better the result. Deep learning algorithms learn from data and if possible, always go for more data collection.
- The testing set should be a good representation of our desired result.(i.e) The training set and testing set should be similar in features and representation.
- Increasing the number of CONV layers has direct impact on accuracy during initial phases(i.e) We can start with fairly simple model at the beginning and keep adding more layers to make the model more complex till we reach an accuracy saturation.
- Augmentation of images is nice to do when we donot have enough data. Augmentation can help push the accuracy to certain extent, but excess augmentation might affect the model generality and hence the accuracy.

- Keeping less number of fully connected layers can drastically improve the speed of the training as it adds to a lot of parameters to train. Also adding excess fully connected layers can cause no learning at all as the data in the end layers will be very sparse. We need to use techniques like drop out to overcome the effect.
- A good model evaluation and visualization pipeline is crucial in understanding and debugging deep learning problems.

6 Conclusion & Futher Development

We covered how convolutional neural network can be used to classify traffic signs with high accuracy, employing a variety of pre-processing and regularization techniques, comparing with featured based models. The code is configurable to be used in computer vision lab for deep learning. Our model reached over 97% accuracy on the test set, achieving 99.7% on the validation set.

We have gained practical experience using Google's machine learning framework Tensorflow and also with other well known plugins such as matplotlib, scikit-learn, OpenCV, etc. with Python as programming language, which are extensively used by deep learning community. We also learned a lot about Convolutional Neural Network architecture and its working. We also learnt proper methods for evaluating deep learning models which are curcial in deep learning pipeline.

In the future, the accuracy can be improved by trying out new architectures, regularization and carefully engineered preprocessing pipeline.

For the purpose of labs, we also decided to make a Jupyter Notebook for the exercise as an additional task. Jupyter Notebook lets us create easy to use interface that can be used to create and share documents that contain live code, equations, visualizations and narrative text. The notebook is in functional state but is not complete. This notebook can be furthur developed and it can serve as an excellent material for the labs. Also the dependant software has to be installed in the computers in the lab.

References

[1] Álvaro Arcos-García, Juan A. Álvarez-García, and Luis M. Soria-Morillo. "Deep neural network for traffic sign recognition systems: An analysis of spatial transformers and stochastic optimisation methods". In: *Neural Networks* 99 (2018), pp. 158–165. ISSN: 0893-6080. DOI: `https://doi.org/10.1016/j.neunet.2018.01.005`. URL: `http://www.sciencedirect.com/science/article/pii/S0893608018300054`.

[2] Dan Cireşan et al. "Multi-column deep neural network for traffic sign classification". In: *Neural Networks* 32 (2012). Selected Papers from IJCNN 2011, pp. 333–338. ISSN: 0893-6080. DOI: `https://doi.org/10.1016/j.neunet.2012.02.023`. URL: `http://www.sciencedirect.com/science/article/pii/S0893608012000524`.

[3] Weiguang (Gavin) Ding. *The figure is generated by adapting the code from.* URL: `https://github.com/gwding/draw_convnet`.

[4] Eric. *Plotting confusion matrix.* URL: `https://stackoverflow.com/questions/35572000/how-can-i-plot-a-confusion-matrix`.

[5] Baris Gecer, George Azzopardi, and Nicolai Petkov. "Color-blob-based COSFIRE filters for object recognition". In: *Image and Vision Computing* 57 (2017), pp. 165–174. ISSN: 0262-8856. DOI: `https://doi.org/10.1016/j.imavis.2016.10.006`. URL: `http://www.sciencedirect.com/science/article/pii/S0262885616301895`.

[6] Yann Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. 1998, pp. 2278–2324.

[7] P. Sermanet and Y. LeCun. "Traffic sign recognition with multi-scale Convolutional Networks". In: *The 2011 International Joint Conference on Neural Networks*. July 2011, pp. 2809–2813. DOI: `10.1109/IJCNN.2011.6033589`.

[8] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014).

[9] J. Stallkamp et al. "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition". In: *Neural Networks* 0 (2012), pp. -. ISSN: 0893-6080. DOI: `10.1016/j.neunet.2012.02.016`. URL: `http://www.sciencedirect.com/science/article/pii/S0893608012000457`.

[10] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* abs/1512.00567 (2015). arXiv: `1512.00567`. URL: `http://arxiv.org/abs/1512.00567`.

