A MINOR PROJECT REPORT ON

**“Traffic Sign Recognition”**

**Submitted**

*In the partial fulfilment of the requirementsfor*

*the award of the degree of*

**BACHELOR OF TECHNOLOGY**

In

**COMPUTER SCIENCE & ENGINEERING**

By

B.Venkat (171FA04007)

HariKrishna (171FA04195)

Hussain (171FA04146)

Sairam(171FA04556)

Under the esteemed guidance of

**Dr. M. Hemanth Kumar**, professor



**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**VIGNAN'S FOUNDATION FOR SCIENCE, TECHNOLOGY AND RESEARCH Deemed to be University**

(**Accredited by NAAC “A” grade**)

**Vadlamudi, Guntur.**

**Abstract:**

Self-driving cars in which the passenger can fully depend on the car for traveling. But to achieve level 5 autonomous, it is necessary for vehicles to understand and follow all traffic rules.

In the world of Artificial Intelligence and advancement in technologies, many researchers and big companies like Tesla, Uber, Google, Mercedes-Benz, Toyota, Ford, Audi, etc are working on autonomous vehicles and self-driving cars. So, for achieving accuracy in this technology, the vehicles should be able to interpret traffic signs and make decisions accordingly.

**Problem Description:**

There are several different types of traffic signs like speed limits, no entry, traffic signals, turn left or right, children crossing, no passing of heavy vehicles, etc. Traffic signs classification is the process of identifying which class a traffic sign belongs to.

Traffic signs are an integral part of our road infrastructure. They provide critical information, sometimes compelling recommendations, for road users, which in turn requires them to adjust their driving behaviour to make sure they adhere with whatever road regulation currently enforced. Without such useful signs, we would most likely be faced with more accidents, as drivers would not be given critical feedback on how fast they could safely go, or informed about road works, sharp turn, or school crossings ahead. In our modern age, around 1.3M people die on roads each year. This number would be much higher without our road signs.  
Naturally, autonomous vehicles must also abide by road legislation and therefore recognize and understand traffic signs.

Traditionally, standard computer vision methods were employed to detect and classify traffic signs, but these required considerable and time-consuming manual work to handcraft important features in images. Instead, by applying deep learning to this problem

A deep neural network model that can classify traffic signs present in the image into different categories. With this model, we are able to read and understand traffic signs which are a very important task for all autonomous vehicles.

# About the project

Every country has some standards set for the design of different traffic signs like U-turn, Left-turn, Right-turn, No-entry, etc. Traffic sign recognition is the process of automatically identifying which of the following class the sign belongs to. The earlier Computer Vision techniques required lots of hard work in data processing and it took a lot of time to manually extract the features of the image. Now, deep learning techniques have come to the rescue and today we will see how to build a traffic recognition system for autonomous vehicles.



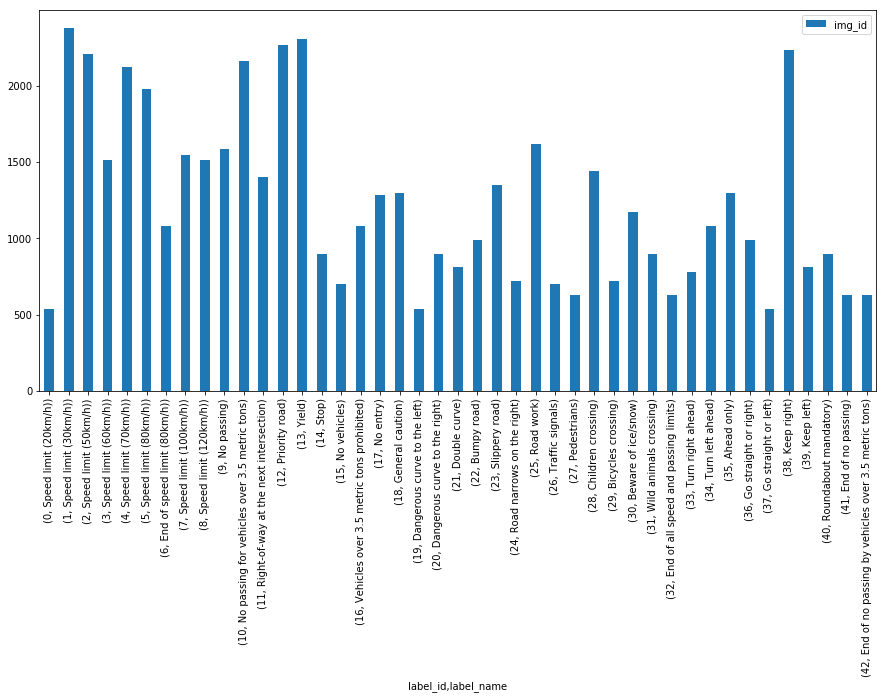
Traffic sign classification is the process of automatically recognizing traffic signs along the road, including speed limit signs, yield signs, merge signs, etc. Being able to automatically recognize traffic signs enables us to build “smarter cars”.

Self-driving cars need traffic sign recognition in order to properly parse and understand the roadway. Similarly, “driver alert” systems inside cars need to understand the roadway around them to help aid and protect drivers.

# DATASET

The dataset we have used for this project is the GTSRB (German traffic sign recognition benchmark). It contains a Train folder that has traffic sign images in 43 different classes, a Test folder that has over 12,000 images for testing purposes. A test.csv file that contains the path of the test images along with their respective classes.

