

Deep Learning Implementation of Road Sign Detection and Recognition

Author: Spyros Lontos

Student Number: C1722325



CM3203

One Semester Individual Project

Module Credits: 40

Supervisor: Dr. Yukun Lai

Moderator: Michael Daley

May 2020

Abstract

Road sign detection and recognition through the use of computer vision and image processing techniques have been a prevalent subject over the past few decades. Traffic recognition systems can be trained with the use of Deep Convolutional Neural Networks to learn and predict these predefined traffic signs. Modern cars implemented this idea as an advanced driver-assistance feature. Mainly being driven by their use on autonomous and self-driving vehicles this futuristic application had been greatly advancing. As deep neural network techniques are constantly being refined the capabilities of this technology have developed immensely.

The aim of this project is to use deep learning techniques with image processing to create a prototype that is able to locate the presence of traffic road signs and distinguishing their classification. With the use of TensorFlow's Python API, I adapted a processed version of The German Traffic Sign Recognition Benchmark (GTSRB) giving me the ability to train different pre-trained models and test their overall performance given a variety of hyperparameter changes and methods.

By conducting the different experiments, I have concluded that the SSD MobileNet V2 model is sufficient enough to be used in real-time applications for road sign detection. With an average accuracy of 80% with a substantially high processing speed providing over 30 Frames Per Second it is suitable to be used in a live environment. Some of the other methods which I had used such as dropout and switching to a new pre-trained model may not have had provided any better results, but the additional evidence supports the use of the faster SSD MobileNet model.

Acknowledgments

I would like to express sincere gratitude to my Supervisor, Dr. Yukun Lai, for giving me this amazing opportunity to explore this interesting application of deep learning techniques. His knowledge and guidance were valuable in steering my research in the right direction allowing me to express my full potential on this project. Additionally, I would also want to thank my family and friends for their emotional support and encouragement through this difficult academic period.

Table of Contents

1. Introduction	7
2. Background	10
2.1 Mapillary Traffic Sign Database	10
2.2 German Traffic Sign Recognition Benchmark Dataset	11
2.3 Deep Learning	11
2.4 TensorFlow's Object Detection API	12
2.5 TensorFlow Detection Models	13
2.5.1 SSD MobileNet	13
2.5.2 Faster-RCNN ResNet	14
2.6 Evaluation Metric Functions	15
2.6.1 Intersection over Union	15
2.6.2 Average Precision	16
2.6.3 Localization Loss	17
2.6.4 Classification Loss	17
2.6.5 Regularization Loss	17
2.6.6 Total Loss	17
2.7 Deep Neural Network Overfitting	18
2.8 Google Colab	18
3. Approach and Implementation	19
3.1 Choosing Dataset	19
3.2 Dataset Processing	21
3.3 Environment Setup	24
3.4 Model and config	25
3.5 Training and Tensorboard	29
3.6 Overfitting Prevention	32
3.6.1 Dropout	32
3.6.2 Early Stopping	33
3.7 Exporting	34
3.8 Testing	36

4. Results and Evaluation	39
4.1 Classes (43) vs Classes (13) Performance Difference	40
4.1.1 Comparing TensorBoard Results	40
4.1.2 Checkpoints Extracted	41
4.1.3 Comparing Performance Results	41
4.2 Dropout	42
4.2.1 TensorBoard Results and Choosing Export	42
4.2.2 Performance Results	43
4.3 Faster-RCNN ResNet	43
4.3.1 TensorBoard Results and Choosing Export	43
4.3.2 Performance Results	44
4.4 Environment Difference	44
5. Conclusion of Results	46
6. Future Work	47
6.1 Different Dataset	47
6.2 Hyperparameter Tuning	47
6.3 Adapting Properly Overfitting Prevention Methods	47
6.4 Additional Training Time	47
7. Conclusion	48
8. Reflection	49
9. Appendices	50
10. References	51

Table of Figures

Figure 1 - TensorFlow's Object Detection sample image [4]	8
Figure 2 - YouTube Video displaying the view of Tesla's Autopilot system [6]	9
Figure 3 - MTSD Annotation Example	10
Figure 4 - GTSRB csv annotations example	11
Figure 5 - Layers of an Artificial Neural Network [12]	12
Figure 6 - Single Shot MultiBox Detector architecture [17]	13
Figure 7 - MobileNet Approach [18]	14
Figure 8 - Faster-RCNN Structure [19]	14
Figure 9 - ResNet Architecture [20]	15
Figure 10 - Intersection over Union Diagram [22]	16
Figure 11 - tp, fp, fn Regions [23]	16
Figure 12 - Precision & Recall formulas [24]	16
Figure 13 - Accessing csv file and collecting class instances	20
Figure 14 - GTSRB Frequency Analysis	20
Figure 15 - Renaming, changing Filename, and discarding unused classes	21
Figure 16 - Grouping annotation files	22
Figure 17 - Extracting Test set	22
Figure 18 - Name and ID for classes used	23
Figure 19 - Generate tfrecords commands	23
Figure 20 - Generating Label Map	23
Figure 21 - Files after Processing the Dataset	24
Figure 22 - Proper setup output results	25
Figure 23 - Resolution Metrics Code	26
Figure 24 - Resolution Metrics Results	26
Figure 25 - 50x50 Scaled test images	27
Figure 26 - mAP on object size amongst models [28]	28
Figure 27 - Performance Differences [28]	28
Figure 28 - model_main changes	29
Figure 29 - TensorBoard Visualizations	30
Figure 30 - Loading TensorBoard	30
Figure 31 - TensorBoard's Graphs Tab Example	31
Figure 32 - TensorBoard's Image Tab Example	31
Figure 33 - Neural Network Dropout [30]	32
Figure 34 - Hyperparameters Changed for Dropout	32
Figure 35 - Early Stopping Principle [31]	33
Figure 36 - Early Stopping Implementation	34
Figure 37 - Evaluation Metrics Example	34
Figure 38 - Total_loss & mAP Graphs	35
Figure 39 - Exporting Script Command with Parameters	35
Figure 40 - ImageDetection Notebook Cell Examples	36

Figure 41 - Fixed Image Resizing	36
Figure 42 - Faster-RCNN ResNet Configuration Image Resizer	37
Figure 43 - Resizing Image, Keeping Aspect Ratio with Capped Values (Code)	37
Figure 44 - Road Sign Detection and Visualization	38
Figure 45 - Model's Evaluation with the use of Test set	39
Figure 46 – 43-Class Evaluation Metrics Graphs	40
Figure 47 – 13-Class Evaluation Metrics Graphs	41
Figure 48 - TensorBoard Results on Model with Dropout	42
Figure 49 - TensorBoard Metric Graphs on Faster-RCNN ResNet	43
Figure 50 - Faster-RCNN ResNet Evaluation Result	44
Figure 51 – ‘Global Sec’ Metric on Laptop	44
Figure 52 – ‘Global Sec’ Metric on Google Colab	45

Table of Abbreviations

GTSRB	German Traffic Sign Recognition Benchmark
MTSD	Mapillary Traffic Sign Database
ML	Machine Learning
TSR	Traffic-Sign Recognition
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
SSD	Single Shot MultiBox Detector
RCNN	Region-Based Convolutional Neural Network
RPN	Region Proposal Network
VOC	Visual Object Classes
AP	Average Precision
mAP	mean Average Precision
FPS	Frames Per Second
IoU	Intersection over Union
tp	True positives
tn	True negatives
fp	False positives
fn	False negatives
IDE	Integrated Development Environment
API	Application Programming Interface
Json	JavaScript Object Notation
ppm	Portable Pixmap
csv	Comma Separated Values

1. Introduction

The development of cars made a huge impact in the way we live, as this method gave the ability to each person to travel large distances. Cars were introduced as expensive luxury thus not a lot of people could afford them. With the establishment of a large number of mass production lines within the automotive industries, it caused the inflated prices of the product to drop significantly. Cars then changed quickly from just a luxury to an everyday necessity. As the number of vehicles in the road rapidly increased so did the number of accidents, having an immediate effect on the number of deaths. From the early 1900s there was a linear and steady increase to motor vehicle deaths recorded in the US [1]. It had been clear that preventative measures needed to be established to stop this rising issue.

A big step in reducing the number of accidents was with the introduction of road traffic signage. They aimed to establish rules, inform drivers of what is to be expected and, to keep a smooth flow of traffic. Despite that, the negligence of these road signs due to distracted driving still played a big problem in the number of traffic accidents.

With an average of over 16000 car accidents per day in the US alone in 2020 [2], many of them resulting to significant injury or death, it is clear that these preventative measures that have been made over the years did not result in the complete reduction of motor vehicle accidents. Many factors can cause road accidents. Drunk driving, speeding, other distractions but it mostly lies on the big problem which is human error.

To improve the driving experience and help reduce the number of accidents due to distractions a relatively new technology was developed. Traffic-Sign Recognition or TSR, is a technology that first appeared in 2008 [3]. It was developed to use image processing techniques to be able to detect and distinguish the road signs found during a normal car route. The applications of this technology were mostly aimed as an autonomous driving feature or a driver-assistance feature which would alert the driver about the road sign. With the use of Convolutional Neural Networks (CNN), companies had the ability to train artificial models to predict road signs observed from attached cameras on motor vehicles.

The application of TSR was built on the technology of Object Detection. The combination of computer vision, image processing and, deep learning techniques could create Deep Convolutional Neural Networks that learn to distinguish a very large number of typical objects. By taking advantage of the general features that make up these objects we could use machine learning-based techniques that could detect and classify the different objects of interest. Using object detection deep learning approaches such as region-based convolutional networks Faster-RCNN and the Single Shot MultiBox Detector (SSD) we could use pre-trained models to both detect and recognize different classes present within an image. As seen in **Figure 1** object detection can distinguish the location and type of objects such as a bicycle, car, and dog. It visualizes the detections by drawing a bounding box around the predicted object as well as giving it an estimated classification name.

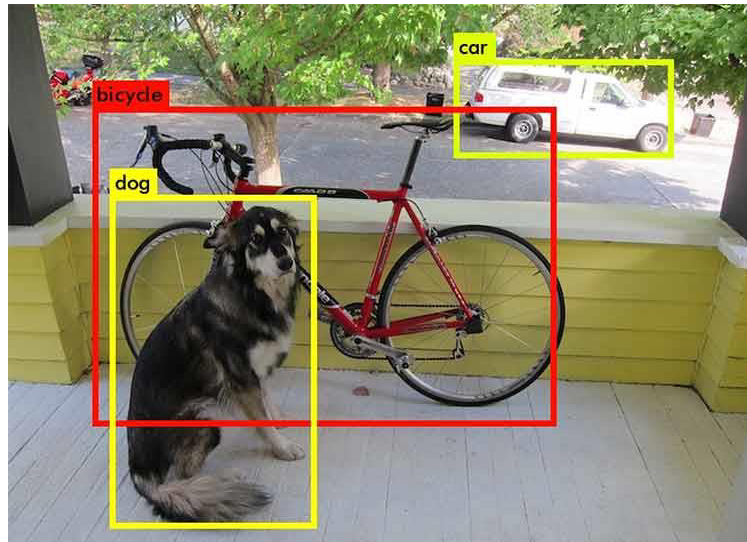


Figure 1 - TensorFlow's Object Detection sample image [4]

Open-source platforms such as TensorFlow developed the Object Detection technology further, by providing clear documentation and a wide variety of functionality features they gave the ability to every person to test, manipulate and learn the applications of such an interesting idea. Additionally, their Python-based Object detection API provides the ability to train pre-trained models using new undefined classes for every specific application. For Traffic Sign detection, we can provide the training model with a large dataset of road sign images which will then learn to recognize and detect on provided image footage.

By correctly processing a large dataset of images I was able to test road sign recognition and detection using various pre-trained models. Each model had its benefits and drawbacks but finding a more appropriate one in the real-time processing application was key. Additionally, with the use of some overfitting prevention methods I tried tackling a problem that is prevalent in deep learning which has significant negative effects on the whole process.

I show how I processed the large GTSRB image dataset to a file format that would be used in TensorFlow's training process. Justifying the change made, in converting the ppm images to png which is the supported image format, and also trimming the dataset to a more manageable size which would improve both performance and relative training time. Additionally, I present the general performance in terms of accuracy and speed on models with minor differences. This would provide me clear results on how the changes made throughout the implementation phase proved to be beneficial.

Although road sign detection and recognition can be used to aid in driving experience it is also being used as an essential component in autonomous driving. Sophisticated systems such as Tesla's autopilot shown by the video in **Figure 2**, they have shown a substantial decrease in the number of accidents with their use. With 1 accident for every 3.07 million miles whilst using Autopilot, it indicates a huge difference whilst comparing it with the 1 accident for every 479 thousand miles a typical automobile has [5]. These statistics reveal that Tesla's Autopilot is up to 6 times safer than a typical human driver.



Figure 2 - YouTube Video displaying the view of Tesla's Autopilot system [6]

2. Background

Adapting deep learning techniques to road sign detection which can be used in real-time processing applications can be challenging. This section focuses on providing context for the technologies, deep learning models, and databases that I have chosen to use throughout my study. With these technologies I was able to create a high performing model that is capable of detecting road signs from an extracted set of test images and is also quick enough that can be used in a real-time processing application.

2.1 Mapillary Traffic Sign Database

The Mapillary Traffic Sign Database (MTSD), is an assortment of high-resolution street-level imagery [7]. Featuring over 100,000 images covering more than 300 classes from all 6 continents makes it the biggest and most diverse road sign dataset in the world. The wide variety of varying image conditions such as weather, season, or time of day makes it a more diverse dataset.