[11] Vivek Yadav. *Image Augmentation.* URL: `https://github.com/vxy10/ImageAugmentation`.

[12] F. Zaklouta, B. Stanciulescu, and O. Hamdoun. "Traffic sign classification using K-d trees and Random Forests". In: *The 2011 International Joint Con-*

ference on Neural Networks. July 2011, pp. 2151–2155. DOI: 10.1109/IJCNN.2011.6033494.

7 Appendix

7.1 Project Codes:

```python
## Imports #################################################
import numpy as np # scientific computations library (http://www.numpy.org/)
import tensorflow as tf # deep learning library (https://www.tensorflow.org/)
import cv2 # OpenCV computer vision library (https://opencv.org/)
import random
import os # file operations
import math
import csv

# Helper function files - plot_utils.py and img_utils.py
from plot_utils import visualize_dataset, plot_confusion_matrix, visualize_training_data_distribution
from img_utils import get_train_images, get_test_images, preprocess_images, transform_image

import matplotlib.pyplot as plt
###############################################################

## Global Variables ##############################################
NUM_CLASSES = 43

curr_dirname = os.path.dirname(os.path.abspath(__file__))
project_root_dir = os.path.dirname(os.path.abspath(curr_dirname))
MODEL_EXPORT_DIR = os.path.join(project_root_dir, 'models/new')
################################################################

class TrafficSignsClassifier:
    def __init__(self):
        self.x_train = None
        self.y_train = None
        self.x_validation = None
        self.y_validation = None
        self.x_test = None
        self.y_test = None

    def train_validation_test_split(self, train_images, train_labels, test_images, test_labels, split_size = 5):
        train_labels = np.array(train_labels, dtype=np.int8)
        train_labels = self.convert_to_one_hot(train_labels, NUM_CLASSES)
        test_labels = np.array(test_labels, dtype=np.int8)
        test_labels = self.convert_to_one_hot(test_labels, NUM_CLASSES)
```

```python

        train_dataset_size = len(train_images)
        # split the train set to get validation set
        num_validation_images = int(train_dataset_size * split_size/100)
        is_for_training = np.ones(train_dataset_size, dtype=bool)
        # randomly choose validation set indexes
        validation_imgs_idx = np.random.choice(np.arange(train_dataset_size), num_validation_images, replace=False)
        is_for_training[validation_imgs_idx] = False

        self.x_train = train_images[is_for_training]
        self.y_train = train_labels[is_for_training]
        self.x_validation = train_images[~is_for_training]
        self.y_validation = train_labels[~is_for_training]
        self.x_test, self.y_test = test_images, test_labels

    def create_placeholders(self, nH, nW, nC, nY):

        X = tf.placeholder(tf.float32, shape=[None, nH, nW, nC], name="X")
        Y = tf.placeholder(tf.float32, shape=[None, nY], name="Y")
        keep_prob = tf.placeholder(tf.float32)

        return X, Y, keep_prob

    def initialize_parameters(self):
      # Weights initialization
        W1 = tf.get_variable("W1", shape = [5, 5, 3, 16], initializer = tf.contrib.layers.xavier_initializer(seed = 0))
        W2 = tf.get_variable("W2", shape = [5, 5, 16, 32], initializer = tf.contrib.layers.xavier_initializer(seed = 0))
        W3 = tf.get_variable("W3", shape = [3, 3, 32, 128], initializer = tf.contrib.layers.xavier_initializer(seed = 0))
        W4 = tf.get_variable("W4", shape = [3, 3, 128, 256], initializer = tf.contrib.layers.xavier_initializer(seed = 0))

        return { "W1": W1, "W2":W2, "W3": W3, "W4": W4 }

    def forward_propagation(self, X, parameters, keep_prob):
        '''
      MODEL ARCHITECTURE:
      CONV --> ReLU --> CONV --> ReLU --> POOL --> CONV --> ReLU --> CONV --> ReLU --> POOL --> FC_1 --> FC_2 --> Output :)
        '''
        W1 = parameters["W1"]
        W2 = parameters["W2"]
        W3 = parameters["W3"]
        W4 = parameters["W4"]
```

```

        # Conv1 layer with stride 1 and same padding
        Z1 = tf.nn.conv2d(X, W1, strides=[1,1,1,1], padding="VALID")

        # Relu
        A1 = tf.nn.relu(Z1)

        # Conv2 with stride 1 and same padding
        Z2 = tf.nn.conv2d(A1, W2, strides=[1,1,1,1], padding="VALID")

        # Relu
        A2 = tf.nn.relu(Z2)

        # max-pool Kernel[2X2] stride 2
        P1 = tf.nn.max_pool(A2, ksize=[1,2,2,1], strides = [1,2,2,1], padding="VALID")

        # Conv3 layer with stride 1 and same padding
        Z3 = tf.nn.conv2d(P1, W3, strides=[1,1,1,1], padding="VALID")

        # Relu
        A3 = tf.nn.relu(Z3)

        # Conv4 with stride 1 and same padding
        Z4= tf.nn.conv2d(A3, W4, strides=[1,1,1,1], padding="VALID")

        # Relu
        A4 = tf.nn.relu(Z4)

        # max-pool kernel[2X2] stride 2
        P2 = tf.nn.max_pool(A4, ksize=[1,2,2,1], strides = [1,2,2,1], padding="VALID")

        # Flatten
        P2 = tf.contrib.layers.flatten(P2)

        #fully connected
        FC_1 = tf.contrib.layers.fully_connected(P2, 256, activation_fn=None)

        # drop outs are used to randomly deactivate neurons in layers.
        drop_out_1 = tf.nn.dropout(FC_1, keep_prob)

        FC_2 = tf.contrib.layers.fully_connected(drop_out_1, 128, activation_fn=None)

        #drop_out_2 = tf.nn.dropout(FC_2, keep_prob)

        #FC_3 = tf.contrib.layers.fully_connected(drop_out_2, 80,