* Images are 32 (width) x 32 (height) x 3 (RGB color channels)
* Training set is composed of 34799 images
* Validation set is composed of 4410 images
* Test set is composed of 12630 images
* There are 43 classes (e.g. Speed Limit 20km/h, No entry, Bumpy road, etc.)



There are a number of challenges in the GTSRB dataset, **the first being that images are low resolution**, and worse, **have poor contrast** (as seen in **Figure 2** above). These images are pixelated, and in some cases, it’s extremely challenging, if not impossible, for the human eye and brain to recognize the sign.

The second challenge with the dataset is **handling class skew:**

The top class *(Speed limit 50km/h)* has over 2,000 examples while the least represented class *(Speed limit 20km/h)* has under 200 examples — that’s an order of magnitude difference!

**In order to successfully train an accurate traffic sign classifier we’ll need to devise an experiment that can:**

* Preprocess our input images to improve contrast.
* Account for class label skew.

# PREREQUISITES

To implement this project we will be using Keras which is a popular deep learning framework for python  and some additional library scikit-learn, numpy, PIL, pandas, tkinter, and jupyterlab

* **OpenCV**
* **NumPy**
* **scikit-learn**
* **scikit-image**
* **imutils**
* **matplotlib**
* **TensorFlow 2.0** (CPU or GPU)
* **Python 3.6**
* Keras 2.3.1

STEPS TO BUILD THE PROJECT

## 1. Setup the Project

Download the dataset into the Traffic sign recognition project folder. We will use the Jupyter notebook which is an interactive development environment.

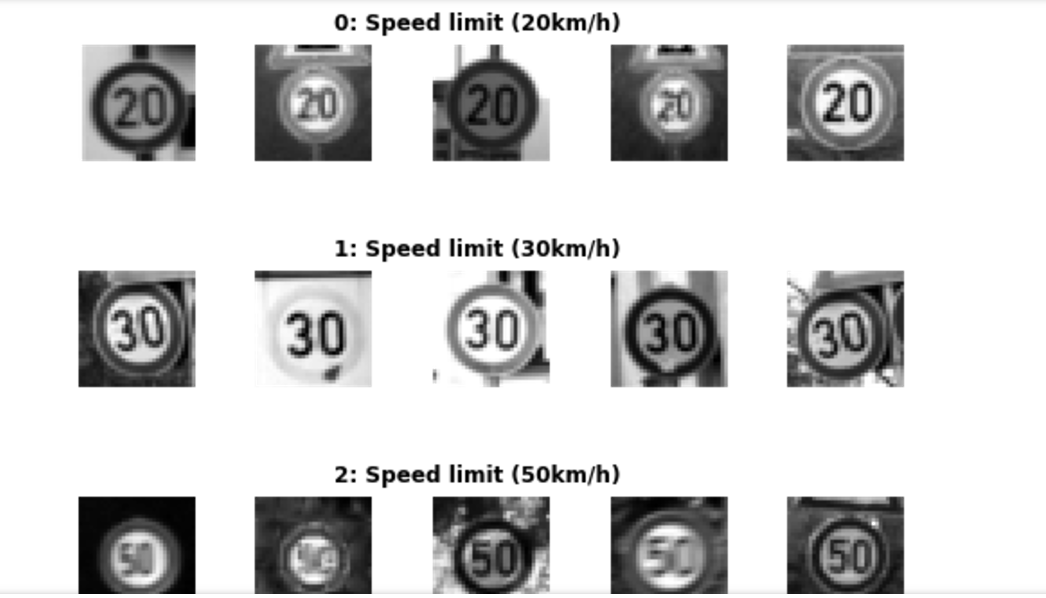
Open up the terminal and traverse to the project folder. To start the jupyter notebook you have to type “jupyter lab” in the terminal and it will open up a web-based interface.

Then you can start the jupyter notebook and rename the file as traffic\_sign\_recognition.

# Pre-Processing Steps

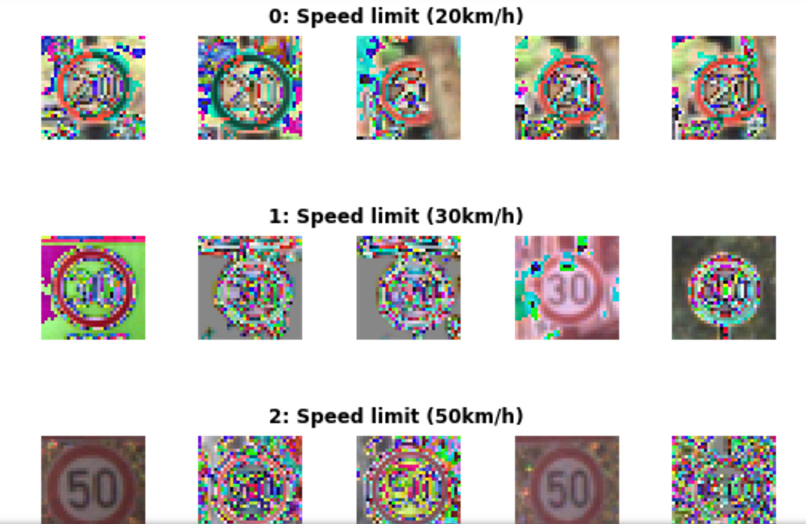
We initially apply two pre-processing steps to our images:

**Grayscale**We convert our 3 channel image to a single grayscale image



Sample Of Grayscale Training Set Images, with labels above

**Image Normalisation**We center the distribution of the image dataset by subtracting each image by the dataset mean and divide by its standard deviation. This helps our model treating images uniformly. The resulting images look as follows:



Normalised images — we can see how “noise” is distributed

# Dropout

In order to improve the model reliability, we turned to dropout, which is a form of regularisation where weights are kept with a probability p: the unkept weights are thus “dropped”. This prevents the model from overfitting. Dropout was introduced by Geoffrey Hinton, a pioneer in the deep learning space. His group’s [paper](http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf) on this topic is a must read to better understand the motivations behind the authors. There’s also a fascinating parallel with biology and evolution.  
In the paper, the authors apply varying degrees of dropout, depending on the type of layer. I therefore decided to adopt a similar approach, defining two levels of dropout, one for convolutional layers, the other for fully connected layers:

p-conv: probability of keeping weight in convolutional layer  
p-fc: probability of keeping weight in fully connected layer

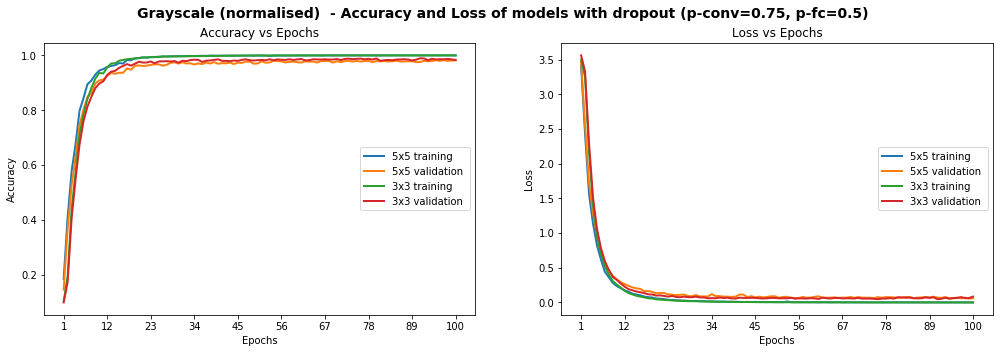
Moreover, the authors gradually adopted more aggressive (i.e. lower) values of dropout as they go deeper in the network. Therefore I also decided:

p-conv >= p-fc

that is, we will keep weights with a greater than or equal probability in the convolutional than fully connected layers. The way to reason about this is that we treat the network as a funnel and therefore want to gradually tighten it as we move deeper into the layers: we don’t want to discard too much information at the start as some of it would be extremely valuable. Besides, as we apply [MaxPooling](https://www.quora.com/What-is-max-pooling-in-convolutional-neural-networks" \t "_blank) in the convolutional layers, we are already losing a bit of information.

We tried different paratemers but ultimately settled on p-conv=0.75 and p-fc=0.5, which enabled us to achieve a test set accuracy of 97.55% on normalised grayscale images with the 3x3 model. Interestingly, we achieved over 98.3% accuracy on the validation set:

Training EdLeNet\_Norm\_Grayscale\_3x3\_Dropout\_0.50 [epochs=100, batch\_size=512]...  
  
[1] total=5.222s | train: time=3.139s, loss=3.4993, acc=0.1047 | val: time=2.083s, loss=3.5613, acc=0.1007  
[10] total=5.190s | train: time=3.122s, loss=0.2589, acc=0.9360 | val: time=2.067s, loss=0.3260, acc=0.8973  
...  
[90] total=5.193s | train: time=3.120s, loss=0.0006, acc=0.9999 | val: time=2.074s, loss=0.0747, acc=0.9841  
[100] total=5.191s | train: time=3.123s, loss=0.0004, acc=1.0000 | val: time=2.068s, loss=0.0849, acc=0.9832  
Model ./models/EdLeNet\_Norm\_Grayscale\_3x3\_Dropout\_0.50.chkpt saved  
[EdLeNet\_Norm\_Grayscale\_3x3\_Dropout\_0.50 - Test Set] time=0.686s, loss=0.1119, acc=0.9755



Models Performance on Grayscale Normalised Images, After The Introduction Of Dropout

The graphs above show that the model is smooth, unlike some of the graphs higher up. We have already achieved the objective of scoring over 93% accuracy on the test set, but can we do better? Remember that some of the images were blurry and the distribution of images per class was very uneven. We explore below additional techniques we used to tackle each point.

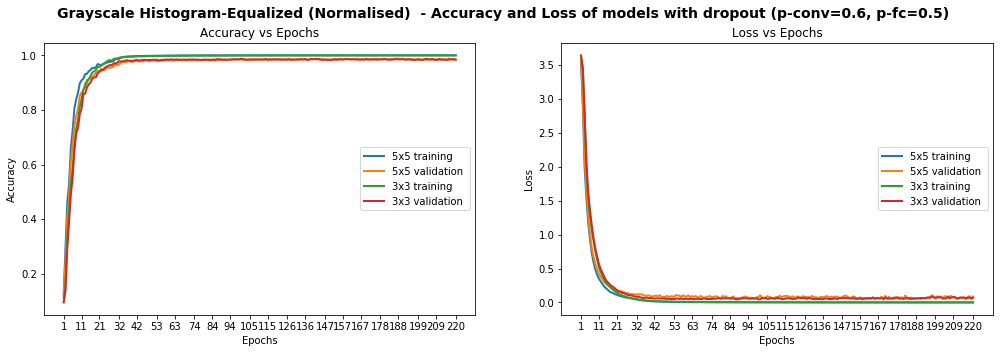
# Histogram Equalization

Histogram Equalization is a computer vision technique used to increase the contrast in images. As some of our images suffer from low contrast (blurry, dark), we will improve visibility by applying OpenCV’s Contrast Limiting Adaptive Histogram Equalization (aka CLAHE) function.

