EURECOM

MALIS - Project

Road Signs Classification with Machine Learning

Completed by: **Adib RACHID**
Mokhles BOUZAIEN

To: **Prof. Maria ZULUAGA**

Fall 2019-2020

Contents

i. List of figures

I. Project Definition

A car company's commercial project would be a combination of detection and classification of road signs inside the car software. This project is highly recommended for autonomous cars and even to automate some car functions such as alerting drivers on a limit speed or other road signs.
However, in this project, the objective will be to work on only classifying road signs into their correct classes ex: speed limit, no stopping, no entry… The difficulty can increase by knowing more information about these classes ex: speed limit value, maximum height value… and then, by detecting the road signs as a further step. This problem can be considered as a computer vision problem so deep learning may be required to solve the classification in order to extract features from the images and use them to correctly classify the image to its exact class.
This report will contain the steps taken in order to implement the project and the required information which are composed in many sections:

- Dataset chosen in order to train and test our model.
- Theoretical part behind some knowledge for the implementation.
- Results of the project by comparing many implementations, many models, and many variations.
- Some improvements that can be done for the implementation in order to enhance the accuracy and the performance of the model to be used in real world.
- References we used in order to complete our work.

II. Data Set

There are many data sets online related to traffic signs, but we chose to work on the 'German Traffic Sign Benchmark' which is a dataset containing single image in a multiclass environment. It has 43 different traffic sign classes labeled from 0 to 42.

Dataset can be downloaded using the following URL link from Kaggle website:

https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign

The dataset contains:

- 3 subfolders:
    - Meta: contains metadata of the different classes.
    - Test: contains around 25000 images to be tested which are from different classes.
    - Train: contains a directory for each class, containing a number of images for the specified class.
- 3 csv label files:
    - Meta.csv: labels of the Meta folder.
    - Test.csv: labels of the Test folder.
    - Train.csv: labels of the Train folder.

III. Implementation

The code will be attached in a folder along with this report and will also be available on our Eurecom Gitlab account following this link:

https://gitlab.eurecom.fr/malis-group13/traffic-signs-detection-and-classification

Running steps:

First of all, we create a new directory called **data** and unzip the downloaded file from Kaggle which contains 3 csv files and 3 directories (Test, Train and Meta).

- Run the python script augmentation.py which will create a new directory called augmented under data having the train directory with augmented data.
- In the Jupyter notebook containing the code you might have to modify the path of the cur path to your project directory.

1. Exploring Data

Training data do not contain the same number of images for all classes, so a data augmentation was needed (explored in Implementation section). Their distribution can be seen in the below

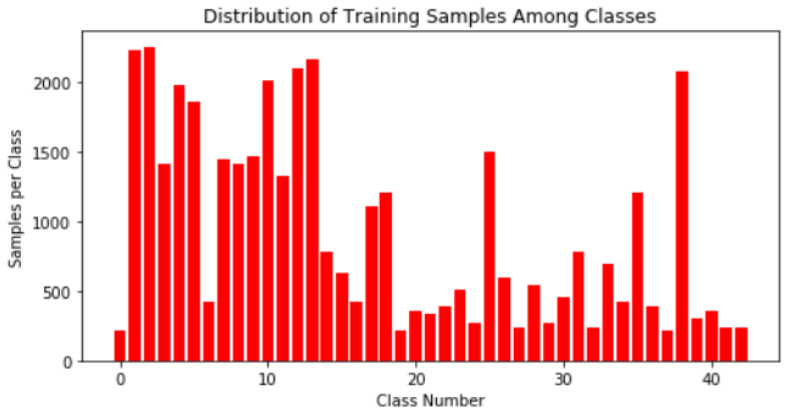

figure.

Labels will contain the position of the sign with some meaningful positions of the position of the sign which could have been used in order to reduce non necessary information from the image by cropping it, along with the path of the image in its corresponding folder.

The problem found with the dataset is that training images where taken from videos. For instance, 30 consecutive images approximately would be from the same class, because they are taken from the same video capturing the same traffic sign at different time stamps with a small interval.