```

```
activation_fn=None)

        Z5 = tf.contrib.layers.fully_connected(FC_2, NUM_CLASSES, activation_fn=None)

        # returning Activations since we use that for visualization.
        return [A1, A2, A3, A4, Z5]

    def compute_cost(self, output, Y):
        cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits = output, labels = Y))
        return cost


    def random_mini_batches(self, X, Y, mini_batch_size = 64, seed = 0):
        # This code was adapted from deeplearning.ai online course.
        # Randomly split data into minibatches
        # Returns list of all minibatch

        m = X.shape[0] # number of training examples
        mini_batches = []
        np.random.seed(seed)
        permutation = list(np.random.permutation(m))
        shuffled_X = X[permutation,:,:,:]
        shuffled_Y = Y[permutation,:]

        num_complete_minibatches = int(math.floor(m/mini_batch_size)) # number of mini batches of size mini_batch_size in your partitionning

        for k in range(0, num_complete_minibatches):
            mini_batch_X = shuffled_X[k * mini_batch_size : k * mini_batch_size + mini_batch_size,:,:,:]
            mini_batch_Y = shuffled_Y[k * mini_batch_size : k * mini_batch_size + mini_batch_size,:]
            mini_batch = (mini_batch_X, mini_batch_Y)
            mini_batches.append(mini_batch)

        # Pick the remaining images. Since we only have collected complete minibatches.
        # This batch will not be complete.
        remaining_num_imgs = m - num_complete_minibatches*mini_batch_size
        missed_batch = (shuffled_X[m-remaining_num_imgs:m], shuffled_Y[m-remaining_num_imgs:m])
        mini_batches.append(missed_batch)

        return mini_batches
```

```python

    def convert_to_one_hot(self, Y, C):
        # Details about one hot: https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/
        Y = np.eye(C)[Y.reshape(-1)]
        return Y

    def get_augmented_images(self, images, labels, epoch):
        # Image augmentation. The number of augmented image is is proportional to 50/epoch+1
        augmented_images = []
        augmented_labels = []
        len_img = len(images)
        num_imgs = int(50/(epoch+1))
        for i in range(num_imgs):
            rand_int = np.random.randint(len_img)
            augmented_images.append(transform_image(images[rand_int],3,3,3))
            augmented_labels.append(labels[rand_int])

        return np.array(augmented_images), np.array(augmented_labels)

    def visualize_filters(self, A1, A2, A3, A4, image, sess, X, keep_prob):
        # This function is purely for visulization purpose. We visualize different activation layers
        activation_units_1 = A1.eval(session=sess,feed_dict={X:image.reshape(1,30,30,3),keep_prob:1.0})
        activation_units_2 = A2.eval(session=sess,feed_dict={A1:activation_units_1,keep_prob:1.0})
        activation_units_3 = A3.eval(session=sess,feed_dict={A2:activation_units_2,keep_prob:1.0})
        activation_units_4 = A4.eval(session=sess,feed_dict={A3:activation_units_3,keep_prob:1.0})

        activations = [activation_units_1, activation_units_2, activation_units_3, activation_units_4]

        for activation in activations:
            num_of_filters = activation.shape[3]
            filtered_images = []
            for i in range(num_of_filters):
                filtered_images.append(activation[0,:,:,i-1])

            visualize_dataset(np.array(filtered_images)[0:15], True, (6,6), 4, 4)

    def build_model(self, restore = True, learning_rate = 0.001, num_epochs = 30, minibatch_size = 100, print_cost = True):
```

```
        costs = []
        accuracys = []

        # Seeding is done so we get same randomness during different runs. Useful during testing
        tf.set_random_seed(1)

        (m, nH, nW, nC) = self.x_train.shape
        nY = self.y_train.shape[1]

        X, Y, keep_prob = self.create_placeholders(nH, nW, nC, nY)

        parameters = self.initialize_parameters()

        A1, A2, A3, A4, Z5 = self.forward_propagation(X, parameters, keep_prob)

        cost = self.compute_cost(Z5, Y)

        optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

        prediction = tf.argmax(tf.nn.softmax(Z5), 1)

        truth = tf.argmax(Y, 1)

        equality = tf.equal(prediction, truth)

        accuracy = tf.reduce_mean(tf.cast(equality, tf.float32))

        init = tf.global_variables_initializer()

        saver = tf.train.Saver()

        with tf.Session() as sess:
            sess.run(init)

            if restore == True: # restore the previously trained model

                saver.restore(sess,tf.train.latest_checkpoint(MODEL_EXPORT_DIR))
                pred, tru, eq, acc, shuffled_Y = self.run_test_in_batches(sess, [prediction, truth, equality, accuracy], X, Y, keep_prob, 1000)
                print('Final Test Accuracy: {} %'.format(acc*100))

                # Visualization utility functions
                self.print_confusion_matrix(shuffled_Y,pred)
                self.plot_failed_cases(eq, pred)
```

```
            self.visualize_filters(A1, A2, A3, A4, self.x_train[20], sess, X, keep_prob)

        else: # start training process

            # epoch is one run through the entire training set.
            # Since our batch is large, we split them into mini batches and run batch gradient descent on them.
            for epoch in range(num_epochs):
                minibatch_cost = 0.