We once again try various configurations, and find the best results, with **test accuracy of 97.75%**, on the 3x3 model using the following dropout values: p-conv=0.6, p-fc=0.5 .

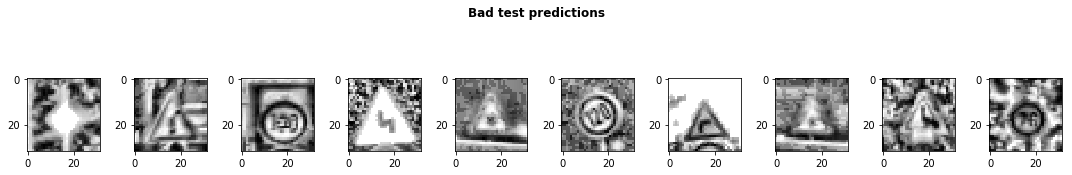
Training EdLeNet\_Grayscale\_CLAHE\_Norm\_Take-2\_3x3\_Dropout\_0.50 [epochs=500, batch\_size=512]...[1] total=5.194s | train: time=3.137s, loss=3.6254, acc=0.0662 | val: time=2.058s, loss=3.6405, acc=0.0655  
[10] total=5.155s | train: time=3.115s, loss=0.8645, acc=0.7121 | val: time=2.040s, loss=0.9159, acc=0.6819  
...  
[480] total=5.149s | train: time=3.106s, loss=0.0009, acc=0.9998 | val: time=2.042s, loss=0.0355, acc=0.9884  
[490] total=5.148s | train: time=3.106s, loss=0.0007, acc=0.9998 | val: time=2.042s, loss=0.0390, acc=0.9884  
[500] total=5.148s | train: time=3.104s, loss=0.0006, acc=0.9999 | val: time=2.044s, loss=0.0420, acc=0.9862  
Model ./models/EdLeNet\_Grayscale\_CLAHE\_Norm\_Take-2\_3x3\_Dropout\_0.50.chkpt saved  
[EdLeNet\_Grayscale\_CLAHE\_Norm\_Take-2\_3x3\_Dropout\_0.50 - Test Set] time=0.675s, loss=0.0890, acc=0.9775

We show below graphs of previous runs where we tested the 5x5 model as well, over 220 epochs. We can see a much smoother curve here, reinforcing our intuition that the model we have is more stable.



Models Performance On Grayscale Equalized Images, With Dropout

We identified 269 images that are model could not identify correctly. We display 10 of them below, chosen randomly, to conjecture why the model was wrong.



Sample of 10 images where our model got the predictions wrong

Some of the images are very blurry, despite our histogram equalization, while others seem distorted. We probably don’t have enough examples of such images in our test set for our model’s predictions to improve. Additionally, while 97.75% test accuracy is very good, we still one more ace up our sleeve: data augmentation.

# Data Augmentation

We observed earlier that the data presented glaring imbalance across the 43 classes. Yet it does not seem to be a crippling problem as we are able to reach very high accuracy despite the class imbalance. We also noticed that some images in the test set are distorted. We are therefore going to use data augmentation techniques in an attempt to:

1. Extend dataset and provide additional pictures in different lighting settings and orientations
2. Improve model’s ability to become more generic
3. Improve test and validation accuracy, especially on distorted images

We use a nifty library called [imgaug](https://github.com/aleju/imgaug" \t "_blank) to create our augmentations. We mainly apply affine transformations to augment the images. Our code looks as follows:

def augment\_imgs(imgs, p):  
 """  
 Performs a set of augmentations with with a probability p  
 """  
 augs = iaa.SomeOf((2, 4),  
 [  
 iaa.Crop(px=(0, 4)), # crop images from each side by 0 to 4px (randomly chosen)  
 iaa.Affine(scale={"x": (0.8, 1.2), "y": (0.8, 1.2)}),  
 iaa.Affine(translate\_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)}),  
 iaa.Affine(rotate=(-45, 45)), # rotate by -45 to +45 degrees)  
 iaa.Affine(shear=(-10, 10)) # shear by -10 to +10 degrees  
 ])   
 seq = iaa.Sequential([iaa.Sometimes(p, augs)])  
   
 return seq.augment\_images(imgs)

While the class imbalance probably causes some bias in the model, we have decided not to address it at this stage as it would cause our dataset to swell significantly and lengthen our training time (we don’t have a lot of time to spend on training at this stage). Instead, we decided to augment each class by 10%. Our new dataset looks as 

Sample Of Augmented Images

The distribution of images does not change significantly of course, but we do apply grayscale, histogram equalization and normalisation pre-processing steps to our images. We train for 2000 epochs with dropout (p-conv=0.6, p-fc=0.5) and achieve**97.86% accuracy on the test set:**

[EdLeNet] Building neural network [conv layers=3, conv filter size=3, conv start depth=32, fc layers=2]  
Training EdLeNet\_Augs\_Grayscale\_CLAHE\_Norm\_Take4\_Bis\_3x3\_Dropout\_0.50 [epochs=2000, batch\_size=512]...  
  
