Understanding street signs

Collector: 1305052

Coder: 1305005

Trainer: 1305012

Writer: 1305029

Leader: 1305002

# Project Overview

An autonomous car is a self-driving car which can travel between destinations without any human operator. It is the future of transport system as it reduces mobility and infrastructure costs, increases safety and mobility, increases customer satisfaction and reduces crime. It will relieve travelers from driving, lower fuel consumption, reduce crime, and provide improved mobility to the elderly, the children and the disabled. Traffic flow will also be smooth and there will be less traffic congestion if there are more autonomous cars in the roads.

Experts predict that once driverless technology has been fully developed, traffic collisions (and resulting deaths and injuries and costs), caused by human error, such as delayed reaction time, tailgating, rubbernecking, and other forms of distracted or aggressive driving should be substantially reduced. Autonomous cars could reduce labor costs relieve travelers from driving and navigation chores, thereby replacing behind-the-wheel commuting hours with more time for leisure or work;[6][86] and also would lift constraints on occupant ability to drive, distracted and texting while driving, intoxicated, prone to seizures, or otherwise impaired. For the young, the elderly, people with disabilities, and low-income citizens, autonomous cars could provide enhanced mobility. Safer driving is expected to reduce the costs of vehicle insurance. Reduced traffic congestion and the improvements in traffic flow due to widespread use of autonomous cars will also translate into better fuel efficiency.

Autonomous cars combine a variety of techniques to perceive their surroundings, including radar, laser light, GPS, odometry, and computer vision. Advanced control systems interpret sensory information to identify appropriate navigation paths, as well as obstacles and relevant signage.

Automated cars must abide by all the traffic rules. So, they have to read the traffic signals perfectly. For tracing the route and highest speed limits for the segments of the roads, the locations of speed breakers and so on, the have to extract real data from traffic signals on traffic.   
  
In our project, we work to build up a system which helps to extract and diagnose the traffic signal from a photo (with or without traffic signals) taken from the street.

# Problem Statement

The datasets are in data.zip in Data folder of our project. There are data from several classes of traffic signals. The data are taken from [The German Traffic Sign Recognition Benchmark](http://benchmark.ini.rub.de/?section=gtsrb&subsection=news)(GTSRB). The problem is to recognize the traffic sign from the images. Solving this problem is essential for self-driving cars to operate on roads.

We are going to perform to major tasks here.

1. Segmentation: Find the bounding box that carries the traffic signal
2. Detection: Detect which signal is this.

# Performance Metrics

Out project has two major portion- one is segmentation and another is classification. For segmentation our evaluation matrix is Mean Average Precision (MAP). It calculates the overlapping area between the given bounding box and the predicted bounding box.

For classification problem, we used accuracy as our performance measure as the dataset we used is balanced.

# Data Description

We have used GTSRB traffic signal dataset. We have classified two types of traffic sign.

Figure : Class 0 (Road Speed Breaker) Figure : Class 1 (Attention sign)

For segmentation, we have total 24 images with environment as background. 16 of the images belong to class 1 and 8 of them belong to class 0. For classification we have used cropped image which contains no environment, only the traffic signal. We have used 80 images of class 0 and 100 images of class 1.

# Data preprocessing

For applying Yolo for image segmentation square image is needed. So, first we resized the 24 images to a size 800x800. The training dataset of 24 images is very small. So, we have applied scaling and translation on these images and generated 1752 images for training. Then we generated an annotation file for each image which is in xml format and which contains the bounding box of the target region.

For classification, we again resized the cropped image to 48x48 pixels. Traffic signal images may be very bright in a sunny day or may be very dark in a rainy day. For this reason, we have applied histogram normalization on them. As the contrast is normalized, it becomes easy to classify the image.

# Architecture Description

**Convolutional neural network (CNN)**

For classification, we have used convolutional neural network. We have used feed forward network with convoluted layers followed by fully connected hidden layer. We have also used dropout layer in between. Description of the layers are given below

1. Convolution: Hyper-parameters of convolutional layer are the number of filters, filter size, padding, stride and activation function. This layer creates a convolution kernel that is convolved with the layer input over a single spatial (or temporal) dimension to produce a tensor of outputs. Convolutional layers convolve around the image to detect edges, lines, blobs of colors and other visual elements.

2. Max-pooling: Pooling layer removes some of the pixels of the image and thus reduces the dimension of the image. Max-pooling replaces an n x n area of an image to the maximum pixel value of that area and thus down-samples the image.

3. Dropout: Dropout regularizes the network and prevents the network from over-fitting.

4. Flatten: This layer flattens the output of the convolution layer to feed into the dense layer.

5. Dense: Dense layers are the fully connected layers. It maps the score of previous layer to the correct label.

We have used two types of activation function-

1. Relu: We have used relu activation in all the convolution layer. Relu computes the function f(x) = max(0,x) to threshold the activation at 0.
2. Softmax: We have used softmax activation in the output layer to convert the scores into probabilities that sum to 1.