Additionally, more than half of the provided images have JavaScript Object Notation (json) styled annotations. These annotations can provide a plethora of details for each image such as the width, height, and a list of present road signs within the image. Such a list has a wide variety of variables (example shown in **Figure 4**) for each road sign but the most important ones being the bounding box coordinates 'bbox' that indicates its actual position on the image and its label 'label'.

```
{
  "width": 3984,
  "objects": [
    {
      "properties": {
        "ambiguous": false,
        "dummy": false,
        "direction-or-information": true,
        "barrier": false,
        "occluded": false,
        "out-of-frame": false,
        "exterior": false,
        "included": false,
        "highway": false
      },
      "bbox": {
        "xmin": 868.58203125,
        "ymin": 1386.7646484375,
        "ymax": 1467.0087890625,
        "xmax": 960.01171875
      },
      "key": "3aIqzDndSDq2sSUXtCKm7w",
      "label": "other-sign"
    }
  ],
  "ispano": false,
  "height": 2988
}
```

Figure 3 - MTSD Annotation Example

2.2 German Traffic Sign Recognition Benchmark Dataset

With over 50,000 images and more than 43 classes the lifelike multi-class German Traffic Sign Recognition Benchmark (GTSRB) dataset [8] still provides a wide road sign image variety. The GTSRB dataset [9] contains low-resolution Portable Pixmap (ppm) format images, varying from 15x15 to 250x250 pixel images.

With the basic structure of the dataset containing one directory per class, each directory containing one comma-separated values (csv) file with annotations as shown in **Figure 4**. This information contains the image resolution, the road sign bounding box, and also its classId. Additionally, the training images are grouped by tracks, each track containing 30 images with varying resolutions, for a single physical traffic sign. The dataset can be accessed from a public archive [10] which is split into different forms such as the official train data and test data.

	A
1	Filename;Width;Height;Roi.X1;Roi.Y1;Roi.X2;Roi.Y2;ClassId
2	00000_00000.ppm;29;30;5;6;24;25;0
3	00000_00001.ppm;30;30;5;5;25;25;0
4	00000_00002.ppm;30;30;5;5;25;25;0
5	00000_00003.ppm;31;31;5;5;26;26;0
6	00000_00004.ppm;30;32;5;6;25;26;0
7	00000_00005.ppm;31;31;6;6;26;26;0
8	00000_00006.ppm;33;34;6;6;28;28;0

Figure 4 - GTSRB csv annotations example

2.3 Deep Learning

Major advancements in modern computing technologies allowed Artificial Intelligence to revolutionize the field of computer science. The use of machine intelligence trying to emulate human intelligence in propositioned tasks has been on the rise over the years. Artificial Intelligence is split amongst a number of different techniques methods such as Machine Learning, Computer Vision, and Robotics.

Machine learning or ML is the general study of implementing algorithms that can improve automatically using past experiences. By defining specific features, the ML model can learn to make predictions and decisions without being explicitly programmed to do so. There are a plethora of applications this technology has already being implemented in [11] such as, though Virtual Personal Assistants like SIRI, or Alexa. Within Social media services like automatic face detection on uploaded Facebook images, or even Amazon's product suggestion. The performance of the implementation of these applications is being constantly refined, as more data are being fed to these machine learning systems better, and more accurate estimates will be provided.

A big subset of Machine Learning is Deep learning. Unlike ML, deep learning does not require any human intervention. It does not need the definition of explicit features to be able to make accurate estimates. In fact, its method bases on the use of Artificial Neural Network (ANN), where the multi-layered approach can progressively extract features from the given input.

The general overview of such ANN is displayed in **Figure 5**. Firstly, the input layer, is highly dependent on the application and the data it is provided. For road sign detection application which the input will be a large set of images the raw input which will be used will be a matrix of pixels.

After the input layer there are the hidden layers. Information is being transferred from each layer to the next over connecting channels. Each channel has a value attached to it which we refer to as the **Weight**. All neurons have a unique number associated with it called **Bias**. This Bias is added to the weighted sum of inputs reaching the neuron, which is then applied to a function known as the activation function. The result of the activation function determines if the neuron gets activated. Every activated neuron passes on information to the following layers which continue until the last layer.

The output layer is the final prediction the ANN will choose, the one neuron activated in the output layer corresponds to the prediction. The weights and biases are continuously adjusted to create a well-trained network.

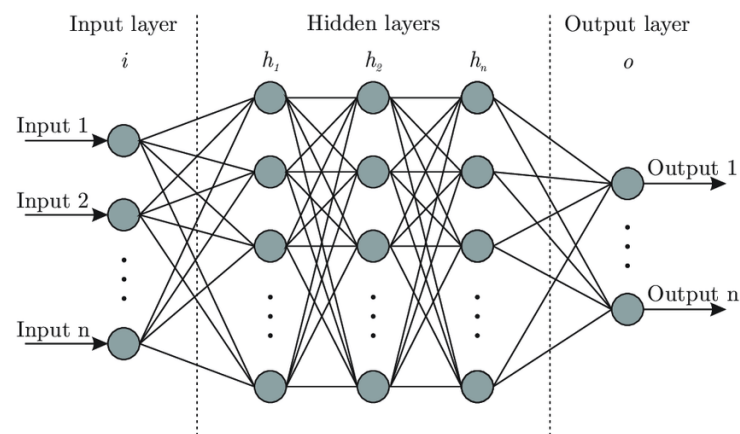


Figure 5 - Layers of an Artificial Neural Network [12]

The drawbacks of deep learning are the amount of time it takes to train a Neural Network. It requires a substantially large dataset of input images and a lot of computational power so it can be computationally expensive to run.

2.4 TensorFlow's Object Detection API

TensorFlow is described as an end-to-end machine learning platform. Version 1.0.0 was released early 2017 [13], it gave the ability for researchers to explore several different topics using machine learning applications using its free open-source library. The constant expansion of this library with tools and extensions as well as the abundance of documentation that has been generated over the years makes it able for novices to easily learn and use such systems to its full capabilities.

TensorFlow has a relatively large library based on Object Detection. Their GitHub repository [14] provides a variety of tools and scripts which can be used to effectively train using a variety of machine learning models a custom dataset. It can also be used to test general object detection with several provided pre-trained models.

2.5 TensorFlow Detection Models

TensorFlow has a long collection of pre-trained detection models [15]. They have been trained on the COCO dataset, the Open Images Dataset, and the Kitti dataset and they can provide the immediate usability of object detection and recognition on a variety of objects. These models can be used as an initializer during training on a custom dataset which they would greatly improve the amount of time it would take from training the model from scratch.

Each model has a certain degree of accuracy as well as its average processing speed. Some models have been to a very high number of accuracy percentages thus making relatively slow. Whilst others have been created with the purpose of being used in real-time processing applications where speed would be a very important aspect, but they may lack in the performance of their overall accuracy. Most of these models use a combination of detector algorithms with different optimization methods to maximize their performance.

2.5.1 SSD MobileNet

The Single Shot MultiBox Detector (SSD) algorithm was created with a new approach that tried to reduce the number of steps needed to detect objects within an image. Instead of using typical standard Region Proposal Network-based approaches that required 2 steps, this technique made it able to perform both object localization and classification in a single path of the CNN. It greatly increased speed performance with substantially good mean average precision (mAP) results. With SSD300 achieving 74.3% at 59 frames per second (FPS), its outperformed R-CNN with 73.2% at just 7FPS [16]. With the general overview Architecture of the SSD deep convolutional network shown in **Figure 6**.

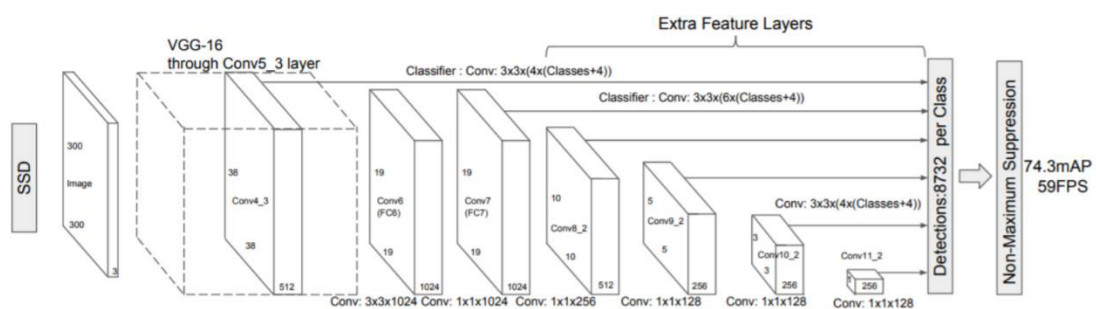


Figure 6 - Single Shot MultiBox Detector architecture [17]

The MobileNet convolutional transformation and modification methods were also aimed at having high speed. It was developed for the purpose of being implemented into computationally limited platforms whilst still providing solid accuracy performance. The method of implementation factorized standard convolution into depthwise convolution and a pointwise convolution [18] as displayed in **Figure 7**. Using this method, it has been observed that MobileNet has an 8x reduction in computational cost when compared to the standard convolution method.

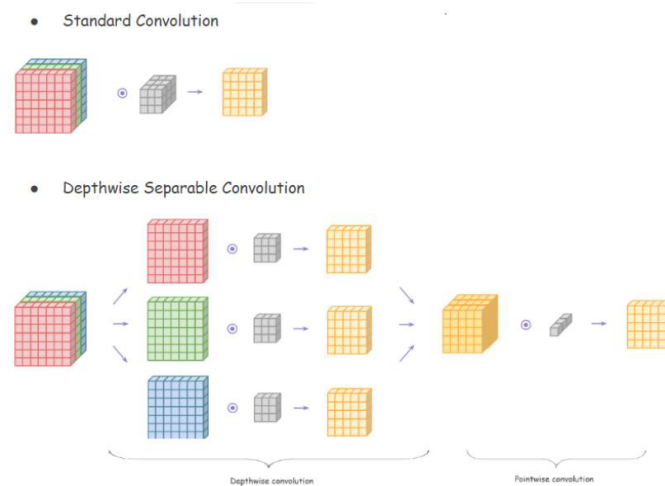


Figure 7 - MobileNet Approach [18]

2.5.2 Faster-RCNN ResNet

The Region-Based Convolutional Neural Network (RCNN) uses a 2-step approach to perform object detection. The first step is responsible for generating the region proposals and the 2nd step for detecting the object for each proposal.

Region proposals are found by the CNN which is referred to as the Region Proposal Network (RPN). It slides on the last feature map of the convolutional layers and predicts whether an object is present while guessing its bounding boxes (shown in red on **Figure 8**).

With those region proposals as input a Fully connected neural network, the Fast-RCNN will predict a classification for the input, shown in blue in **Figure 8**.

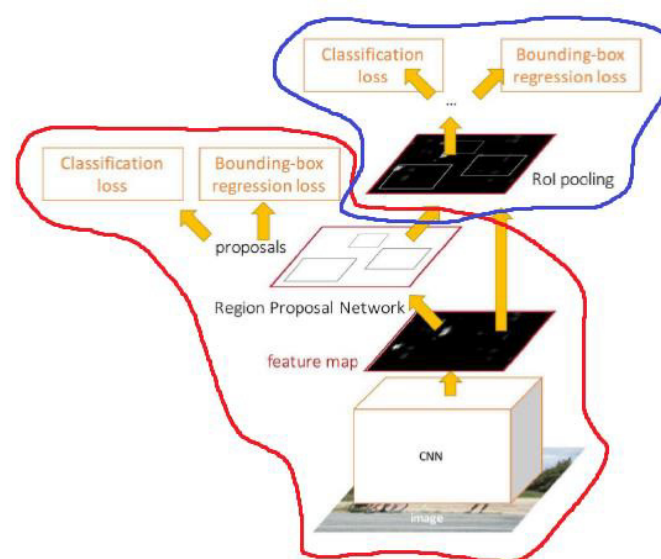


Figure 8 - Faster-RCNN Structure [19]

The ResNet method bases its method on an optimization problem that states, “deeper models are harder to optimize”. A deeper model should be able to perform at least as well as the shallower model. “Residual blocks enables the network to preserve what it had learnt previously (if there’s nothing to learn) by having an identity mapping weight function where the output is equal to the input, preserving what the neural network has learnt by not applying diminishing transformations or if the layer is able to learn something it’ll add on to what the network has learnt.”[20] The architecture of the ResNet optimization method is shown in **Figure 9**.

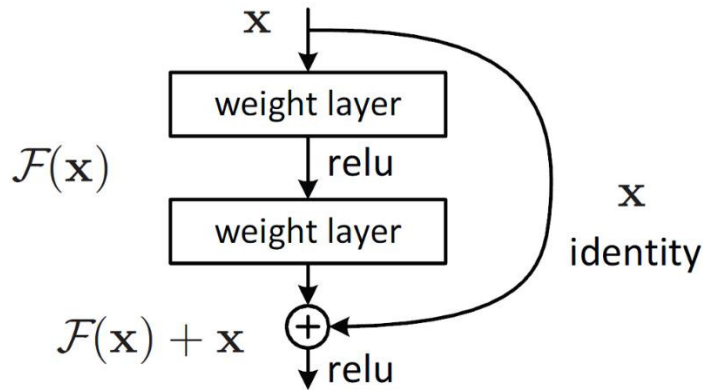


Figure 9 - ResNet Architecture [20]

2.6 Evaluation Metric Functions

Different evaluation metrics are used to observe current the fitness of training models. It allows for proper evaluation through the whole process, and it is a general indicator of how the model performs on the provided dataset.