Below are the traffic signs that will be trained and tested on.

Figure 1- Distribution of training samples among classes

Figure 2- Meta Data

Knowing that in real life, we will not have the same scenario of having the traffic sign in the middle of the image, very clear, no noise, good light resolution, etc. Training data will contain images from different scenarios and a sample can be seen below:

Figure 3- Testing Data

We can notice that some images are not clear for the naked eye but these are taken from real videos so cameras will have to face such images that's why they are very useful for training.
Images also have different brightness levels and sizes as we can see in the next figure.

Figure 4- Distribution of images

We will deal with those problems in the preprocessing part.

2. Data Augmentation

This task has a major role in the performance of the model due to the variation in the number of images of each class.

First of all, some classes in the training directory needed to have a data augmentation due to the big variance in the number of images in different classes as seen in a previous section. Data augmentation is normally used to increase the diversity of data available in the training data without the need of collecting new data. There are different techniques that can be used as cropping, padding, flipping, changing colors.

In our case, we did some rotations and random scaling from the borders. The parameters of the scaling were randomly chosen as for the rotations, it depends on the needed number of images. Each image will go through a rotation step then a scaling step.

For the flow, we will have all classes containing the same number of images which is the maximum number of images of all the classes initially.

This step should be done one time, before running the code (not every time we run the code), and the new generated will be saved locally. The script to be called using python is ‘augmentation.py’.

After doing the augmentation, we don’t have anymore the variation of number of images between classes as seen in the below figure.

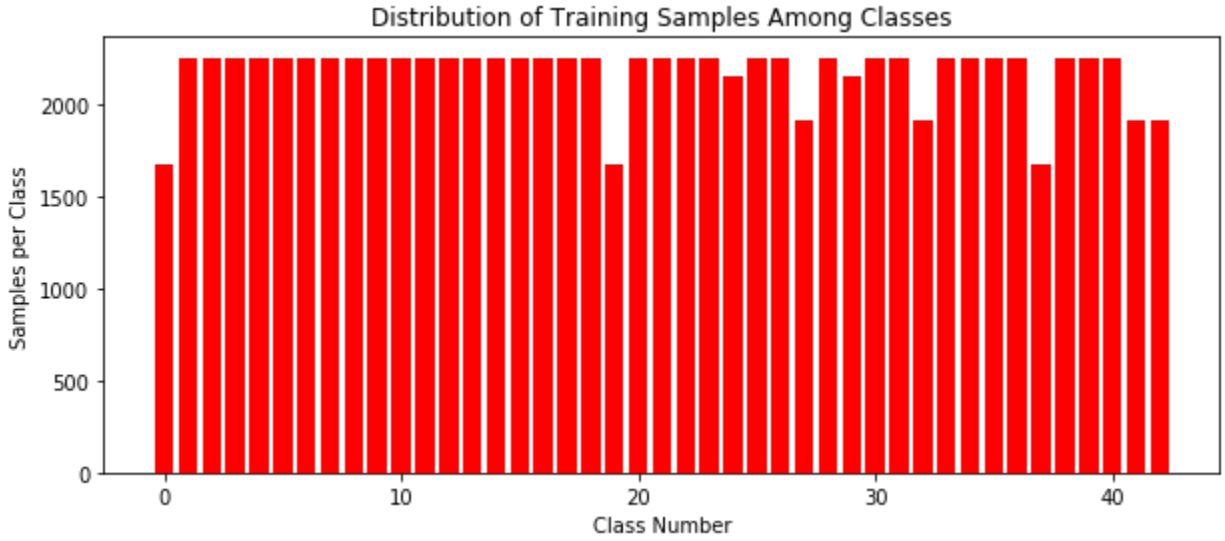

Figure 5- Distribution of Training Samples after data augmentation

3. Data Preprocessing

In this part, we are mainly applying two preprocesses in order to get input samples with similar properties. First step is converting images from RGB to grayscale and applying histogram equalization in order to improve contrast. Then all images are resized to 32x32. This way, image are small enough to be processed as fast as possible and not too small to keep all information we need. Converting to grayscale images won’t cause information loss because all we need is shapes and edges but not colors.