Our architecture contains multiple series of two convolutional layers, one pooling layer and one flatten layer and one or two dense layer. The block diagram is shown below-

Input Conv(3x3) Conv(3x3) Maxpool(2x2) Flatten Dense

*Figure 3: Block diagram of CNN architecture*

## Yolo

Yolo architecture is divided in front-end and back-end. In, back-end there are some pre trained models which work as the basic feature extractors. Front end uses the extracted features provided by the back-end to detect the target object.

There are several back-ends available like Full Yolo, Tiny Yolo, Inception3, VGG16, ResNet50, MobileNet, SqueezeNet etc. We used Full Yolo in this project.

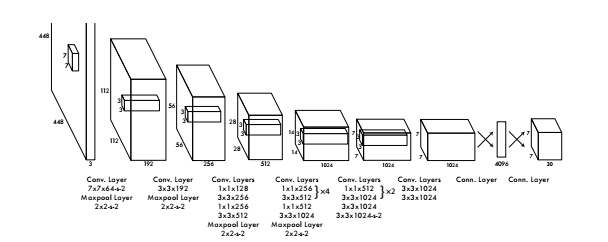


Figure 4: Full Yolo Architecture

# Model description:

## Yolo

For all the layers, we used stride=1.

Table 1: Model for Full Yolo Back-end

|  |  |  |  |
| --- | --- | --- | --- |
| Layer | Kernel | Output shape | Trainable parameters |
| Input |  | 800X800X3 | 0 |
| Convolution | 3X3 | 800X800X3 | 32\*3\*3\*3 |
| MaxPooling | 2X2 | 400X400X3 | 0 |
| Convolution | 3X3 | 400X400X3 | 64\*3\*3\*3 |
| MaxPooling | 2X2 | 200X200X3 | 0 |
| Convolution | 3X3 | 200X200X3 | 128\*3\*3\*3 |
| Convolution | 1X1 | 200X200X3 | 64\*1\*1\*3 |
| Convolution | 3X3 | 200X200X3 | 128\*3\*3\*3 |
| MaxPooling | 2X2 | 100X100X3 | 0 |
| Convolution | 3X3 | 100X100X3 | 256\*3\*3\*3 |
| Convolution | 1X1 | 100X100X3 | 128\*1\*1\*3 |
| Convolution | 3X3 | 100X100X3 | 256\*3\*3\*3 |
| MaxPooling | 2X2 | 50X50X3 | 0 |
| Convolution | 3X3 | 50X50X3 | 512\*3\*3\*3 |
| Convolution | 1X1 | 50X50X3 | 256\*1\*1\*3 |
| Convolution | 3X3 | 50X50X3 | 512\*3\*3\*3 |
| Convolution | 1X1 | 50X50X3 | 256\*1\*1\*3 |
| Convolution | 3X3 | 50X50X3 | 512\*3\*3\*3 |
| MaxPooling | 2X2 | 25X25X3 | 0 |
| Convolution | 3X3 | 25X25X3 | 1024\*3\*3\*3 |
| Convolution | 1X1 | 25X25X3 | 512\*1\*1\*3 |
| Convolution | 3X3 | 25X25X3 | 1024\*3\*3\*3 |
| Convolution | 1X1 | 25X25X3 | 512\*1\*1\*3 |
| Convolution | 3X3 | 25X25X3 | 1024\*3\*3\*3 |
| Convolution | 3X3 | 25X25X3 | 1024\*3\*3\*3 |
| Convolution | 3X3 | 25X25X3 | 1024\*3\*3\*3 |
| Convolution | 1X1 | 25X25X3 | 64\*1\*1\*3 |
| Convolution | 3X3 | 25X25X3 | 1024\*3\*3\*3 |

## CNN

We have used 3 CNN models. In all layers of all models we have used stride = 1. The layers of the models are described below in the tables sequentially-

Table 2: Model 1 for CNN

|  |  |  |  |
| --- | --- | --- | --- |
| Layer | Kernel | Output Shape | Trainable Parameters |
| Convolution | 3x3 | 48x48x3 | 32\*3\*3\*3 |
| Convolution | 3x3 | 48x48x3 | 32\*3\*3\*3 |
| Maxpooling | 2x2 | 24x24x3 | 0 |
| Convolution | 3x3 | 24x24x3 | 64\*3\*3\*3 |
| Convolution | 3x3 | 24x24x3 | 64\*3\*3\*3 |
| Maxpooling | 2x2 | 12x12x3 | 0 |
| Convolution | 3x3 | 12x12x3 | 128\*3\*3\*3 |
| Convolution | 3x3 | 12x12x3 | 128\*3\*3\*3 |
| Maxpooling | 2x2 | 6x6x3 | 0 |
| Flatten |  | 108 | 0 |
| Dense |  | 512 | 108\*512 |
| Dense |  | 2 | 512\*2 |