Popular competitions and metrics, such as the Pascal Visual Object Classes (VOC) challenge, the COCO object detection challenge, and the Open Images challenge all provide different important metrics and principles that are used to evaluate object detection models. [21]

The most important evaluation metrics which would be used are the ‘total loss’ which shows a general effect on the overall fitness of the model and the mAP which indicates the overall accuracies for all of the used classes.

2.6.1 Intersection over Union

Intersection over Union or IoU is a way to measure the accuracy of a predicted bounding box with the ground-truth box. By measuring the area of overlap between the bounding box and the ground-truth box and compare it with their combined total area (shown in **Figure 10**), we can get a value between 0 and 1 which can indicate how good the predicted detection was.

IoU can be further used to determine whether predictions made where either True Positives (tp), True Negatives (tn), False Positives (fp), or False Negatives (fn) described by **Figure 11**, by comparing the IoU results to specific threshold values.

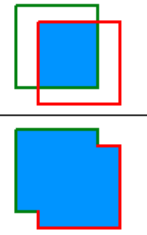
$$IOU = \frac{\text{area of overlap}}{\text{area of union}} = \frac{\text{area of overlap}}{\text{area of union}}$$


Figure 10 - Intersection over Union Diagram [22]

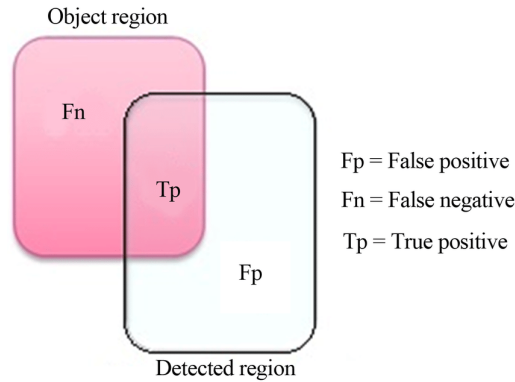


Figure 11 - tp, fp, fn Regions [23]

2.6.2 Average Precision

Precision and Recall are both used to determine the Average Precision (AP) of a class. Where precision is used to measure the accuracy of the model's predictions and recall measures how good the detection of all the positives was by comparing them to the sum of the predicted ground-truth boxes. Both of their equations are displayed in **Figure 12**.

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

Figure 12 - Precision & Recall formulas [24]

To find the average precision, we will first need to find the Precision-Recall curve; this indicates the association between these metrics at different threshold values. To calculate AP, we take the average precision values across all the unique recall levels.

With AP we can observe the fitness for each class we use to train our Neural Network. In order to perceive the fitness amongst all of the classes we will take their average called mean average precision (mAP).

2.6.3 Localization Loss

Localization loss is used to provide an estimation value to the detection aspect of the object detection process. By comparing the predicted bounding box, it will use L2 distance and compare it with the ground truth.

2.6.4 Classification Loss

Classification loss is used to provide an estimated fitness level on the performance of the model on how well it performs in the recognition aspect of object detection. By using the predicted bounding box and comparing it with the ground truth an IoU value will be generated. If this IoU value exceeds a threshold (e.g. 0.5) and the predicted class matches the actual class of the object it will be counted as a correct prediction.

2.6.5 Regularization Loss

The regularization loss punishes the model for being too complex. Since the goal is generalizing the predictions of the model it provides the ability to detect and recognize trained objects on images with varying conditions. On the problem of road signs, it should be able to detect and recognize them in different weather conditions, lighting conditions, and on different resolution images. The model should not be only fitted to the training data that would provide. Regularization loss shows how the model learns to generalize the extracted features making it more capable in predicting accurately on new unseen data.

2.6.6 Total Loss

This metric is the sum of all 3 previously described metrics. It combines the values of localization loss, classification loss, and regularization loss and provides a general description of the overall performance amongst all 3. During training a steadily decreasing total loss shows clearly that the overall fitness of the Neural Network is correctly adapting to the provided dataset.

2.7 Deep Neural Network Overfitting

Leaving the Deep Neural Network to keep training as much as possible may have negative effects on its performance. The neural network becomes too closely fitted to the training set where it loses generalization and predictability in new unseen data. Several methods have been implemented to prevent deep neural networks in reaching this point.

To be able to observe if the model has reached the point of overfitting during training a common method of splitting the whole dataset into 3 smaller sets is used. The train set, the evaluation set, and the test set. The train set is used to train the actual model and the evaluation set is used to cross-validate its performance. This cross-validation is a significant indicator of presenting the actual fitness of the model. Our goal would be to try to minimize the error rate between the training and evaluation set. Additionally, with the use of the test set, it allows us to obtain an unbiased opinion on how well the model performs on an unseen set of images.

2.8 Google Colab

Google Colaboratory is a free browser-based platform that allows anyone with a google account to set up an environment and executes “arbitrary python code” [25]. It is a notebook styled environment similar to Jupyter, with much more available resources. It has the option of adding either CPU (Central Processing Unit), GPU (Graphics Processing Unit), or TPU (Tensor Processing Unit) processing acceleration in the environment’s resources which can provide a substantial improvement in computer-intensive processes such as machine learning

The limitations of this Cloud Based environment include the memory and a maximum Virtual Machine runtime. But, its overall ease of use, the adaptation of both Google Drive and GitHub, makes managing code and backing up much easier.

3. Approach and Implementation

My general approach in creating a structured solution for this specific application was to first dissect it into multiple sections. This modular perspective helped me focus on and improve each specific aspect during the implementation process making it easier to adapt to any unforeseen changes.

3.1 Choosing Dataset

The image datasets that I would expect to use should contain a very high number of high-resolution images, split evenly amongst several classes. The high-resolution images should give me a better overall performance when testing on different test images and the even spread amongst classes should provide an overall balanced model that performs well to all of the trained classes.

After requesting the Research Edition of the MTSD I received a confirmation email that gave me access to the dataset. Only after downloading the 50GB file containing the images, annotation files, and the split text files, I realized the scale of this dataset. The memory on my computer was relatively low so it would have been unreasonable to use such big of a dataset for my project. Additionally, by the dataset having such a vast number of classes it meant training a model to perform well on all of them would take a substantial amount of time. All in all, the dataset was not suited for the scale of my project, so I had to find a more reasonable one.

After discussing with my supervisor, I was directed to The German Traffic Sign Recognition Benchmark [9]. Its relative scale was big enough that it could train a relatively accurate model and the low-resolution images meant that memory was not going to be an issue. With a much reasonable number of 43 classes it seemed more logical to go with the approach of using this dataset.

To observe the dataset's image Frequency, I collected the number of images for each class in both the train set and evaluation set. By iteratively going through the image annotation file I was storing the class instance for each image in a dictionary, sorting them according to the number of images and finally storing them in a csv file in which they can be visualized. (Code is shown in **Figure 13**).

```

# Reading csv file and iteratively storing class instances
with open('train_labels.csv', 'r') as currentfile:
    reader = csv.reader(currentfile)
    next(reader)
    for row in reader:
        if row[3] not in classFrequencyTrain:
            classFrequencyTrain[row[3]] = 1
        else:
            classFrequencyTrain[row[3]] += 1

# Reading csv file and iteratively storing class instances
with open('eval_labels.csv', 'r') as currentfile:
    reader = csv.reader(currentfile)
    next(reader)
    for row in reader:
        if row[3] not in classFrequencyEval:
            classFrequencyEval[row[3]] = 1
        else:
            classFrequencyEval[row[3]] += 1

# Combining the 2 dictionaries used with the same key holding a tuple with both number of Images in the Train Set
# and the number of images in the Evaluation set as its key
classFrequencies = dict([(k, [classFrequencyTrain[k], classFrequencyEval[k]]) for k in classFrequencyTrain])

# Sorting the dictionary by number of images
for i in sorted(classFrequencies.items(), key=lambda x: x[1], reverse=True):
    value = i[0], i[1][0], i[1][1]
    csv_list.append(value)

# Creating a csv file storing the results which can be then visualized
column_name = ['class', 'numTrainImages', 'numEvalImages']
csv_df = pd.DataFrame(csv_list, columns=column_name)
csv_df.to_csv('FrequencyAnalysis.csv', index=None)

```

Figure 13 - Accessing csv file and collecting class instances

By plotting the collected results as shown in **Figure 14**, I saw a great unevenness in how the images were spread amongst the classes. With the biggest “Speed limit (50km/h)” class having more than 2200 in the train set and the lowest being “Go straight or left” class with just 220 images. This indicated that the performance for each class was going to vary greatly during training and testing. To overcome this issue, I decided to trim the dataset to a smaller number of classes. I chose to keep the first 13 with the highest number of overall images. They all exceeded 1400 images in their train set and 450 images in their evaluation set. By using just these 13 classes I should have been able, in theory, get better overall results in a much shorter time, as fitting the model to the dataset would have been much more efficient.

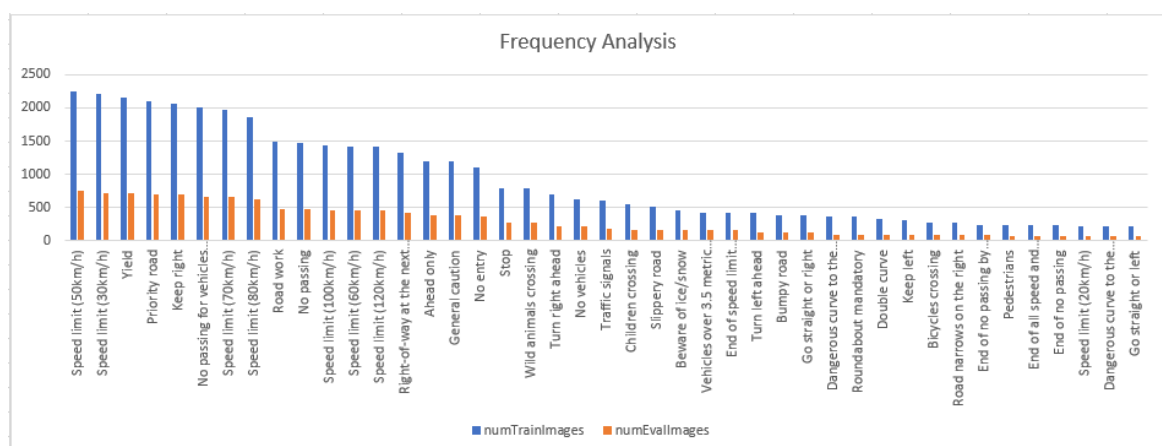


Figure 14 - GTSRB Frequency Analysis

3.2 Dataset Processing

The TensorFlow Object Detection library expects the dataset information to be in the form of a TensorFlow record (tfrecord). I will need to find a way to adapt all the information of each image such as the filename, width, height, and class, and transform them into the TensorFlow records.

To process the dataset to generate the appropriate files that I would need for the training process I had to split it into multiple steps. By using the public archives that store the GTSRB dataset [10] I chose to download 3 different zipped files that I could use to generate the training set, the evaluation set, and the test set as well as the image annotations. The 3 .zip files that I used were, “GTSRB_Final_Training_Images”, “GTSRB_Final_Test_Images” but because the .csv file within this zipped file did not contain information for the classes for each image I had to also download the “GTSRB_Final_Test_GT” and use that csv.

To process my dataset, I used multiple python scripts that I had created.

The “train_csv_generator.py” discards all the class directories which are not going to be used and keeps only the 13 classes. Additionally, it extracts every image from their class directories into the main folder whilst also performing several additional actions (as shown in **Figure 15**). It renames each filename by adding their class ID to the front because files have the same names throughout the class directories. Also, it converts them from their ‘ppm’ extension to a .png extension because the ppm format was is not supported within TensorFlow’s object detection API. I had decided to use the ‘png’ format since it has lossless compression and it would not introduce any artifacts that could impact the learning process.

```
# Converting the images to pngs and grouping them in the same folder
# Grouping all of the csvs in the same folder
for folder in os.listdir(os.getcwd()):
    if folder in folderIDList:
        os.chdir(folder)
        # Change the images from .ppm to .png and rename by adding their classId in the beginning
        # (renaming is necessary due to filenames being the same in the different class folders
        for ppmfile in glob.glob('*.ppm'):
            i = cv2.imread(ppmfile)
            os.chdir('..')
            cv2.imwrite(folder+'_'+ppmfile.split('.')[0]+'.png', i)
            os.chdir(folder)
            os.remove(ppmfile)

        # Grouping all of the csvs in the same folder
        for csvfile in glob.glob('*.csv'):
            shutil.move(csvfile, parentDirectory)
            print('Moved: ', csvfile)

        os.chdir('..')
        os.removedirs(folder)
        print('Removed directory: ', folder)
    else:
        shutil.rmtree(folder)
        print('Removed directory: ', folder)
```

Figure 15 - Renaming, changing Filename, and discarding unused classes

Finally, the python script groups and processes all the csv files that contain annotation information for each image into a new one named “train_labels.csv” whilst also shuffling and mixing their stored order. Shuffling the order, prevents the model from overfitting to the first observed group of images which would all belong in the same class. (Code in Figure 16)

```
csv_list = []
# Retrieves the information for each image from the csv file
# and stores the image names which we will discard as they are not part of the 13 classes we want
for oldfile in glob.glob('*.csv'):
    with open(oldfile) as csvfile:
        readCSV = csv.reader(csvfile, delimiter=',')
        next(readCSV)

        for row in readCSV:
            classID = row[7].zfill(5)
            firstPart = row[0].split('.')[0]
            filename = classID + '_' + firstPart + '.png'
            width = row[1]
            height = row[2]
            label = labelMaker(row[7])
            ymin = row[4]
            xmin = row[3]
            ymax = row[6]
            xmax = row[5]

            # print(filename, width, height, label, ymin, xmin, ymax, xmax)
            value = filename, width, height, label, ymin, xmin, ymax, xmax
            csv_list.append(value)

    os.remove(oldfile)

# Shuffle the list
random.shuffle(csv_list)
os.chdir('..')

column_name = ['filename', 'width', 'height', 'class', 'ymin', 'xmin', 'ymax', 'xmax']
csv_df = pd.DataFrame(csv_list, columns=column_name)
csv_df.to_csv('train_labels.csv', index=None)
print('Successfully merged the CSVs and created the train_labels csv')
```