[1] total=5.824s | train: time=3.594s, loss=3.6283, acc=0.0797 | val: time=2.231s, loss=3.6463, acc=0.0687  
...  
[1970] total=5.627s | train: time=3.408s, loss=0.0525, acc=0.9870 | val: time=2.219s, loss=0.0315, acc=0.9914  
[1980] total=5.627s | train: time=3.409s, loss=0.0530, acc=0.9862 | val: time=2.218s, loss=0.0309, acc=0.9902  
[1990] total=5.628s | train: time=3.412s, loss=0.0521, acc=0.9869 | val: time=2.216s, loss=0.0302, acc=0.9900  
[2000] total=5.632s | train: time=3.415s, loss=0.0521, acc=0.9869 | val: time=2.217s, loss=0.0311, acc=0.9902  
Model ./models/EdLeNet\_Augs\_Grayscale\_CLAHE\_Norm\_Take4\_Bis\_3x3\_Dropout\_0.50.chkpt saved[EdLeNet\_Augs\_Grayscale\_CLAHE\_Norm\_Take4\_Bis\_3x3\_Dropout\_0.50 - Test Set] time=0.678s, loss=0.0842, acc=0.9786

**Testing On New Images**

We decided to test our model on new images as well, to make sure that it’s indeed generalised to more than the traffic signs in our original dataset. We therefore downloaded five new images and submitted them to our model for predictions.



Download 5 new traffic signs — color

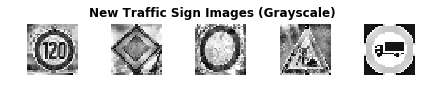
The ground truth for the images is as follows:

['Speed limit (120km/h)',  
 'Priority road',  
 'No vehicles',  
 'Road work',  
 'Vehicles over 3.5 metric tons prohibited']

The Images were chosen because of the following:

* They represent different traffic signs that we currently classify
* They vary in shape and color
* They are under different lighting conditions (the 4th one has sunlight reflection)
* They are under different orientations (the 3rd one is slanted)
* They have different background
* The last image is actually a design, not a real picture, and we wanted to test the model against it
* Some of them are in under-represented classes

The first step we took was to apply the same CLAHE to those new images, resulting in the following:



We achieve perfect accuracy of 100% on the new images. On the original test set, we achieved 97.86% accuracy. We could explore blurring/distorting our new images or modifying contrast to see how the model handles those changes in the future.

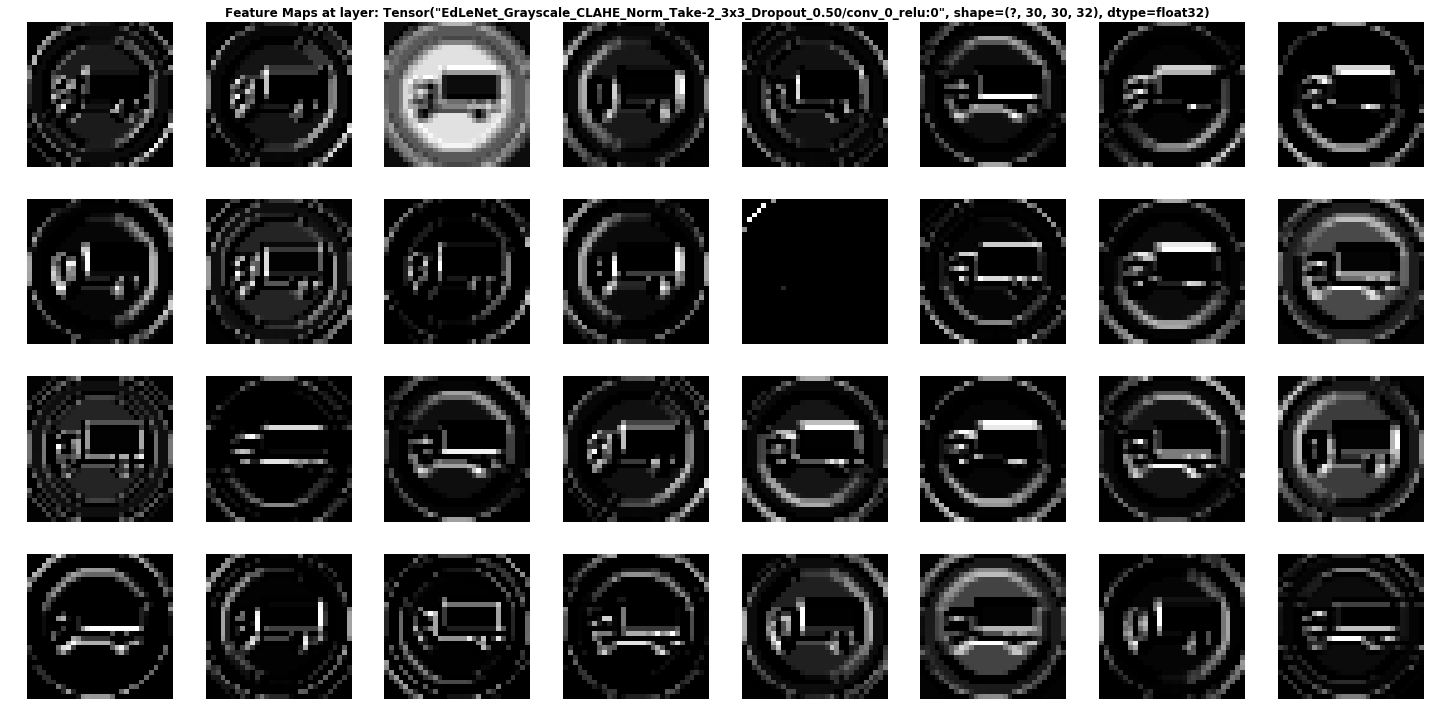
new\_img\_grayscale\_norm\_pred\_acc = np.sum(new\_img\_lbs == preds) / len(preds)  
print("[Grayscale Normalised] Predictional accuracy on new images: {0}%".format(new\_img\_grayscale\_norm\_pred\_acc \* 100))  
...  
[Grayscale Normalised] Predictional accuracy on new images: 100.0%