                num_minibatches = int(m / minibatch_size) # number of minibatches
                minibatches = self.random_mini_batches(self.x_train, self.y_train, minibatch_size, seed=3)

                for minibatch in minibatches:
                    (minibatch_X, minibatch_Y) = minibatch
                    # augmenting images during training
                    aug_images, aug_labels = self.get_augmented_images(minibatch_X, minibatch_Y, epoch)

                    if len(aug_images):
                        minibatch_X = np.append(minibatch_X, aug_images, axis = 0)
                        minibatch_Y = np.append(minibatch_Y, aug_labels, axis = 0)
                    _ , temp_cost = sess.run([optimizer, cost], feed_dict = {X: minibatch_X, Y: minibatch_Y, keep_prob: 0.5})
                    minibatch_cost += temp_cost / num_minibatches

                if print_cost == True:
                    train_acc= sess.run(accuracy, feed_dict = {X: self.x_validation, Y: self.y_validation, keep_prob: 1.0})
                    print('Validation Data Accuracy: {} %'.format(train_acc*100))
                    costs.append(minibatch_cost)
                    accuracys.append(train_acc)

                if print_cost == True and epoch % 5 == 0:
                    self.save_model(sess, epoch)
                    print ("############ EPOCH %i SUMMARY: ############" % epoch)
                    print("Copy of model saved...")
                    print ("Cost after epoch %i: %f" % (epoch, minibatch_cost))
                    print('Validation Data Accuracy: {} %'.format(train_acc*100))
                    print ('#############################################')
```

```

                pred, tru, eq, acc, shuffled_Y = self.run_test_in_batches(sess, [prediction, truth, equality, accuracy], X, Y, keep_prob, 1000)

                print('Final Test Accuracy: {} %'.format(acc*100))

                # Visualization utility functions
                self.print_confusion_matrix(shuffled_Y,pre)
                self.plot_failed_cases(eq, pred)
                self.visualize_filters(A1, A2, A3, A4, self.x_train[20], sess, X, keep_prob)

    def run_test_in_batches(self, sess, information, X, Y, keep_prob, size=1000):
        # Splitting the test data into batches and running to model on the it
        # to find the overall accuracy. Since our test batch contains about 12630 images,
        # the computer memory is not sufficient in a single pass so splitting them into
        # mini batches.

        test_minibatches = self.random_mini_batches(self.x_test, self.y_test, 1000)
        total_accuracy = 0
        predictions = np.array([])
        truth = np.array([])
        equality = np.array([])
        shuffled_Y = []

        for test_minibatch in test_minibatches:
            test_minibatch_x, test_minibatch_y = test_minibatch
            pred, tru, eq, acc= sess.run(information, feed_dict = {X: test_minibatch_x, Y: test_minibatch_y, keep_prob: 1.0})
            total_accuracy += acc
            predictions = np.concatenate((predictions, pred), axis=0)
            truth = np.concatenate((truth, tru), axis=0)
            equality = np.concatenate((equality, eq), axis=0)
            test_minibatch_y = test_minibatch_y.tolist()
            shuffled_Y += test_minibatch_y

        total_accuracy = (total_accuracy)/len(test_minibatches)

        return predictions, truth, equality, total_accuracy, np.array(shuffled_Y)


    def print_confusion_matrix(self, label, prediction):
```

```python
        label = np.argmax(label, 1)
        sess = tf.Session()
        cnfn_matrix = sess.run(tf.confusion_matrix(label, prediction))
        np.set_printoptions(precision=2)

        # Plot non-normalized confusion matrix
        plot_confusion_matrix(cnfn_matrix, classes=label,
                              title='Confusion matrix, without
    normalization')

    def save_model(self, sess, epoch):
        saver = tf.train.Saver()
        saver.save(sess, MODEL_EXPORT_DIR + '/my-model', global_step =
    epoch)

    def plot_failed_cases(self, equality, prediction):
        incorrect = (equality == False)
        test_imgs = self.x_test
        test_lbls = self.y_test
        incorrect_images = test_imgs[incorrect]
        incorrect_predictions = prediction[incorrect]
        correct_labels = np.argmax(test_lbls[incorrect], 1)

        visualize_dataset(incorrect_images[25:50], False, (8,8), 5, 5,
    correct_labels, incorrect_predictions)


if __name__ == "__main__":

    # Get the image paths
    train_img_path = os.path.join(project_root_dir, 'GTSRB/
    Final_Training/Images')
    test_img_path = os.path.join(project_root_dir, 'GTSRB/Final_Test/
    Images')

    # Get the images and store them
    train_images, train_labels = get_train_images(train_img_path)
    test_images, test_labels = get_test_images(test_img_path)

    visualize_training_data_distribution(train_labels)

    # Preprocess images
    preprocessed_train_images = preprocess_images(train_images, False)
    preprocessed_test_images = preprocess_images(test_images, False)

    # Model building and evaluation
    traffic_sign_classifier = TrafficSignsClassifier()
    traffic_sign_classifier.train_validation_test_split(np.array(
    preprocessed_train_images), train_labels, np.array(
```

```
    preprocessed_test_images), test_labels)
    traffic_sign_classifier.build_model(restore = True)
```