Figure 16 - Grouping annotation files

The “eval_test_csv_generator.py” script has very similar functionalities as the previously described script whilst also extracting 150 images the evaluation set and storing them into a newly created test set (shown in Figure 17). The images in the test set were never going to be seen during the training and evaluation process making the testing process less biased. With the completion of the scripts 2 folders containing the evaluation and test set having multiple .png images. Also, 2 .csv files called ‘test_labels.csv’ and ‘eval_labels.csv’ were created which contain annotation information for each image.

```
# Shuffle the list
random.shuffle(eval_csv_list)

# Randomly picks 150 images and moves them to the test folder
# These images will not be used in training
test_csv_list = []
destination = os.path.join(parentDirectory, 'test')
for i in range(150):
    choice = random.choice(eval_csv_list)
    test_csv_list.append(choice)
    shutil.move(choice[0], destination)
    eval_csv_list.remove(choice)
```

Figure 17 - Extracting Test set

To generate the 2 TensorFlow record files I used pre-existing code [26] that I had found online, but modified it to use '.png' files instead of .jpg files and also changed the "class_text_to_int(row_label)" function to fit my specific 13 classes with their appropriate name and ID (**Figure 18**).

```
def class_text_to_int(row_label):
    if row_label == 'Speed limit (30km/h)':
        return 1
    if row_label == 'Speed limit (50km/h)':
        return 2
    if row_label == 'Speed limit (60km/h)':
        return 3
    if row_label == 'Speed limit (70km/h)':
        return 4
    if row_label == 'Speed limit (80km/h)':
        return 5
    if row_label == 'Speed limit (100km/h)':
        return 6
    if row_label == 'Speed limit (120km/h)':
        return 7
    if row_label == 'No passing':
        return 8
    if row_label == 'No passing for vehicles over 3.5 metric tons':
        return 9
    if row_label == 'Priority road':
        return 10
    if row_label == 'Yield':
        return 11
    if row_label == 'Road work':
        return 12
    if row_label == 'Keep right':
        return 13
    else:
        return 0
```

Figure 18 - Name and ID for classes used

By executing the following python commands (**Figure 19**), I was returned with the 2 .record files containing both image and image annotation information.

```
# Create train data:
python generate_tfrecord.py --csv_input=train_labels.csv --image_dir=train --output_path=train.record

# Create test data:
python generate_tfrecord.py --csv_input=eval_labels.csv --image_dir=eval --output_path=eval.record
```

Figure 19 - Generate tfrecords commands

To generate the label map that would be needed in the pipeline configuration, I first needed to create a labels.csv file that contained the ClassID and the Name for each of the 13 classes. By executing my created python script "label_map_generator.py" as shown in **Figure 20**), it would use that 'labels'.csv' and would generate a "label_map.pbtxt" file.

```
import csv

with open('labels.csv', 'r') as currentfile:
    reader = csv.reader(currentfile)
    next(reader)
    with open('label_map.pbtxt', 'w') as label_map:
        for i, row in enumerate(reader, start=1):
            label_map.write('item {\n')
            label_map.write('  id: '\n')
            label_map.write(str(i)\n')
            label_map.write('  name: '\n')
            label_map.write(''+row[1]+''\n')
            label_map.write('}\n')
            label_map.write('\n')

print('Successfully created the label map')
```

Figure 20 - Generating Label Map

A [link](#) to my GitHub repository provides every python file that I have used for processing the dataset as well as the labels.csv file. By correctly executing the processing scripts the directory now contained the files and folders shown in **Figure 21** which were needed for the model training process.

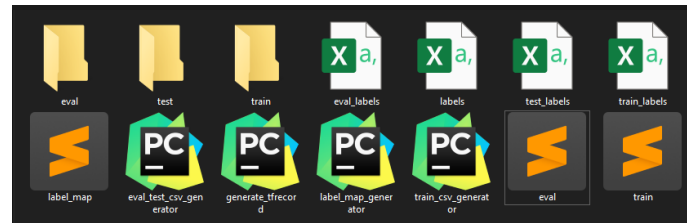


Figure 21 - Files after Processing the Dataset

3.3 Environment Setup

I expect by following TensorFlow's installation procedures [27] to be able to download any dependencies on my laptop and I should be able to set up an environment where I can work on the comfort of my personal space. The specifications of my laptop having 8GB of RAM and a mobile GPU (GTX 840M with 2GB of virtual RAM) I should be able to test and run everything.

By installing an Anaconda environment, installing all of the suggested libraries, and following TensorFlow's installation guide I managed to create the Object Detection Environment. The issue that I had during the process was mostly the version mismatch between some of the libraries. I had to downgrade some, such as TensorFlow from version 2.0 to 1.4 because many functionalities were deprecated and could not be properly used within the Object Detection API. As of May 2020, TensorFlow's object detection has yet to be properly upgraded to its latest 2.0 version so much older versions are needed to run everything properly. To test that everything had been implemented correctly I was able to use a modified version of the provided "object_detection_tutorial.ipynb" and observed that everything was running properly.

Although I had set up everything on my laptop I was limited by the capabilities of my hardware. Every time I was trying to test object detection by using some of the pre-trained models, I saw that my computer was struggling a lot, by showing a lot of 'running out of memory' warnings. So, I began searching for any cloud-based alternatives. Several of the options I had found seemed good, but a friend's suggestion guided me to Google Colab. It could sync any project with Google Drive and since it also had GPU acceleration, it also solved my hardware limitation issue.

Setting up the environment in a Google Colab notebook was much easier than on my laptop. I created a new folder in my google drive where I set up the whole environment in. All the libraries needed came pre-installed, I only had to import the ones I wanted. I had to specify the imported TensorFlow version to 1.x because it automatically imported the 2.x. version. To know that everything had been set up correctly I executed "model_builder_test.py" from the 'builders' folder and from its output (**Figure 22**) I could see that all the tests were passed successfully.


```
|python object_detection/builders/model_builder_test.py

WARNING:tensorflow:
The TensorFlow contrib module will not be included in TensorFlow 2.0.
For more information, please see:
* https://github.com/tensorflow/community/blob/master/rfcs/20180907-contrib-sunset.md
* https://github.com/tensorflow/addons
* https://github.com/tensorflow/tfjs (for I/O related ops)
If you depend on functionality not listed there, please file an issue.

Running tests under Python 3.6.9: /usr/bin/python3
[ RUN      ] ModelBuilderTest.test_create_experimental_model
[ OK       ] ModelBuilderTest.test_create_experimental_model
[ RUN      ] ModelBuilderTest.test_create_faster_rcnn_model_from_config_with_example_miner
[ OK       ] ModelBuilderTest.test_create_faster_rcnn_model_from_config_with_example_miner
[ RUN      ] ModelBuilderTest.test_create_faster_rcnn_models_from_config_faster_rcnn_with_matmul
[ OK       ] ModelBuilderTest.test_create_faster_rcnn_models_from_config_faster_rcnn_with_matmul
[ RUN      ] ModelBuilderTest.test_create_faster_rcnn_models_from_config_faster_rcnn_without_matmul
[ OK       ] ModelBuilderTest.test_create_faster_rcnn_models_from_config_faster_rcnn_without_matmul
[ RUN      ] ModelBuilderTest.test_create_faster_rcnn_models_from_config_mask_rcnn_with_matmul
[ OK       ] ModelBuilderTest.test_create_faster_rcnn_models_from_config_mask_rcnn_with_matmul
[ RUN      ] ModelBuilderTest.test_create_faster_rcnn_models_from_config_mask_rcnn_without_matmul
[ OK       ] ModelBuilderTest.test_create_faster_rcnn_models_from_config_mask_rcnn_without_matmul
[ RUN      ] ModelBuilderTest.test_create_rfcn_model_from_config
[ OK       ] ModelBuilderTest.test_create_rfcn_model_from_config
[ RUN      ] ModelBuilderTest.test_create_ssd_fpn_model_from_config
[ OK       ] ModelBuilderTest.test_create_ssd_fpn_model_from_config
[ RUN      ] ModelBuilderTest.test_create_ssd_models_from_config
[ OK       ] ModelBuilderTest.test_create_ssd_models_from_config
[ RUN      ] ModelBuilderTest.test_invalid_faster_rcnn_batchnorm_update
[ OK       ] ModelBuilderTest.test_invalid_faster_rcnn_batchnorm_update
[ RUN      ] ModelBuilderTest.test_invalid_first_stage_nms_iou_threshold
[ OK       ] ModelBuilderTest.test_invalid_first_stage_nms_iou_threshold
[ RUN      ] ModelBuilderTest.test_invalid_model_config_proto
[ OK       ] ModelBuilderTest.test_invalid_model_config_proto
[ RUN      ] ModelBuilderTest.test_invalid_second_stage_batch_size
[ OK       ] ModelBuilderTest.test_invalid_second_stage_batch_size
[ RUN      ] ModelBuilderTest.test_session
[ SKIPPED  ] ModelBuilderTest.test_session
[ RUN      ] ModelBuilderTest.test_unknown_faster_rcnn_feature_extractor
[ OK       ] ModelBuilderTest.test_unknown_faster_rcnn_feature_extractor
[ RUN      ] ModelBuilderTest.test_unknown_meta_architecture
[ OK       ] ModelBuilderTest.test_unknown_meta_architecture
[ RUN      ] ModelBuilderTest.test_unknown_ssd_feature_extractor
[ OK       ] ModelBuilderTest.test_unknown_ssd_feature_extractor
-----
Ran 17 tests in 0.251s
```

Figure 22 - Proper setup output results

3.4 Model and config

The next step after setting up the environment and processing the dataset was the appropriate model and modifying its configuration. Since the application of my projects aims at road-sign detection that can be implemented in real-time processing examples I needed to choose a faster performing model to train the neural network on my image dataset.

As previously expressed TensorFlow provides a wide variety of object detection models within their detection model zoo. Since the metrics they have used are a good representation of their accuracy and speed performance, I could choose a pre-trained model that should give me similar metric performances.

Going through this model zoo I had observed that the **ssd_mobilenet_v2_coco** had a distinctly low speed for the mAP it provided. Since MobileNet and Single Shot Detection were created and optimized on being used on low-end systems the choice of this pre-trained model and configuration was appropriate.

A tar version of the pre-trained model contained in its directory the following files:

- a graph proto
- a checkpoint
- a frozen graph proto with weights baked into the graph as constants
- a config file (pipeline.config) which was used to generate the graph

Also, I needed to find and download its corresponding configuration file (ssd_mobilenet_v2_coco.config) from the samples/config directory which I then modified to my preferences.

To adjust the hyperparameters of the configuration I needed to go through them and change them to more appropriate values. Firstly, I modified the number of classes field to a number of 13, since I used the trimmed version of the GTSRB dataset. I then had to decide which resolution values I should use in the “fixed_shape_resizer” parameter. To find the appropriate values I gathered some resolution metrics using the csv files as shown in **Figure 23** that contained resolution information for all the images. The results that I got (shown in **Figure 24**) indicated that the average resolution was around 50 pixels by 50 pixels.

```
files = ['train_labels.csv', 'eval_labels.csv', 'test_labels.csv']

for file in files:
    with open(file, 'r') as currentfile:
        reader = csv.reader(currentfile)
        next(reader)
        for row in reader:
            widthList.append(int(row[1]))
            heightList.append(int(row[2]))

print("Min Width:", min(widthList))
print("Min Height:", min(heightList))
print('-----')
print("Max Width:", max(widthList))
print("Max Height:", max(heightList))
print('-----')
print("Average Width:", sum(widthList)/len(widthList))
print("Average Height:", sum(heightList)/len(heightList))
```

Figure 23 - Resolution Metrics Code

```
C:\Users\plays\Anaconda3\envs\RoadSig
Min Width: 25
Min Height: 25
-----
Max Width: 243
Max Height: 225
-----
Average Width: 48.98543046357616
Average Height: 48.9242825607064

Process finished with exit code 0
```

Figure 24 - Resolution Metrics Results

It made sense to use these values in the “fixed_shape_resizer” field so it would not oversample or downsample by too much. Although, soon after testing using images scaled to 50 by 50 as shown in **Figure 25** the resulting images were too small for the bounding box and categories to be correctly visualized. It was awkward to use images with that size, so I opted to change the resolution to 150 by 150. Testing with these new resolution sizes scale the image to an okay size where the bounding box and category were easily visible and identifiable.



Figure 25 - 50x50 Scaled test images

I changed the paths of the label map, the evaluation, and the training record files and added the ability to gather evaluation metrics from Pascal VOC, including metrics for each class. These provided a clear indication of the fitness of the training model throughout the training process. Additionally, since I would be using Google Colab with GPU accelerated processing, I could take advantage of the better hardware, so I changed the batch size to 24.

The configuration had some pre-set data augmentation options such as random horizontal flip and SSD random crop which I took advantage of. These data augmentations can have a significant impact in training time, but they help generalize the neural network making it able to recognize images in varying conditions.