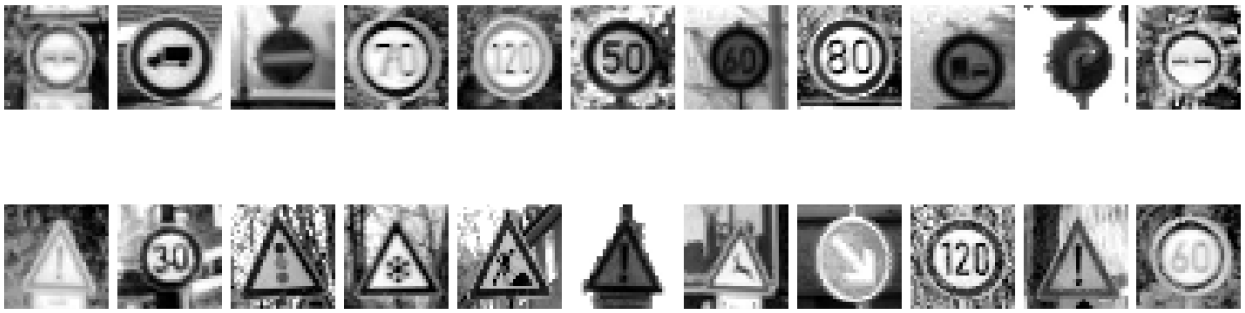

Figure 6- Preprocessed Data

4. Models

We have worked on two main implementations:

- By creating our own neural network from scratch.
- By using TensorFlow

a. Using our own Neural Network from Scratch

We started by creating a simple Neural Network model taking a 1024-dimensional vector as input. The input features are the values of each pixel in the 32x32 image.
The main problem with this model was the lack of resources to train it (training one epoch with a two-layer neural network took more than 6 hours on Google Colab server). So we were unable to change and test with different model parameters such as number of hidden layers, number of neurons in each layer, learning rate, number of epochs, etc.

b. Using TensorFlow

TensorFlow is an open source AI library which uses data flow graphs to build models and one main use for it is for classification. It simplifies the process of creating everything from scratch.
Split Folders into training, validation and testing:
For this part, we will use the 'split_folders.py' script in order to split the data into 3 folders by giving the ratio of the images needed for training (75%), validation (25%) and testing will be in

another folder along with the seed. After this step, we will have 3 different folders: test, train and val which will be used by TensorFlow libraries.

Load directories:

Directories have to be mapped to variables and this is done by creating lists containing the images path.

Model definition:

For the models, we tested our own sequential basic model and ResNet50 which can be found in the notebook.

It is a convolutional neural network since it has at least one convolutional layer. It also has convolutional layers, pooling layers among with dense layers.

- A convolutional layer has a convolutional filter over its input matrix which is an actor of the convolutional operation in order to see which image parts to keep.
- A pooling layer to reduce a matrix to a smaller one by taking the maximum or average value across a pooled area.
- A dense layer which is a fully connected layer

.

CNN is very known in image classification problems.

The basic model is shown in the below image.

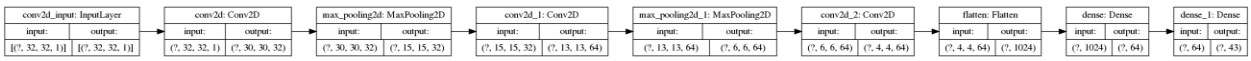

Figure 7- Basic Model

The RseNet50 model took too much time on training 1.5 hours per epoch and had very bad results on 4 classes. We might assume that the model is too complex for 4 classes but it should have been tested on all the classes but could take a long time for training.

After defining the models, we compile it. Then, we create a data generation for the training and the validation by rescaling the images. After that, we run the fit generator in order to train the

model giving the training and validation data. The weights and the model will be saved in a ‘models’ directory.

Prediction:

We load the model with its weights in case we don’t want to train it again, and we predict the output with a predict function from TensorFlow by giving the testing image for each of the testing images.

Visualization:

We visualized the results by showing the training and validation accuracy along with their loss.