Listing 1: Traffic_signs_classifier.py

```
import matplotlib.pyplot as plt # Plotting library (https://matplotlib.
    org/)
import numpy as np

def visualize_dataset(images, to_gray = True, fsize=(8,8), rows = 5,
    cols = 5, labels = [], predictions = []):

  fig = plt.figure(figsize=fsize)
  num_imgs = images.shape[0]

  for i in range(0, num_imgs):
    ax = fig.add_subplot(rows, cols, i + 1)
    if len(predictions) and len(labels):
      ax.set_title("Prediction: " + str(predictions[i]) + "\nTrue Label
    : " + str(labels[i]))

    # Matplot lib plots gray scale image only if dimension is like (x,y
    ) and not (x,y,1)
    if to_gray:
      image = images[i].reshape(images[i].shape[0], images[i].shape[1])
      plt.imshow(image,  cmap='gray')
    else:
      plt.imshow(images[i])

  plt.tight_layout()
  plt.show()

def visualize_training_data_distribution(training_labels):
    unique, counts = np.unique(training_labels, return_counts=True)
    unique = unique.astype(int)

    fig = plt.figure()
    fig.suptitle('No. of images per class', fontsize=20)
    plt.bar(unique, counts)
    plt.xticks(np.arange(min(unique), max(unique)+1, 1.0)) #to set the
    x axis tick freq to 1
    plt.show()

def plot_confusion_matrix(cm, classes, title='Confusion matrix', cmap=
    plt.cm.jet):

    #This function prints and plots the confusion matrix.
    print('Confusion matrix, without normalization')
    print(cm)
```

```python
    fig = plt.figure()
    plt.clf()

    res = plt.imshow(cm, cmap=plt.cm.jet,
                interpolation='nearest')

    width, height = cm.shape

    for x in range(width):
      for y in range(height):
          plt.annotate(str(cm[x][y]), xy=(y, x),
                      horizontalalignment='center',
                      verticalalignment='center',
                      size=6
                      )

    cb = plt.colorbar(res)
    plt.title(title)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

Listing 2: Plot_utils.py