All in all, the `ssd_mobilenet_v2` pre-trained model was a very good starting point but I believed it had additional room for improvement. By doing some further research I found this very clear and descriptive article [28] by Jonathan Hui which tested the performance on a wide variety of models. As shown in **Figure 26** found within the article I had observed that SSD MobileNet was one of the poorest performing models on small objects. Since my dataset only uses low-resolution images it made sense to choose and train a different model that I could compare the performances with.

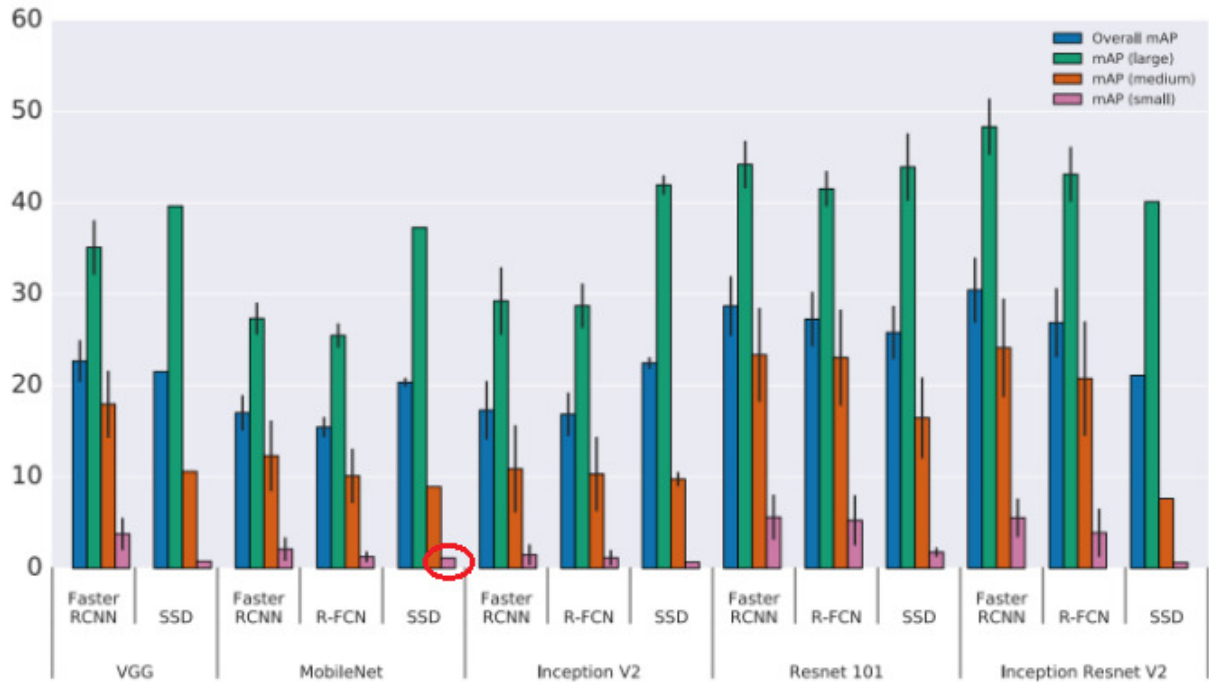


Figure 26 - mAP on object size amongst models [28]

With this information I decided to choose a new accuracy oriented whilst still having an okay speed performance. By using the data given in **Figure 27** I decided to use the Faster-RCNN with ResNet101. Faster-RCNN is not that fast as R-CFN and SSD but by lowering the number of proposals as it was suggested “Faster R-CNN with Resnet can attain similar performance if we restrict the number of proposals to 50.” I was supposed to get a similar performance with the faster models. With this newly chosen model I supposed to get even better than I had whilst using SSD MobileNet without seeing any significant decrease in speed.

Model summary	minival mAP	test-dev mAP
(Fastest) SSD w/MobileNet (Low Resolution)	19.3	18.8
(Fastest) SSD w/Inception V2 (Low Resolution)	22	21.6
(Sweet Spot) Faster R-CNN w/Resnet 101, 100 Proposals	32	31.9
(Sweet Spot) R-FCN w/Resnet 101, 300 Proposals	30.4	30.3
(Most Accurate) Faster R-CNN w/Inception Resnet V2, 300 Proposals	35.7	35.6

Test-dev performance of the “critical” points along our optimality frontier.

Figure 27 - Performance Differences [28]

Again, following the same process, I downloaded the appropriate pre-trained model file and its configuration “faster_rcnn_resnet101_coco.config” file and went through changing the hyperparameters. Since this configuration could maintain image aspect ratio, I decided to again use a minimum value of 150 to account for the resolution of the test images. To maintain the speed as I previously explained, I changed the number of proposals to just 50 from 300. I added the appropriate file paths added the evaluation metrics.

3.5 Training and Tensorboard

Having done all the previous steps, I had all the files needed as well as the environment where I could train my custom model. Since I will be using a pre-trained model as an initializer, it meant that the training process would provide results much faster. Several online tutorials suggested using the training script supplied in TensorFlow's Object Detection repository. Using the 'train.py' script found within the 'legacy' folder, I was able to start the training process and observe the training loss metric and observe how the overall fitness of the model was improving on the provided dataset. Although I could observe this training loss, I was not getting any evaluation metric outputs. With further research, I found out that I needed to run the 'eval.py' found in the same 'legacy' folder concurrently to get an evaluation.

Creating another cell inside the Google Colab notebook and running them concurrently did not perform as expected since it was waiting for one of them to finish to run the other. So, I acted by creating another Google Colab notebook since I could have up to 2 runtimes at the same time. But, by using this method, I encountered the issue of one of the runtimes stalling. It was not getting enough processing time and it kept crashing.

Since evaluation seemed to be the most important aspect of correctly observing the training state of the model, I had to find another way to run both training and evaluation. The solution was the already provided 'model_main.py' which could perform both training and evaluation.

I needed to do some further modifications to it (as shown in **Figure 28**) such as changing the number of logging steps to 10 from the default of 100. Additionally, an important change was on the frequency of checkpoints made. I modified it from creating a checkpoint every 10 minutes to be done every 2500 steps. I observed that after it reached the checkpoint mark it began the evaluation process. This took around 7-8 minutes. So, with the previous configuration after it finished evaluating it began training for just 2 minutes then again created another checkpoint and began evaluating. This issue caused training time to be very short, so it was not really fitting on the training set, but it was instead fitting on the evaluation set. By changing it to a specific number of steps I had stopped the time caused issue.

```
tf.logging.set_verbosity(tf.logging.INFO)

def main(unused_argv):
    flags.mark_flag_as_required('model_dir')
    flags.mark_flag_as_required('pipeline_config_path')
    config = tf.estimator.RunConfig(model_dir=FLAGS.model_dir, log_step_count_steps=10, save_checkpoints_steps=2500)
```

Figure 28 - model_main changes

Since I now had the ability to train and evaluate properly, I needed a way to observe the process which for that I used TensorBoard. TensorBoard is TensorFlow's visualization tool that tracks and visualizes the metrics gathered (Example in **Figure 29**). It can be used to draw graphs to show the performance changes and display image data to show how the model progresses over time.

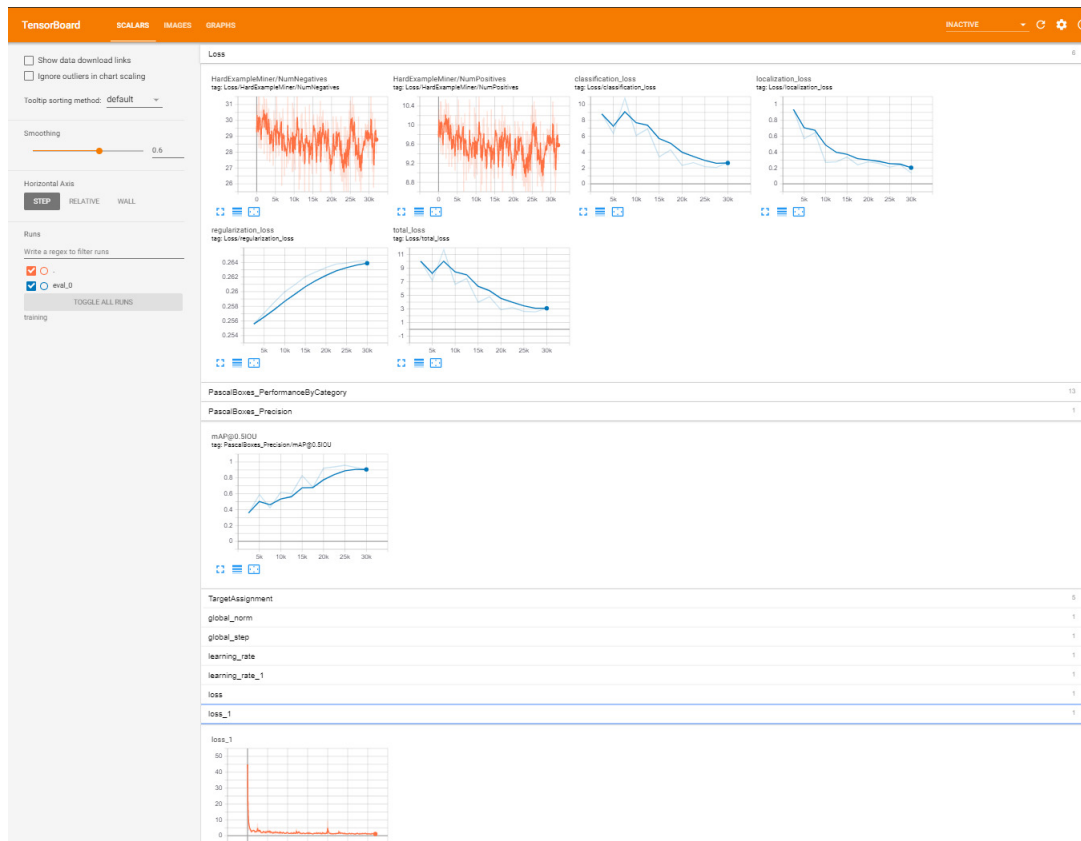


Figure 29 - TensorBoard Visualizations

For that I created a new Google Colab Notebook which I then loaded the TensorFlow library in. I was then able to instantiate TensorBoard by directing it to the training directory which was storing the training files and I was then returned with a page that had all the training and evaluation metrics in the form of graphs, shown by **Figure 30**.

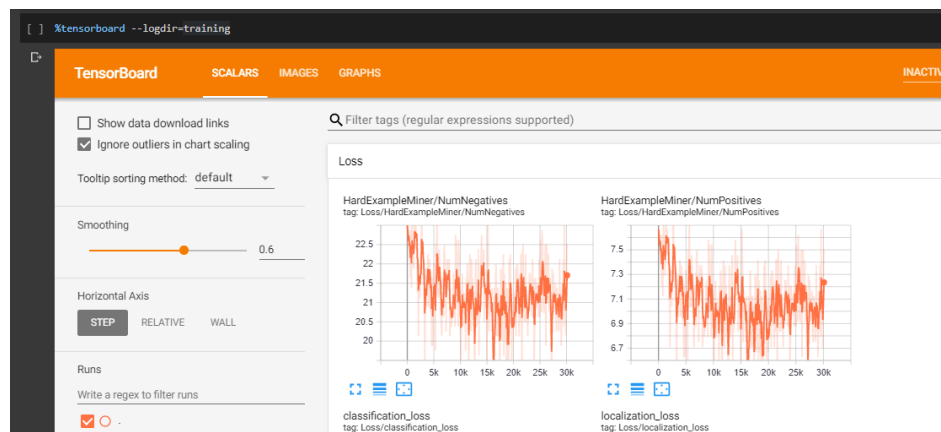


Figure 30 - Loading TensorBoard

Additionally, TensorBoard has a 'Graphs' tab which has a visualization of the model which was used during training. It can clearly show the model's layers and connections and its overall huge complexity as shown by **Figure 31**.

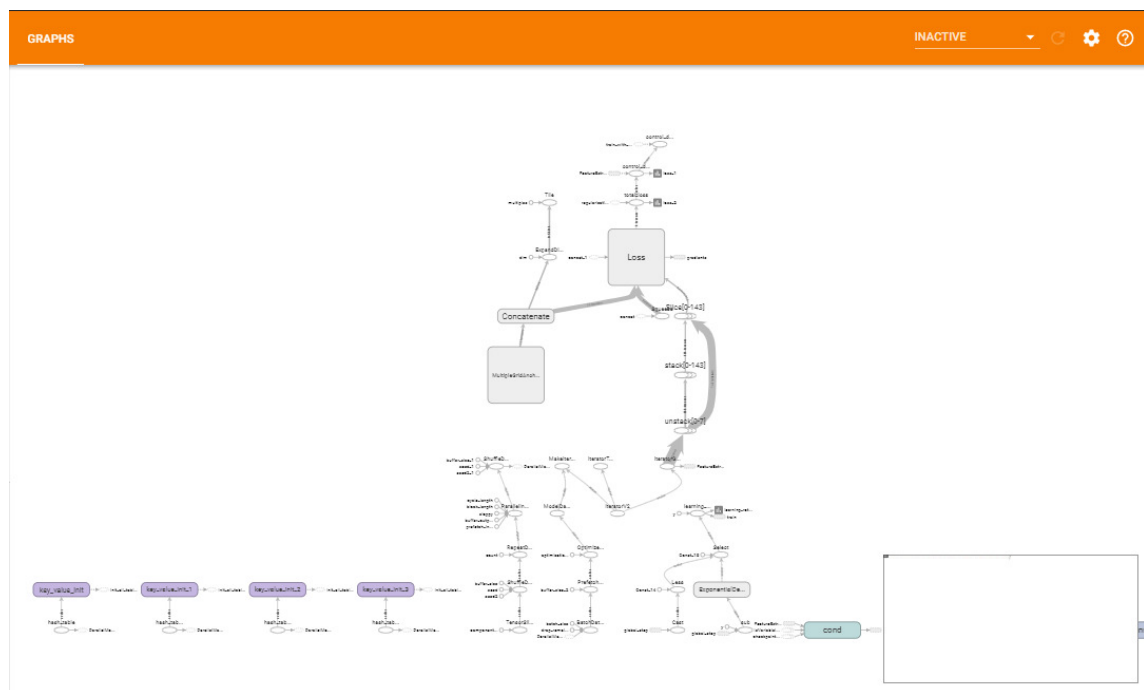


Figure 31 - TensorBoard's Graphs Tab Example

Lastly, it also has an 'Images' tab, shown by **Figure 32**, which at every checkpoint visualizes the model's predictions for specific images. It has a side by side display of 2 images, the left is the Image with the bounding-box and classification predictions and on the right is the actual image ground-truth boxes and the correct class. We can use this to observe how the model fits on those specific images and see how it evolves from each checkpoint.