Table 3: Model 2 for CNN

|  |  |  |  |
| --- | --- | --- | --- |
| Layer | Kernel | Output Shape | Trainable parameters |
| Convolution | 3x3 | 48x48x3 | 32\*3\*3\*3 |
| Convolution | 3x3 | 48x48x3 | 32\*3\*3\*3 |
| Maxpooling | 2x2 | 24x24x3 | 0 |
| Convolution | 3x3 | 24x24x3 | 64\*3\*3\*3 |
| Convolution | 3x3 | 24x24x3 | 64\*3\*3\*3 |
| Maxpooling | 2x2 | 12x12x3 | 0 |
| Flatten |  | 432 | 0 |
| Dense |  | 512 | 432\*512 |
| Dense |  | 2 | 512\*2 |

Table 4: Model 3 for CNN

|  |  |  |  |
| --- | --- | --- | --- |
| Layer | Kernel | Output Shape | Trainable parameters |
| Convolution | 3x3 | 48x48x3 | 32\*3\*3\*3 |
| Convolution | 3x3 | 48x48x3 | 32\*3\*3\*3 |
| Maxpooling | 2x2 | 24x24x3 | 0 |
| Convolution | 3x3 | 24x24x3 | 64\*3\*3\*3 |
| Convolution | 3x3 | 24x24x3 | 64\*3\*3\*3 |
| Maxpooling | 2x2 | 12x12x3 | 0 |
| Flatten |  | 432 | 0 |
| Dense |  | 2 | 432\*2 |

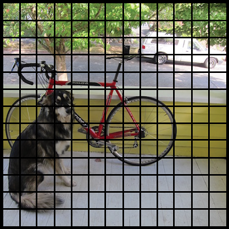
Table: Model 3 for CNN

# Algorithm and Techniques

## Yolo:

YOLO actually looks at the image just once (hence its name: You Only Look Once) but in a clever way.

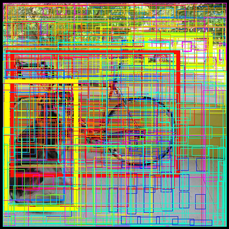
YOLO divides up the image into a grid of 25 by 25 cells:

[](http://machinethink.net/images/yolo/Grid@2x.png)

Each of these cells is responsible for predicting 5 bounding boxes. A bounding box describes the rectangle that encloses an object.

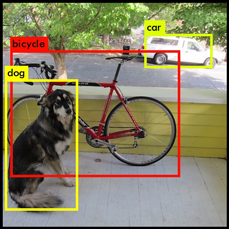
YOLO also outputs a confidence score that tells us how certain it is that the predicted bounding box actually encloses some object. This score doesn’t say anything about what kind of object is in the box, just if the shape of the box is any good.

The predicted bounding boxes may look something like the following (the higher the confidence score, the fatter the box is drawn):

[](http://machinethink.net/images/yolo/Boxes@2x.png) [](http://machinethink.net/images/yolo/Scores@2x.png)

Since there are 25×25 = 625 grid cells and each cell predicts 5 bounding boxes, we end up with 3125 bounding boxes in total. It turns out that most of these boxes will have very low confidence scores, so we only keep the boxes whose final score is 30% or more (you can change this threshold depending on how accurate you want the detector to be).

The final prediction is then:

[](http://machinethink.net/images/yolo/Prediction@2x.png)

From the 3125 total bounding boxes we only kept these three because they gave the best results. But note that even though there were 3125 separate predictions, they were all made at the same time — the neural network just ran once. And that’s why YOLO is so powerful and fast.

Our final layer predicts bounding box coordinates. We normalize the bounding box width and height by the image width and height so that they fall between 0 and 1. We parametrize the bounding box x and y coordinates to be offsets of a particular grid cell location so they are also bounded between 0 and 1.

We use a linear activation function for the final layer and all other layers use the following leaky rectified linear activation:

We optimize for sum-squared error in the output of our model. We use sum-squared error because it is easy to optimize, however it does not perfectly align with our goal of maximizing average precision. It weights localization error equally with classification error which may not be ideal.

Also, in every image many grid cells do not contain any object. This pushes the “confidence” scores of those cells towards zero, often overpowering the gradient from cells that do contain objects. This can lead to model instability, causing training to diverge early on.

To remedy this, we increase the loss from bounding box coordinate predictions and decrease the loss from confidence predictions for boxes that don’t contain objects. We use two parameters, to accomplish this. We set . Sum-squared error also equally weights errors in large boxes and small boxes. Our error metric should reflect that small deviations in large boxes matter less than in small boxes. To partially address this we predict the square root of the bounding box width and height instead of the width and height directly.