We also show the top 5 SoftMax probabilities computed for each image, with the green bar showing the ground truth. We can clearly see that our model is quite confident in its predictions. In the worst case (last image), the 2nd most likely prediction has a probability of around 0.1% . In fact our model struggles most on the last image, which I believe is actually a design and not even a real picture. Overall, we have developed a strong model!

# Visualizing Our Activation Maps

We show below the results produced by each convolutional layer (before max pooling), resulting in 3 activation maps.

## ****Layer 1****



We can see that the network is focusing a lot on the edges of the circle and somehow on the truck. The background is mostly ignored.

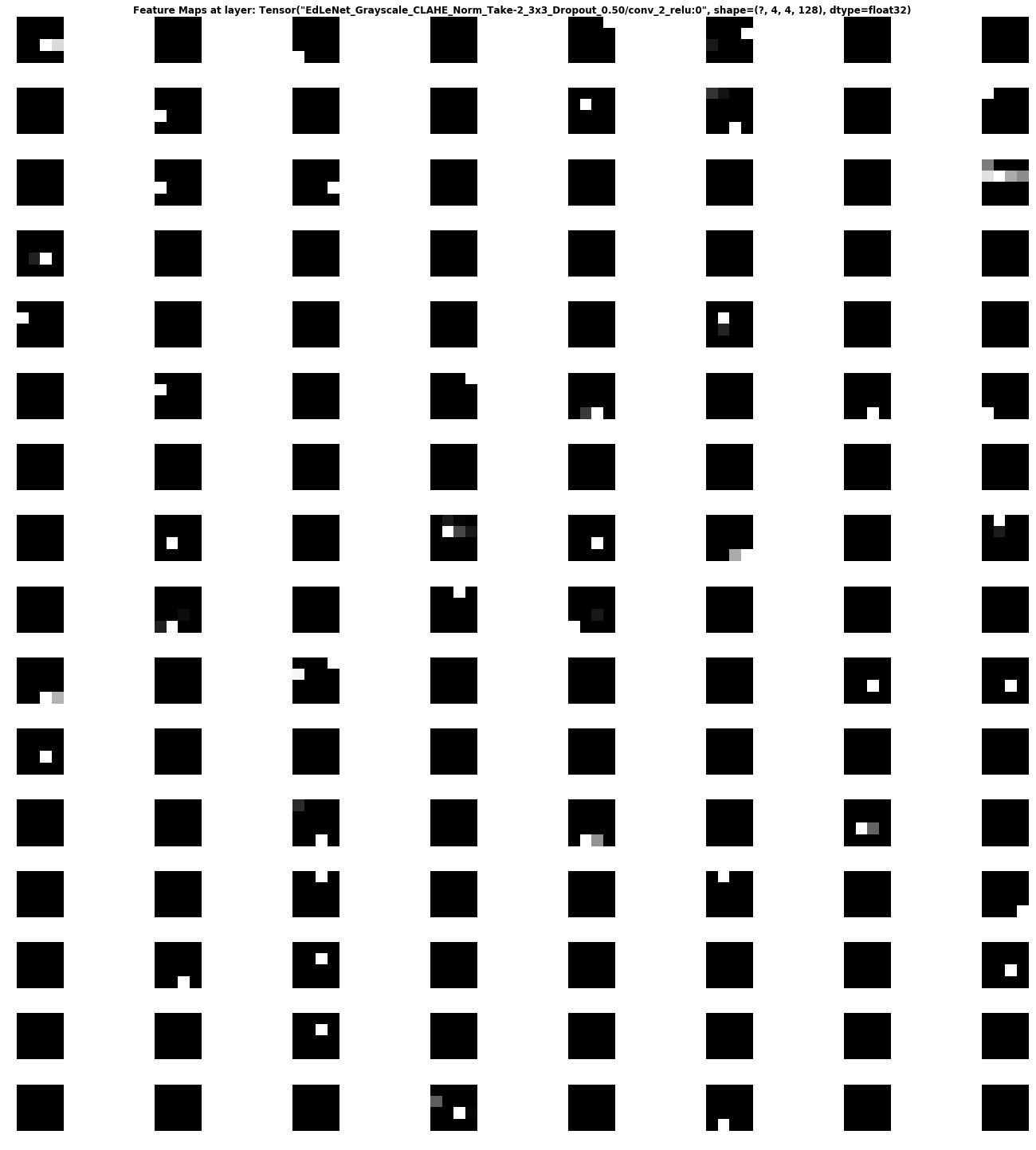
## Layer2:

Activation Map Of Second Convolutional Layer

It is rather hard to determine what the network is focusing on in layer 2, but it seems to “activate” around the edges of the circle and in the middle, where the truck appears.