Figure 32 - TensorBoard's Image Tab Example

3.6 Overfitting Prevention

“Lack of control over the learning process of our model may lead to overfitting” [29]. It had been clear that by letting the model keep training without any clear goals on when I was going to stop the process I was going to reach the point of my model overfitting and having a negative effect on its performance. I found 2 very popular methods in which their implementation seemed relatively simple to adapt.

3.6.1 Dropout

A very popular method of overfitting prevention is dropout. Every neuron within the neural network is given a probability of being temporarily ignored on calculations (example shown in **Figure 33**). Although this method may sound counter-intuitive since it limits the whole capabilities of the neural network, it can in fact help generalize the predictions made down the line improving its overall performance. After every iteration, any input values can be randomly eliminated. So, the neuron by trying to balance the risk and not use the same feature dependencies it generalises the values used in the weight matrix making them more evenly distributed.

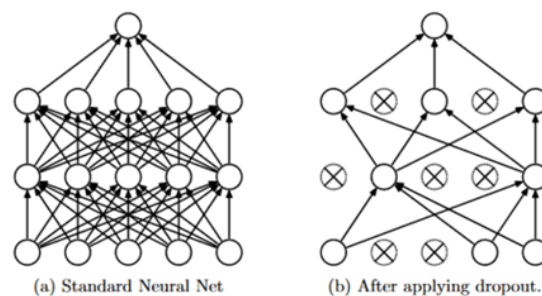


Figure 33 - Neural Network Dropout [30]

Firstly, I added the method of dropout which was very simple to adapt to my chosen model. In the model's configuration file in the 'convolutional_box_predictor' section there were 2 parameters that I could change (**Figure 34**). Either true or false to use or not use dropout, and a field named 'dropout_keep_probability' which was a percentage given for every unit in the NN to be kept during calculations. At a value of 0.8 there was a 20% chance that a neuron could be temporarily ignored.

```
box_predictor {
  convolutional_box_predictor {
    min_depth: 0
    max_depth: 0
    num_layers_before_predictor: 0
    use_dropout: true
    dropout_keep_probability: 0.8
    kernel_size: 1
    box_code_size: 4
    apply_sigmoid_to_scores: false
    conv_hyperparams {
      activation: RELU_6,
      regularizer {
        l2_regularizer {
          weight: 0.00004
        }
      }
    }
  }
}
```

Figure 34 - Hyperparameters Changed for Dropout

3.6.2 Early Stopping

Another overfitting prevention method would be an adaptation of Early Stopping. During training several metrics are provided, these can be used to measure the fitness of the model at its current state. By using these values gathered and by setting a limit on the maximum number of iterations during no progress has been made on the metrics then we stop the training process. We want to prevent the model from fitting to the validation data thus decreasing its overall fitness so it should be stopped at its optimal point (example shown in **Figure 35**). I would wish to prevent my model from reaching such a point since it will both waste additional time but also decrease its overall performance.

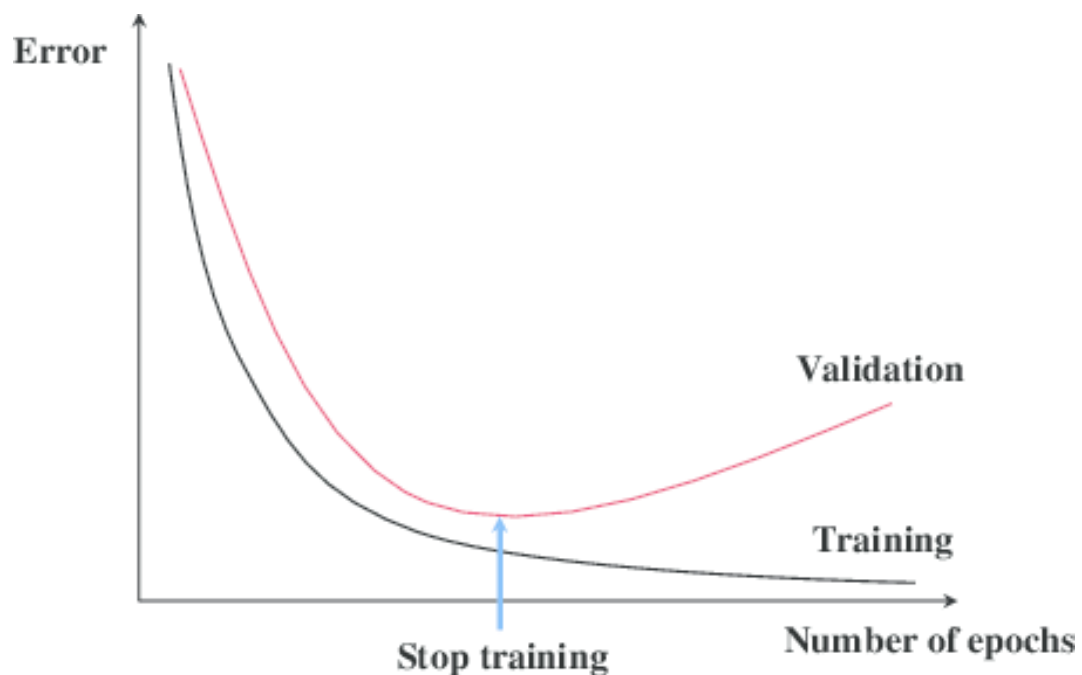


Figure 35 - Early Stopping Principle [31]

The idea seemed very intuitive. In theory, by iteratively looking at the metrics generated during training and evaluation you could calculate when no significant improvements had been made and stop the training process. By using TensorFlow's documentation [32] it seemed relatively simple to adapt as there were some premade functions within their experimental section that had exactly what I was looking for.

The issue was found only after I tried implementing those functions within the 'model_main' training script. Testing with some of the available functions I had to pass the metric name as well as a threshold value of when the hook would then stop the process. My implementation process as shown in **Figure 36** seemed to be wrong as I saw no results. The training process never stopped even when I used both training metrics and evaluation metrics with easily reachable threshold values.

```

early_stopping1 = tf.estimator.experimental.stop_if_no_decrease_hook(estimator, "loss_1", 100)

early_stopping2 = tf.estimator.experimental.stop_if_lower_hook(estimator, "loss_1", 3)

early_stopping3 = tf.estimator.experimental.stop_if_lower_hook(
    estimator,
    metric_name='total_loss',
    threshold=5,
    eval_dir=None,
    min_steps=0,
    run_every_secs=None,
    run_every_steps=100
)

early_stopping4 = tf.estimator.experimental.stop_if_higher_hook(
    estimator,
    metric_name='PascalBoxes_Precision/mAP@0.5IOU',
    threshold=0.75,
    eval_dir=None,
    min_steps=0,
    run_every_secs=None,
    run_every_steps=100
)

hooks = [early_stopping1, early_stopping2, early_stopping3, early_stopping4]

train_spec = tf.estimator.TrainSpec(
    input_fn=train_input_fn, max_steps=train_steps, hooks=[hooks])

```

Figure 36 - Early Stopping Implementation

3.7 Exporting

Since I had the ability to train properly but did not have the benefit of implementing early stopping, I had to constantly keep track of the training and evaluation process. Evaluation results were only generated and could be observed in TensorBoard only after the training process had been interrupted. In order to not interrupt the process at every single checkpoint I had to find a simpler solution.

After every checkpoint where the evaluation step was finished, I was returned with output such as the one shown in **Figure 37** which showed all of the metric values at that specific checkpoint. Using those values, I could observe the state of the model and see how well the model was fitting to the dataset.

```

I0512 14:16:09.981783 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.911732
I0512 14:16:10.309222 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.301308
I0512 14:16:10.617907 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.543204
I0512 14:16:11.091267 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.969437
I0512 14:16:11.451738 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.390205
I0512 14:16:11.828519 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.969790
I0512 14:16:12.190884 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.835946
I0512 14:16:12.639389 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.998240
I0512 14:16:13.011811 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.976850
I0512 14:16:13.366005 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.996679
I0512 14:16:13.749262 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.999962
I0512 14:16:14.147357 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.983222
I0512 14:16:14.542906 139985401157376 object_detection_evaluation.py:1311] average_precision: 0.973855
INFO:tensorflow:Finished evaluation at 2020-05-12-14:16:14
I0512 14:16:14.770621 139989110572928 evaluation.py:275] Finished evaluation at 2020-05-12-14:16:14
INFO:tensorflow:Saving dict for global step 22500: Loss/BoxClassifierLoss/classification_loss = 0.5849913, Loss/
I0512 14:16:14.770987 139989110572928 estimator.py:2049] Saving dict for global step 22500: Loss/BoxClassifierLo

```

Figure 37 - Evaluation Metrics Example

I was constantly checking that the total loss was decreasing and that the mAP for all classes was rising. At every 10,000 steps I preferred interrupting the process, where it caused an evaluation event file to be uploaded. With the use of my TensorBoard Notebook I was able to visually observe how the evaluation and training metrics were changing overtime, and I was able to observe if any irregularities occurred.

The key indicators which would show that the model was at its peak fitness in terms of training performance were when the mAP and total loss metrics had reached a plateau. An example of those 2 metrics within TensorBoard can be seen in **Figure 38**.

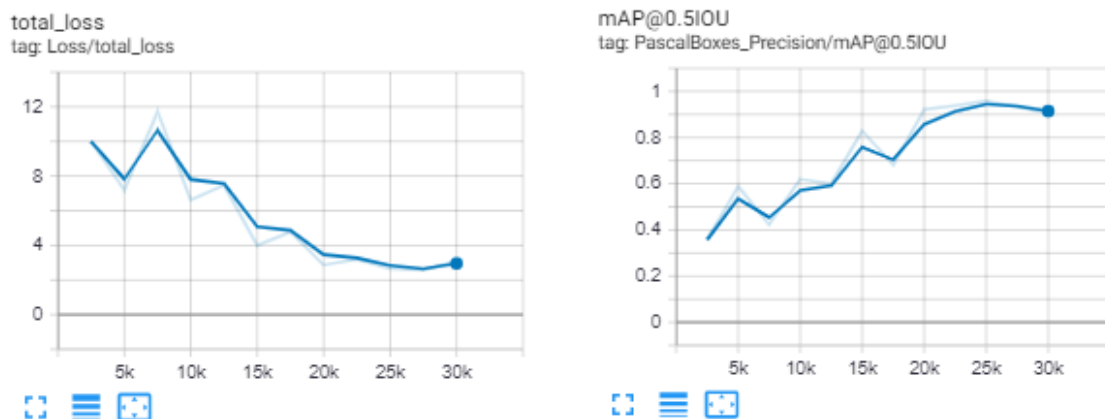


Figure 38 - Total_loss & mAP Graphs

When I believed that it had reached its optimal point, I would let it train for a few more checkpoints so I could have more options when deciding which checkpoint, I would like to export as my trained model. With the graphs in **Figure 34** as an example, I chose to export the checkpoint which was at 25,000 steps. The mAP was at its highest point and 'total_loss' had shown almost no change between 30,000 steps 25,000.

To export the model at that specific checkpoint I used TensorFlow's provided script 'export_inference_graph.py' as shown in **Figure 39**. Using that I pass the image tensor input type parameter, the pipeline's configuration path, the specific checkpoint which I wanted, and the output directory. The directory which was created had the same type of files as the pre-trained model but with the changes in the Neural Network of the new weights and biases. This model now had the ability to detect and recognize the road signs classes it had been trained on.

```
!python export_inference_graph.py \
--input_type image_tensor \
--pipeline_config_path ssd_mobilenet_v2_coco.config \
--trained_checkpoint_prefix training/model.ckpt-25000 \
--output_directory inference_graph_1/
```

Figure 39 - Exporting Script Command with Parameters

3.8 Testing

To deploy my trained exported model, I needed to create a Google Colab Notebook to tests its overall ability and examine its performance using the exported test images.