IV. Results

The training set is composed of around 80000 images from 43 classes.

Model parameters:

epochs = 15
img_width, img_height = 32, 32
batch_size = 32
samples_per_epoch = 1000
validation_steps = 300
nb_filters1 = 32
nb_filters2 = 64
conv1_size = 3
conv2_size = 2
pool_size = 2
classes_num = 9
lr = 0.004

Basic Model Testing Results

4 classes:

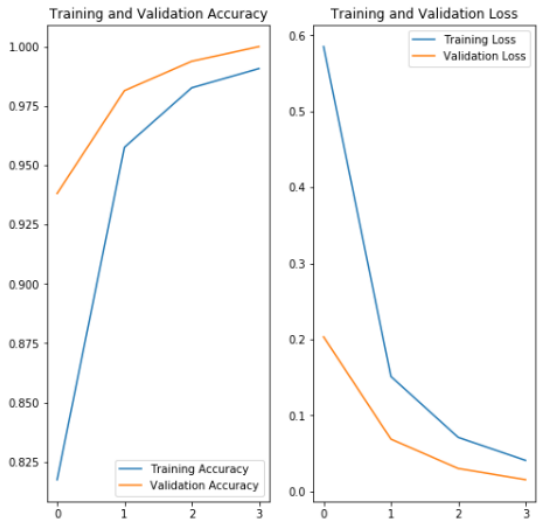

Figure 8- Basic Model on 4 classes

Testing results:

```
Execution Time: 26.383342742919922 seconds
```

```
Correct: 319
Incorrect: 6
```

We can notice that we have a high training and validation accuracy from the graph above. The testing results are also good, there are 6 incorrect vs 319 correct images which is good.

9 classes:

Figure 9- Basic Model on 9 classes

```
Execution Time: 5.876867783069611 minutes
Correct: 46
Incorrect: 555
```

The results on the testing data are really bad, 46 correct vs 555 incorrect.

43 classes:

Figure 10- Basic Model on 43 classes

```
Execution Time: 21.989105463027954 seconds
```

```
Correct: 32
Incorrect: 293
```

When trained and tested on 43 classes, we can see that there was a kind of overfitting, the training accuracy was very high, validation accuracy low and testing results very low only 32 correct vs 293 incorrect.

- ⇨ **OVERFITTING!**

After seeing such results, we notice that our model is overfitting.
In fact, training data should be filtered since it contains 30 images from the same scene but different time frame, it could be that it is trying to overfit on these images knowing that the testing images are from elsewhere.

A solution to this would be to filter out the images and to add more complexity to the data augmentation not only shifting and a small rotation.

Next step was to enhance the model to add more layers but we are getting some implementation problems with this part.

Below we can also find a model accuracy and loss of a model using keras without augmentation and the second with augmentation.