YOLO predicts multiple bounding boxes per grid cell. At training time we only want one bounding box predictor to be responsible for each object. We assign one predictor to be “responsible” for predicting an object based on which prediction has the highest current IOU with the ground truth. This leads to specialization between the bounding box predictors. Each predictor gets better at predicting certain sizes, aspect ratios, or classes of object, improving overall recall.

# Hyper-parameter Tuning

For tuning hyper-parameter, we have used a smaller dataset. For training in segmentation, we have taken 250 images and for validation, we have taken 175 images from the 1752 images. For training in classification, we have split the entire dataset into 8:2 ratio for training and validation.

## Tuning in segmentation

If one image is trained only once while training, very little update is done on weights. So, we trained one image multiple times. This parameter is called train time. We have used train time 2, 4 and 8. We have used epoch 1, 2 and 3 while training. We have tried for batch size greater than 1 but failed each time for limited machine configuration. So, we used batch size one only.

## Tuning in classification

We have tuned with the three CNN models we have developed.

# Result Description

We have kept two hyper-parameter fixed and tuned the other. We can see that we get the best result for train times 4, Yolo epochs 3 and CNN model 1.

We have trained the ‘best\_hyperparameter\_80\_percent’ training dataset with our best model and saved the Yolo model as model1.h5 and the classifier model as model2.h5. The results of tuning is shown below-

Table 5: Hyper-parameter tuning result

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hyper-parameter 1 Train Times | Hyper-parameter 2 Yolo Epochs | Hyper-parameter 3 CNN Model | Mean Average Precision | Train score | Validation score |
| 2 | 1 | 1 | 0.0 | 0.9097 | 0.9444 |
| 4 | 0.3029 | 0.8611 | 0.9444 |
| 8 | 0.2368 | 0.8681 | 1.0 |
| 4 | 1 | 1 | 0.2442 | 0.8958 | 1.0 |
| 2 | 0.5988 | 0.9583 | 1.0 |
| 3 | 0.7128 | 0.8611 | 0.9444 |
| 4 | 2 | 1 | 0.7127 | 0.8333 | 0.9167 |
| 2 | 0.7127 | 0.9931 | 1.0 |
| 3 | 0.7127 | 0.9931 | 1.0 |

First, we have detected our desired bounding box of traffic signal from the images of ‘Test\_10\_percent’ folder using model1.h5. Then cropping the images according to the detected bounding box, we have generated our test dataset for classification. Finally, we have classified the cropped dataset with model2.h5. In our test run, we have got the following results-

Table 6: Final training and test result

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hyper-parameter 1 Train Times | Hyper-parameter 2 Yolo Epochs | Hyper-parameter 3 CNN Model | Mean Average Precision | Train score | Test score |
| 4 | 3 | 1 | 0.9244 | 0.9719 | 0.9138 |



**Yolo Model**

Input image (800x800)





Cropped Street Sign

# Detected Street Sign

# Class 1

**CNN Model**

Figure 5: Flow Chart of street sign detection and classification

# Challenges

Image segmentation requires high configuration machine. If we want to classify all the street signs of GTSRB (43 street signs), we need at least 1000 images with background of each class to train a proper segmentation model. As we don’t have access to any high configuration machine, we have taken only two classes. For segmentation, we have tried different architectures. First we tried Haarcascade. It takes the background as negative image and the desired street sign as positive image and generates a dataset for training by placing the positive image randomly onto the negative image. In this approach, we need a large amount of different environment images. We tried with 1000 images but failed to detect any bounding box. Then we tried FRCNN. FRCNN generates sliding window of different sizes and applies CNN in each sliding window. This approach requires a huge runtime and computation and we again failed for lacking of high configuration machine. We also tried RetinaNet which is a modified version of FRCNN and tries to optimize the sliding window size. But we could not get any success here also. Finally, we became successful using Yolo. Yolo is light-weighted than any other architectures described above but uses a lot of memory at a time.

We didn’t face any challenges in classification.

# Conclusion

We have developed a tool for predicting two traffic signs. We can use the same approach to classify all the street signs of GTSRB. Our work can be a good prototype for people who will be working with this traffic sign identification problem in future. We have tried to document all the problems we have faced and also where we have been successful. We have found this project very interesting and learnt a lot about working with machine learning tools which will surely help us in future.

# Appendix

To run our project successfully, the following requirements must be fulfilled –

* Storage: At least 4GB free memory in the hard disk.
* Graphics: Minimum 6GB dedicated nVidia GPU.
* Python: version 3.x
* Keras: version 2.x

# Execution instructions

Before executing the program, it is required to download ‘full\_yolo\_backend.h5’ from the following Google drive link and place it to the ‘lib’ folder.