I adapted an online tutorial which I had found [33] which was using live webcam feed to perform object detection with TensorFlow's provided pre-trained models. By splitting the different functionalities into multiple cells, I simplified the whole process which made it easier to follow and test the custom trained models as shown in **Figure 40**.

```
Any model exported using the export_inference_graph.py tool can be loaded here simply by changing the path.

[ ] MODEL_NAME = 'inference_Final_SSD_13'

# Path to frozen detection graph. This is the actual model that is used for the object detection.
PATH_TO_CKPT = MODEL_NAME + '/frozen_inference_graph.pb'

Load the Tensorflow model into memory.

[ ] detection_graph = tf.Graph()
with detection_graph.as_default():
    od_graph_def = tf.GraphDef()
    with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
        serialized_graph = fid.read()
        od_graph_def.ParseFromString(serialized_graph)
        tf.import_graph_def(od_graph_def, name='')

    sess = tf.Session(graph=detection_graph)

Loading label map

Label maps map indices to category names, so that when our convolution network predicts 2, we know that this corresponds to Speed limit (50km/h). Here we use internal utility functions, but anything that returns a dictionary mapping integers to appropriate string labels would be fine

[ ] PATH_TO_LABELS = 'training/label_map.pbtxt'
NUM_CLASSES = 13

# Adding Labels and categories
label_map = label_map_util.load_labelmap(PATH_TO_LABELS)
categories = label_map_util.convert_label_map_to_categories(label_map, max_num_classes=NUM_CLASSES, use_display_name=True)
category_index = label_map_util.create_category_index(categories)
```

Figure 40 - ImageDetection Notebook Cell Examples

The main functionalities of Road Sign detection are within the function 'ObjectDetection'. It initially loads the tensors that would be used. Then with the use of the CV2 library I read the image and with 2 different ways I could resize the resolution of the image. Since I wanted to keep the same testing environment as it had during training, I wanted the resize test images to more appropriate height and width values.

By using the SSD MobileNet model during training I had to adapt fixed resolution sizes of 150 by 150. Since I wanted to maintain the same testing environment, I also had to change the resolution of the test images to those exact values (as shown by **Figure 41**)

```
# Resize image without aspect ratio
width = 150
height = 150
image = cv2.resize(image, (int(width), int(height)))
```

Figure 41 - Fixed Image Resizing

For the Faster-RCNN ResNet model a different image resizer was used in the pipeline configuration (Shown by **Figure 42**). It resized the image size whilst maintaining aspect ratio and also having a set of minimum and maximum dimensions.

```
image_resizer {
  keep_aspect_ratio_resizer {
    min_dimension: 150
    max_dimension: 300
  }
}
```

Figure 42 - Faster-RCNN ResNet Configuration Image Resizer

To adapt this method during the testing phase I had to check with the use of IF statements for the dimensions of the minimum and maximum resolution values and modifying them whilst also maintaining the image's aspect ratio.

```
# Resize image with aspect ratio
# With Min Dimensions:150 and Max Dimensions: 300
resolutionDict = {}
width = int(image.shape[1])
height = int(image.shape[0])
resolutionDict['width'] = width
resolutionDict['height'] = height
aspectRatio = height / width
minResol = min(resolutionDict.items(), key=lambda x: x[1])
maxResol = max(resolutionDict.items(), key=lambda x: x[1])

# Resizing to 150 minimum dimension
# Whilst having a cap for the other dimension for 300
if minResol[1] < 150:
    if minResol[0] == 'width':
        width = 150
        height = width * aspectRatio
        if height > 300:
            height = 300
    elif minResol[0] == 'height':
        height = 150
        width = height / aspectRatio
        if width > 300:
            width = 300

# Resizing to 300 maximum dimension
# Whilst having a cap for the other dimension for 150
elif maxResol[1] > 300:
    if maxResol[0] == 'width':
        width = 300
        height = width * aspectRatio
        if height < 150:
            height = 150
    if maxResol[0] == 'height':
        height = 300
        width = height / aspectRatio
        if width < 150:
            width = 150

image = cv2.resize(image, (int(width), int(height)))
```

Figure 43 - Resizing Image, Keeping Aspect Ratio with Capped Values (Code)

By using the provided image and the pre-loaded tensors a TensorFlow's session run operation is run and the actual object detection happens. The results are then stored in a tuple with a set of variables that contain the predicted bounding boxes for the detection, the scores of the predicted class as well as the class name. These variables are then passed with the image in a built-in TensorFlow visualization function where it modifies the image to now contain the predicted bounding box as well as the class the predicts the object is with a percentage number that indicates the confidence level for the predicted class (Shown by **Figure 44**).



Figure 44 - Road Sign Detection and Visualization

4. Results and Evaluation

To evaluate the decisions I had made, during the implementation process, I decided to compare trained models with only some minor property differences to show what the effect was. By keeping most environment variables, the same, I could accurately show the performance differences by comparing the evaluation metrics that would be gathered and also show its actual testing performance in terms of speed and accuracy. I decided to train every model on the same number of 30,000 train steps and use the same 'PASCAL VOC DETECTION' metrics to measure the evaluation when cross-referencing with the 'eval' dataset.

With the models' exported graphs, I would use the ImageDetection notebook where with the use of the TEST set, I could measure each model's performance as shown in **Figure 45**. To find an approximate measurement for the model's speed I measured the instance of time before and after all 150 images had been processed, I could use 2 very simple formulas to find the average time each image needed to be processed and also find an estimate for Frames Per Second (FPS) the model could run on a real-time processing application.

$\text{numImages} / (\text{endTime} - \text{startTime})$ would give me the model's processed FPS

$(\text{endTime} - \text{startTime}) / \text{numImages} * 1000$ would give me the average processed speed in milliseconds.

To find the accuracy of the model, I compared the class predicted with the actual class of the test image. When processing the GTSRB dataset when I extracted the 150 test images from the evaluation set, I also created a 'test_label.csv' that had all the information for each of the test images. During testing I could load the csv, store the actual class for each file name, and compare the predicted result with the actual class. If the predicted class were the same, I would count it as correct. By comparing the correct predictions with the number of images that had been processed (150) I could find a percentage accuracy of the model.

```
testImagesInfo = {}
with open('test_labels.csv', 'r') as currentfile:
    reader = csv.reader(currentfile)
    next(reader)
    for row in reader:
        testImagesInfo[row[0]] = row[3]

os.chdir('test_images')
TEST_IMAGE_PATHS = glob.glob('*.jpg')
TEST_IMAGE_PATHS.extend(glob.glob('*.png'))
correctDetections = 0
numImages = 0

startTime = time.time()

for imagePath in TEST_IMAGE_PATHS:
    numImages += 1
    correctDetections += ObjectDetection(imagePath, testImagesInfo.get(imagePath))

endTime = time.time()

print('FPS: ', round(numImages/(endTime - startTime),2))
print('Speed(ms): ', round(((endTime - startTime)/numImages)*1000,2))
print('Accuracy: ', round(correctDetections/numImages * 100,2),'%')
os.chdir("../")
```

Figure 45 - Model's Evaluation with the use of Test set

4.1 Classes (43) vs Classes (13) Performance Difference

Firstly, to evaluate the performance difference amongst using the whole 43 class dataset vs the trimmed 13 classed dataset I decided to create 2 models. I used the same 'ssd_mobilenet_v2_coco_2018_03_29' pre-model as an initializer whilst keeping mostly the same hyperparameter values. The only changes I had made between the 2 was within the configuration file and specifically on the number of classes and the number of examples in the evaluation set.

4.1.1 Comparing TensorBoard Results

After the 30,000 training step mark, I interrupted both training processes so I could visualize the results with the use of TensorBoard. I had 2 training folders that contained all evaluation and training metrics for each model.

By first visualizing the 43 class model I had observed a steady improvement on all evaluation metric graphs as shown in **Figure 46**. With 'total loss' declining from an initial evaluation at 2,500 steps with a result of 19.68 to 5.18 at 30,000 steps. Additionally, the mAP had increased to a point of 0.5747 which indicated that the general predictions of the model were improving significantly.

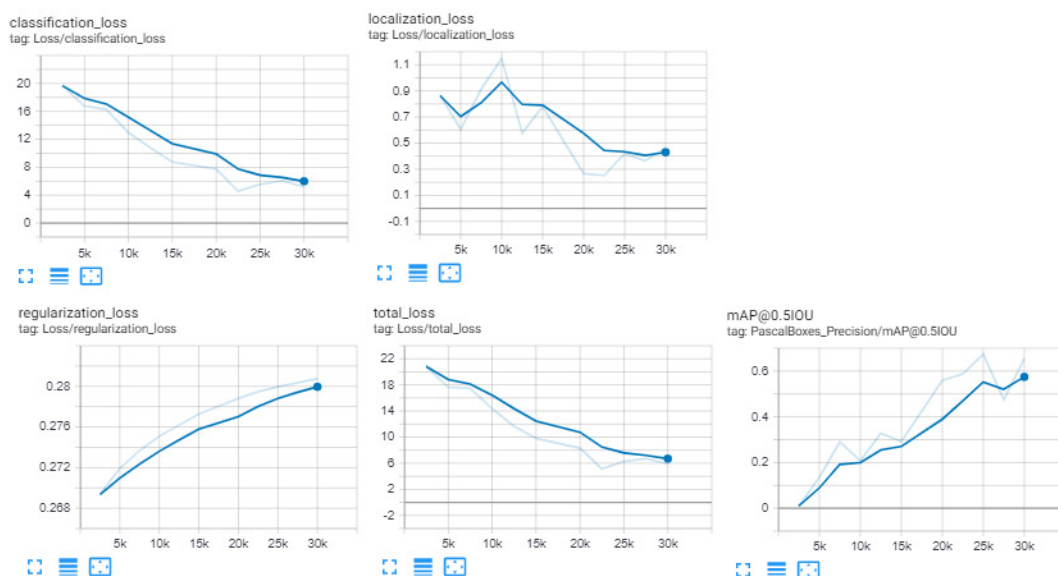


Figure 46 – 43-Class Evaluation Metrics Graphs

When comparing with the model which had trained on the 13 classes I saw a substantial difference amongst these metric values (**Figure 47**). With 'total loss' dropping at just 1.5 and with mAP of 0.99 this model was performing much better given the restricted training step process. The one issue that I saw was the 'regularization loss' which began dipping down half-way, but because it was varying on just 2 hundredths of a unit, I decided to discard it. This sign may have been an indicator that the model was beginning to overfit to the evaluation set but because it was so negligible, I could simply overlook it.

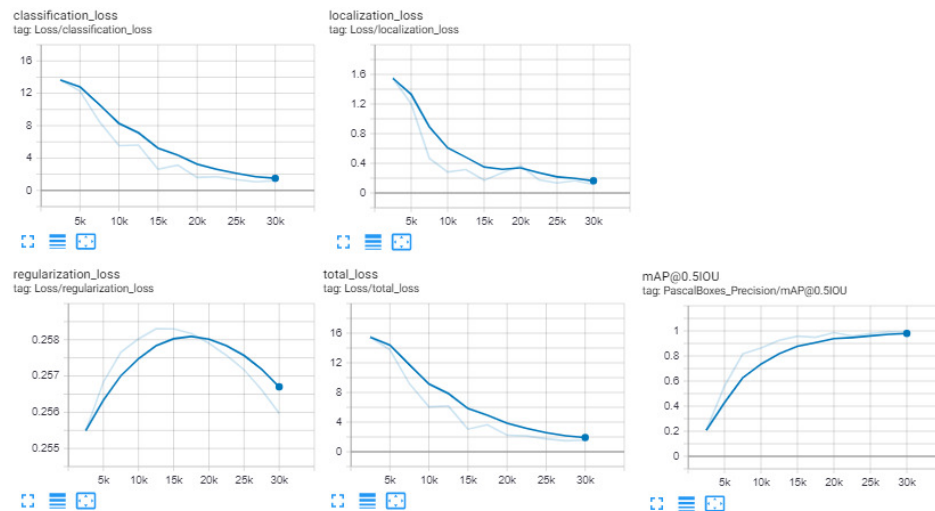


Figure 47 – 13-Class Evaluation Metrics Graphs

4.1.2 Checkpoints Extracted

By using the metrics observed within TensorBoard I had to pick which checkpoint I wanted to extract to test Road Sign Detection on.

For the 43 class model it was obvious that it could have been trained for much longer which would have provided with much better overall results but since I wanted to keep my experimentation process the same I just had to use the checkpoints up to 30,000 steps. With that in mind I decided to export at the 30,000 step mark which had the best overall results with the highest mAP and lowest total loss.

Additionally, this was the case for the 13 class trained model. The checkpoint at 30,000 steps was at its lowest point in terms of total loss and with a very high mAP of over 0.99 it was an easy choice for my exported deep learning model.

4.1.3 Comparing Performance Results

By testing the 43-class model I had observed it was running on **21.3FPS**. Each image took **46.9ms** on average to be processed which was relatively fast, but it had a pretty low accuracy of **32.67%**. It was obvious that the training process had been stopped prematurely for this specific model as all of its evaluation metrics had the potential to improve significantly. But, this also showed that at 30,000 steps the model is not reliable enough to be used in a real-time application, and since I was limited in the amount of time I could have spent in training, continuing the 43-class trained model was not viable.

For the 13-class trained model I saw that processing speed was much faster. By taking only **32.47ms** to process each image running at almost **30.8 FPS**, it was much faster than the previously tested model. With a substantial difference in the accuracy of **80%** I could now see that training with just 13 classes was much more feasible in terms of training time relative to performance.

4.2 Dropout

To generalize the previously used SSD MobileNet I decided to test the dropout method. This was in theory, going to help the model perform better on the unseen set of test images. By using similar configuration hyperparameters as the previously trained SSD MobileNet model I wanted to create a new one with the dropout method implemented. With the hyperparameter of 'use_dropout' set at true and 'dropout_keep_probability' with the value of 0.8, I was trained new model to see how much of an improvement the dropout method could give.

4.2.1 TensorBoard Results and Choosing Export

By following the same method as before I interrupted the training process at the 30,000 step mark. Reaching a total loss of 1.8 and with mAP of 0.98 the results were almost identical to the previously created trained model. The fitness visualized through the TensorBoard graphs as shown in **Figure 48** was almost the exact same. With only the minor increase of 0.3 in total loss it seemed that dropout may have did not affect the performance of the model.

With the use of these metric graphs I decided to export the 30,000 step checkpoint which again had the highest value of mAP and the lowest total loss.