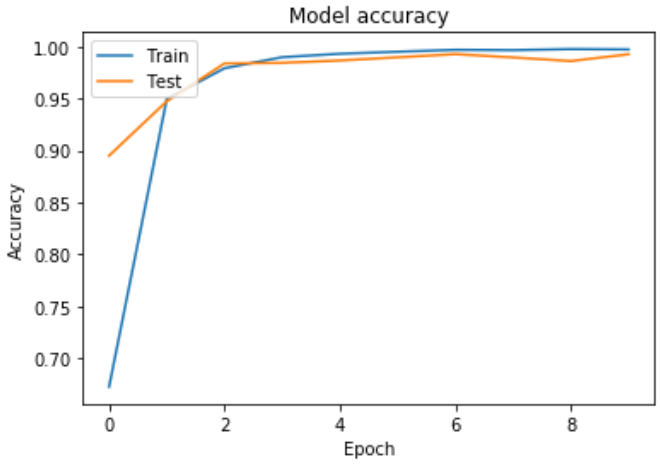

Figure 11-Model accuracy 1

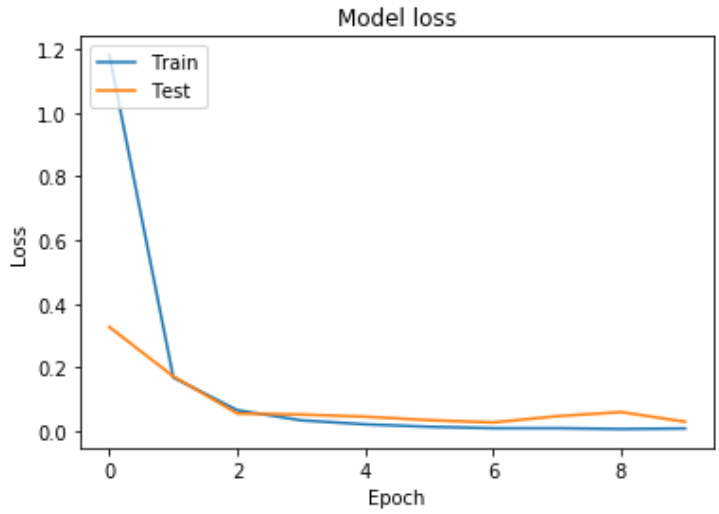

Figure 12- Model loss 1

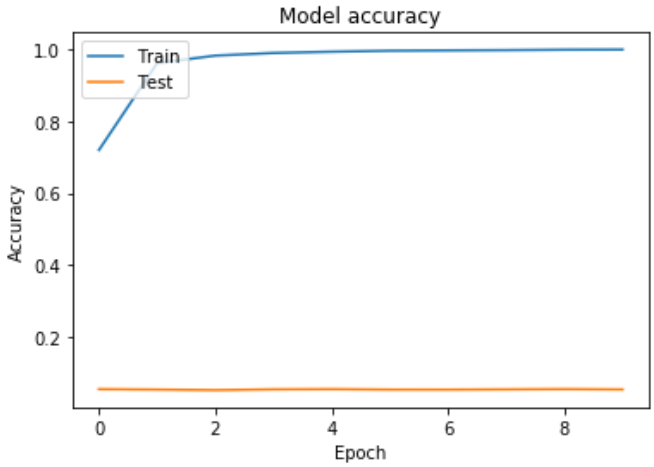

Figure 13- Model accuracy 2

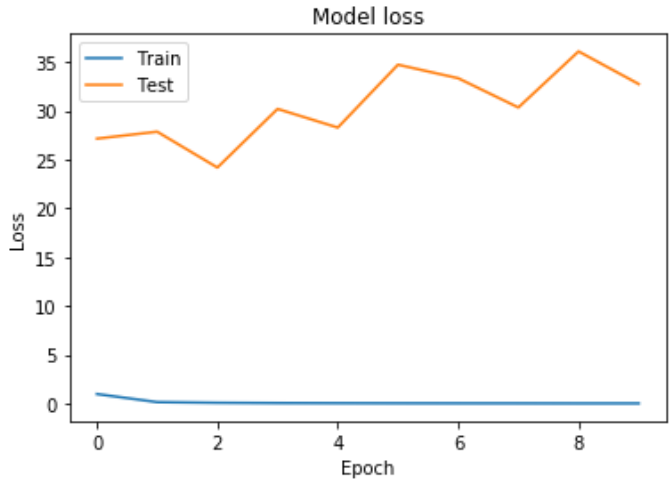

Figure 14- Model loss 2

V. Improvement Ideas

Below are some ideas or suggestions that can be made or tried which can affect positively the results of this project:

- Test other Neural Network models in order to find the most performant one for our case.
- Use other Visualization techniques in order to measure the performance of our model.
- Use Variational Auto-Encoders in order to have a robust model ignoring the noise in the images which works well on the variations of data within the same class.
- Use (or find if not given) the labels of the sign position by removing everything else not needed in the picture.
- Use other preprocessing features for the training images.
- Modify the data augmentation script to be more optimized and not generate a number of images equal to the maximum number of images of all the classes.

VI. References

TensorFlow, https://www.tensorflow.org/

Kaggle German Traffic Sign Recognition Benchmark,

https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign

Prof. Maria Zuluaga lectures and labs, EURECOM