```python
import numpy as np
import cv2 # OpenCV computer vision library (https://opencv.org/)
import os # file operations
import math
import csv
import matplotlib.pyplot as plt # Plotting library (https://matplotlib.org/)

def get_train_images(img_path):

  images = []
  labels = []

  for c in range(0,43):
    prefix = img_path + '/' + format(c, '05d') + '/' # subdirectory for class
    gtFile = open(prefix + 'GT-'+ format(c, '05d') + '.csv') # annotations file
    gtReader = csv.reader(gtFile, delimiter=';') # csv parser for annotations file
    next(gtReader) # skip header

    for row in gtReader:
      images.append(plt.imread(prefix + row[0])) # the 1th column is the filename
      labels.append(row[7]) # the 8th column is the label
```

```
    gtFile.close()

  return images, labels

def get_test_images(img_path):

  images = []
  labels = []

  gtFile = open(img_path + '/GT-final_test.csv') # annotations file
  gtReader = csv.reader(gtFile, delimiter=';') # csv parser for annotations file
  next(gtReader) # skip header

  for row in gtReader:
    images.append(plt.imread(img_path + '/' + row[0])) # the 1th column is the filename
    labels.append(row[7]) # the 8th column is the label
  gtFile.close()

  return images, labels


def preprocess_images(images, to_gray = False, size=(30,30)):
  processed_imgs = []
  for image in images:
    image = cv2.resize(image, (size[0], size[1]), interpolation=cv2.INTER_LINEAR)
    if to_gray:
      image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
      image = cv2.equalizeHist(image)

    norm_image = cv2.normalize(image, image, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_32F)

    processed_imgs.append(norm_image)

  return processed_imgs

def transform_image(img,max_rotation,max_shear,max_translation):
    '''
    Helper functions for augmenting images.
    https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_geometric_transformations/py_geometric_transformations.html
    Uses cv2.warpAffine() function
    Outputs:
    Random rotated image in range (-max_rotation to max_rotation)
    Random translated image in range (-max_translation to max_translation)
```

```
    Random sheared image with max_shear
    returns the image after applying all three operations.
    '''
    # Rotation
    ang_rotation = np.random.uniform(-max_rotation, max_rotation)
    rows,cols,ch = img.shape
    Rot_M = cv2.getRotationMatrix2D((cols/2,rows/2),ang_rotation,1)

    # Translation
    tr_x = np.random.uniform(-max_translation, max_translation)
    tr_y = np.random.uniform(-max_translation, max_translation)
    Trans_M = np.float32([[1,0,tr_x],[0,1,tr_y]])

    # Shear
    pts1 = np.float32([[5,5],[20,5],[5,20]])

    pt1 = 5+np.random.uniform(-max_shear, max_shear)
    pt2 = 20+np.random.uniform(-max_shear, max_shear)

    pts2 = np.float32([[pt1,5],[pt2,pt1],[5,pt2]])

    shear_M = cv2.getAffineTransform(pts1,pts2)

    img = cv2.warpAffine(img,Rot_M,(cols,rows))
    img = cv2.warpAffine(img,Trans_M,(cols,rows))
    img = cv2.warpAffine(img,shear_M,(cols,rows))

    return img
```

Listing 3: img_utils.py