**Layer3:**

****

This activation map is also hard to decipher… But it seems the network reacts to stimuli on the edges and in the middle once again.

# Building a Graphical User Interface

Now let’s take a step ahead and build a nice graphical user interface for our deep learning model. A graphical user interface will save a lot of time in testing and seeing the results of our model prediction. The Tkinter is an inbuilt library of python to make a graphical user interface.

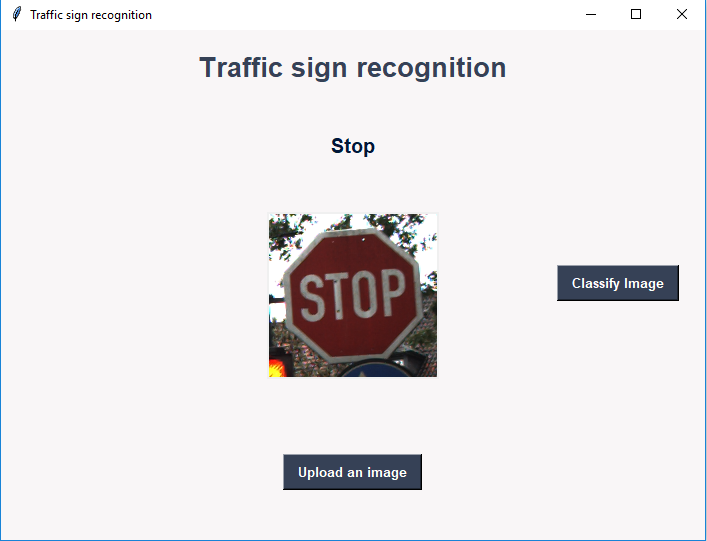
From the interface of the GUI application, we will ask the user for an image and extract the file path of the image. Then we use the trained model that will take the image data as input and provide us the class our image belongs to. We will then use the [**dictionary**](https://data-flair.training/blogs/python-dictionary/) to see the name of the class. Create a new python file, you can name it as traffic\_gui.py. Here’s the source code of our graphical user interface so you can run the file from the terminal using the “python traffic\_gui.py” command.

# Code:

import tkinter as tk  
from tkinter import filedialog  
from tkinter import \*  
from PIL import ImageTk, Imageimport numpy  
**#load the trained model to classify sign**from keras.models import load\_model  
model = load\_model(‘traffic\_recognition.h5’)**#dictionary to label all traffic signs class**classes = { 1:’Speed limit (20km/h)’,  
 2:’Speed limit (30km/h)’,  
 3:’Speed limit (50km/h)’,  
 4:’Speed limit (60km/h)’,  
 5:’Speed limit (70km/h)’,  
 6:’Speed limit (80km/h)’,  
 7:’End of speed limit (80km/h)’,  
 8:’Speed limit (100km/h)’,  
 9:’Speed limit (120km/h)’,  
 10:’No passing’,  
 11:’No passing veh over 3.5 tons’,  
 12:’Right-of-way at intersection’,  
 13:’Priority road’,  
 14:’Yield’,  
 15:’Stop’,  
 16:’No vehicles’,  
 17:’Veh > 3.5 tons prohibited’,  
 18:’No entry’,  
 19:’General caution’,  
 20:’Dangerous curve left’,  
 21:’Dangerous curve right’,  
 22:’Double curve’,  
 23:’Bumpy road’,  
 24:’Slippery road’,  
 25:’Road narrows on the right’,  
 26:’Road work’,  
 27:’Traffic signals’,  
 28:’Pedestrians’,  
 29:’Children crossing’,  
 30:’Bicycles crossing’,  
 31:’Beware of ice/snow’,  
 32:’Wild animals crossing’,  
 33:’End speed + passing limits’,  
 34:’Turn right ahead’,  
 35:’Turn left ahead’,  
 36:’Ahead only’,  
 37:’Go straight or right’,  
 38:’Go straight or left’,  
 39:’Keep right’,  
 40:’Keep left’,  
 41:’Roundabout mandatory’,  
 42:’End of no passing’,  
 43:’End no passing veh > 3.5 tons’ }def classify(file\_path):  
 image = Image.open(file\_path)  
 image = image.resize((30,30))  
 image = numpy.expand\_dims(image, axis=0)  
 image = numpy.array(image)  
 pred = model.predict\_classes([image])[0]  
 sign = classes[pred+1]  
 print(sign)  
 result.configure(text=sign)def show\_classify\_btn(file\_path):  
 classify\_b=Button(top,text=”Classify Image”,command=lambda: classify(file\_path),padx=10,pady=5)  
 classify\_b.configure(bg=’#364156', fg=’white’,font=(‘arial’,10,’bold’))  
 classify\_b.place(relx=0.79,rely=0.46)def upload\_image():  
 try:Whoa!!!  
 file\_path=filedialog.askopenfilename()  
 uploaded=Image.open(file\_path)uploaded.thumbnail(((top.winfo\_width()/2.25),(top.winfo\_height()/2.25)))  
 im=ImageTk.PhotoImage(uploaded) sign\_image.configure(image=im)  
 sign\_image.image=im  
 result.configure(text=’’)  
 show\_classify\_btn(file\_path)  
 except:  
 passif \_\_name\_\_==”\_\_main\_\_”:  
 **#initialise GUI** top=tk.Tk()  
 top.geometry(‘800x600’)  
 top.title(‘Traffic sign recognition’)  
 top.configure(bg=’#f9f6f7') heading = Label(top, text=”Traffic sign recognition”,pady=20, font=(‘arial’,20,’bold’))  
 heading.configure(background=’#f9f6f7',fg=’#364156')  
 heading.pack() result=Label(top, font=(‘arial’,15,’bold’))  
 result.configure(fg=’#011638',bg=’#f9f6f7') sign\_image = Label(top) upload=Button(top,text=”Upload an image”,command=upload\_image,padx=10,pady=5)  
 upload.configure(background=’#364156', fg=’white’,font=(‘arial’,10,’bold’)) upload.pack(side=BOTTOM,pady=50)  
 sign\_image.pack(side=BOTTOM,expand=True)  
 result.pack(side=BOTTOM,expand=True)  
 top.mainloop()

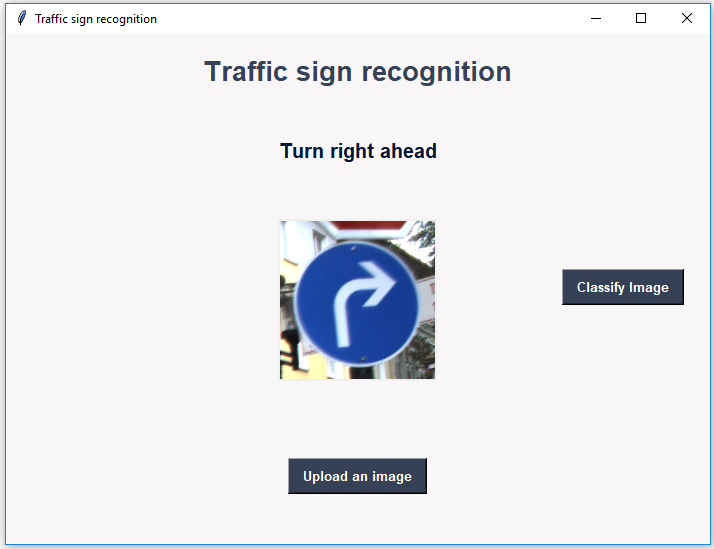
# Output

## 1.



## 2.





# SUMMARY

# understanding of how autonomous vehicles can take advantage of Computer vision and Deep Learning techniques to automatically recognize and classify from multiple classes. You get to learn how to implement a Convolutional neural network for image classification tasks. Moreover, we have to build a nice interface to ease up our interaction with traffic sign recognition.

**References:-**

* [*Bishop, C. M.*](https://en.wikipedia.org/wiki/Christopher_M._Bishop) (2006), Pattern Recognition and Machine Learning, Springer, [*ISBN*](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [*978-0-387-31073-2*](https://en.wikipedia.org/wiki/Special:BookSources/978-0-387-31073-2)
* [*Friedman, Jerome H.*](https://en.wikipedia.org/wiki/Jerome_H._Friedman) (1998). "Data Mining and Statistics: What's the connection?". Computing Science and Statistics. **29** (1): 3–9.
* Mitchell, T. (1997). Machine Learning. McGraw Hill. p. 2. [*ISBN*](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [*978-0-07-042807-2*](https://en.wikipedia.org/wiki/Special:BookSources/978-0-07-042807-2).
* [*Jeff Leek*](https://en.wikipedia.org/wiki/Jeffrey_T._Leek) (12 December 2013). [*"The key word in "Data Science" is not Data, it is Science"*](http://simplystatistics.org/2013/12/12/the-key-word-in-data-science-is-not-data-it-is-science/). Simply Statistics.
* Donoho, David (18 September 2015). [*"50 years of Data Science"*](http://courses.csail.mit.edu/18.337/2015/docs/50YearsDataScience.pdf)
* Data science and its application. Escoufier, Yves., Hayashi, Chikio (1918.)., Fichet, Bernard. Tokyo: Academic Press/Harcourt Brace. 1995
* Deep learning for computer vision(pyimagesearch.com)
* Datascience with python(dataflair.com)
* [Contrast Limiting Adaptive Histogram Equalization](http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_histograms/py_histogram_equalization/py_histogram_equalization.html) (aka CLAHE) function. And lane line detection by towardsdatascience.com
* Medium.com articles on datacience and visualization
* Tensorflow.org
* Keras.io
* Anaconda.com
* Opencv.org

**(P.S)Conclusion:-**

The project is made successfull by our team(171FA04007,171FA04023,171FA04554,171FA04556) efforts and massive help from various articles and documentations mentioned in references.

Our complete project is available in my github repo link including the dataset,code.

<https://github.com/sairam-py/Traffic_signal_detection>

The project can be executed using anaconda. python 3.6 installed and on tensorflow 2.0 environment with keras 2.3.1

contact via [sairamkollimarla11@gmail.com](mailto:sairamkollimarla11@gmail.com) if there are any issues/queries.