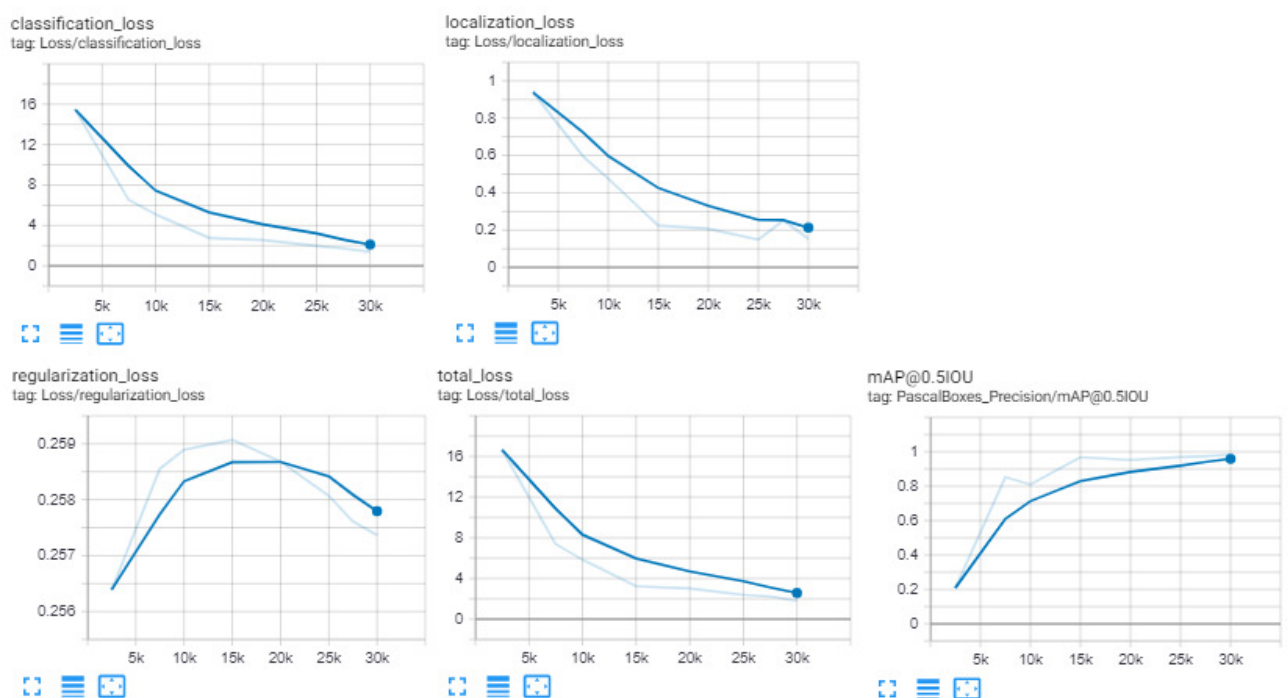


Figure 48 - TensorBoard Results on Model with Dropout

4.2.2 Performance Results

The overall performance of the model was almost identical which was to be expected. With an average processing time for each image at **32.84ms** and **30.45 FPS** this model had the same speed as the other trained model which did not use dropout. The unexpected difference was in the accuracy which showed a value of **74.67%**. Dropout did in fact had a negative effect on the performance at those 30000 training steps which was an unexpected result for a method that was supposed to improve the model's performance when testing on those unseen set of images.

4.3 Faster-RCNN ResNet

Since I had seen that SSD MobileNet had a substantially good performance for those set of training images I wanted to compare the performance with another chosen model and see how far I could improve it. Again, by keeping the same properties such as the number of classes at 13 I wanted to see how much of a difference the 'faster_rcnn_resnet101' model has from the 'ssd_mobilenet_v2' model.

4.3.1 TensorBoard Results and Choosing Export

The training process was mostly the same but the results that I saw such as total loss had reached a very low value of 1.19 which seemed very promising. The mean average precision for all of the classes had again a high value of 0.96 and all of the other metric graphs showed the same relative relationships (**Figure 49**). Again, my choice of export was at the same checkpoint of 30,000 since all of the metrics were at their best points.

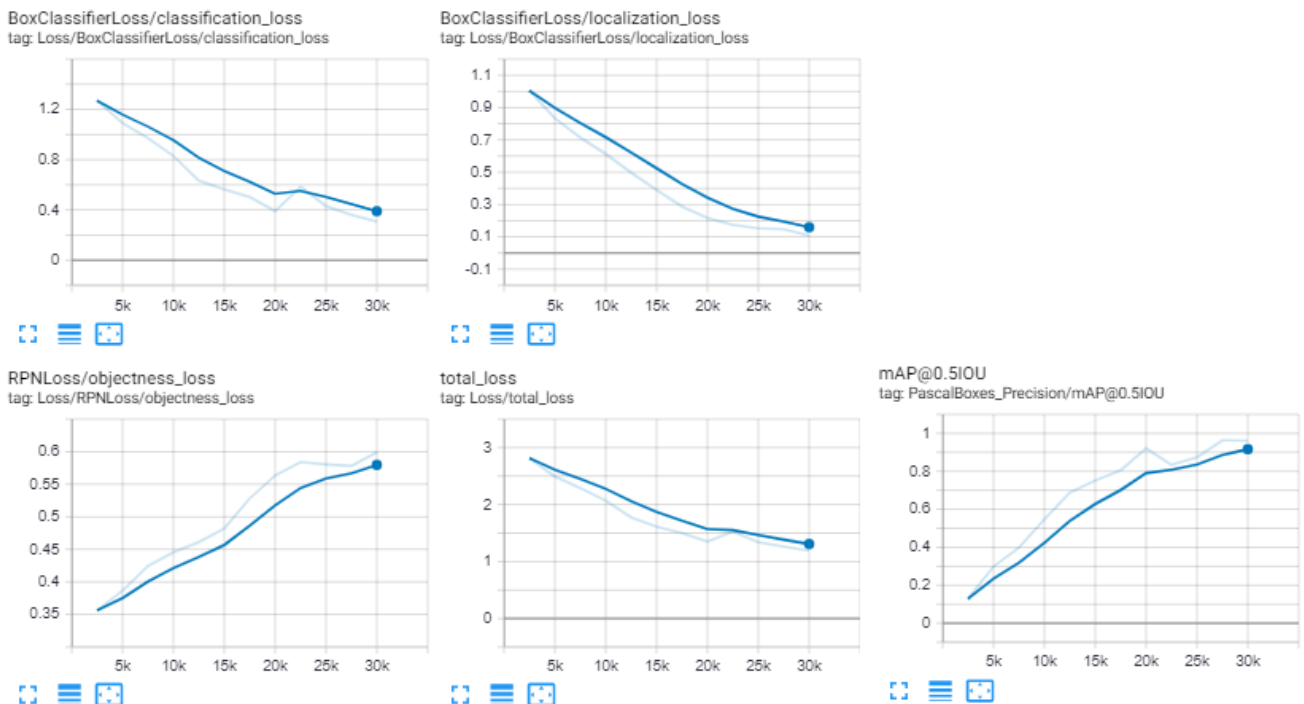


Figure 49 - TensorBoard Metric Graphs on Faster-RCNN ResNet

4.3.2 Performance Results

The testing results of this Faster-RCNN ResNet model were supposed to show a significant increase in accuracy. With the results as shown in **Figure 50** this seemed to not be the case. This image processing speed was substantially affected and dropped down to just **8.89FPS**, but the accuracy had very little effect to just **80.67%**.

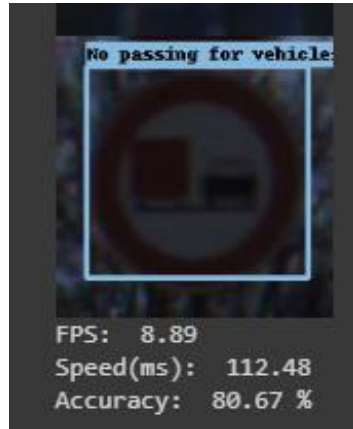


Figure 50 - Faster-RCNN ResNet Evaluation Result

4.4 Environment Difference

Finally, I wanted to observe how much effect switching to Google Colab had on my training process. With the use of a training metric 'global_step/sec' which is visualized in TensorBoard I could identify how many training steps per second were performed. By also knowing the batch size which I had used for each configuration I could calculate the performance increase I had.

The limitations of my GPU memory meant that I had to lower the batch size to the value of 8 in order to not run to the 'Out of Memory' errors. With the value of this metric I could observe as also shown in **Figure 51** that there were approximately 3 training steps of batch 8 at every second. Multiplying these 2 provides an estimated processing value of 24.

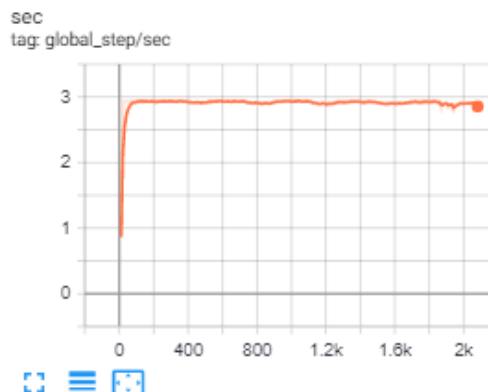


Figure 51 – 'Global Sec' Metric on Laptop

By using Google Colab's GPU I could take advantage of the higher memory capacity, so I increased the batch size to a value of 24. After carrying on with the training process I saw that the 'global_step/sec' metric was averaging at around a value of 5.75, as shown in **Figure 52**. Meaning that for every second almost 5.75 steps of batch size 24 had been carried out giving a value of 138.

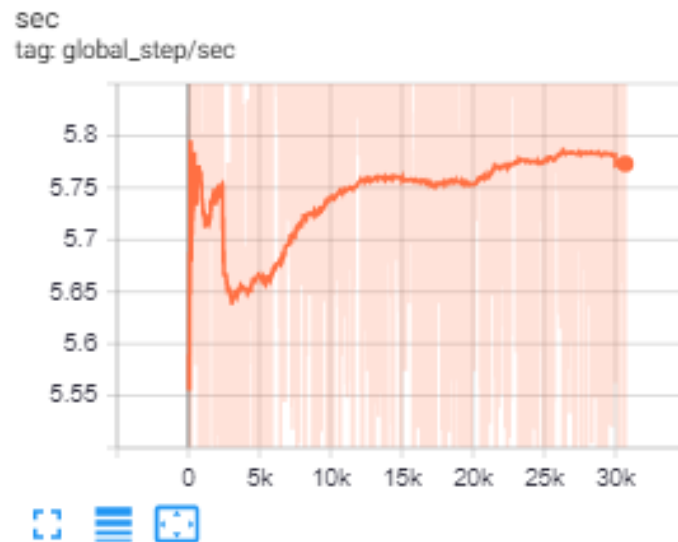


Figure 52 – 'Global Sec' Metric on Google Colab

By comparing the values from my laptop and Google Colab I could see a huge difference. With a 5.75 improvement when using this Cloud Based alternative, this showed that my early approach in switching from my laptop to this testing environment had saved me a substantial amount of time.

5. Conclusion of Results

By comparing the results which I had gathered I identified the effect my choices had done to the overall outcome of my results. At every evaluation step I was always choosing the last checkpoint as my preferred export as it was always the best available one. By interrupting the process at 30,000 steps I had never reached a point where the metrics of my model would reach a plateau. There was always extra room for improvement which also means that I should have trained the models for more.

The results which I had gathered as shown from the Following table indicate what effect each change had.

	FPS	Processing Time(ms)	Accuracy (%)
SSD MobileNet (43 Classes)	21.3	46.9	32.67
SSD MobileNet (13 Classes)	30.8	32.47	80.0
SSD MobileNet (13 Classes + Dropout)	30.45	32.84	74.67
Faster-RCNN ResNet (13 Classes)	8.89	112.48	80.67

Firstly, when comparing a 13-class model with the 43-classes I could see how much difference trimming the GTSRB dataset to a smaller number of classes had in all of the performance metrics relative to the training time that was carried out. Using the 13 classes I could conclude that the speed of the model had a **30% increase** and with an improvement in accuracy of **over 47%**.

The dropout method may have had an effect in the long run which would have prevented the model from overfitting to the training and evaluation image datasets but for the 30,000 training steps which I had used I saw a negative effect in the outcome of the performance. With the average speed being almost exactly the same I had seen a **decrease of more than 5%** in the overall accuracy of the model with dropout vs the one without.

Furthermore, when comparing the SSD MobileNet vs the Faster-RCCN ResNet model on the same number of 13 classes I was expecting a bigger difference to their overall performance. With a **70% negative effect** in speed and with only a **0.67% difference** in accuracy, using the Faster-RCNN ResNet model would not have been suitable for my real-time processing application. In an environment where car speeds could exceed over 100Km/h, choosing this relatively slow model for just an extra bit of accuracy would be unreasonable.

6. Future Work

The aim of my project to successfully recognize and identify road signs was met, but there are multiple areas where further improvements can be made.

6.1 Different Dataset

If I were not limited to the memory of my cloud storage, I could have used the MTSD which had much more available data than GTSRB. MTSD images could give better overall results as it is observed that the Deep Learning Models perform substantially better in high-resolution images than the low-resolution which the GTSRB dataset has. It will increase the needed training time exponentially, but the results of the trained model should be superior.

6.2 Hyperparameter Tuning

The task of furtherly tuning the number of hyperparameters of the trained model can provide better training results. Given the time constraints and the scale of my project, going through the large amount of the implicit and explicit hyperparameters and tuning them was not timely plausible.

For example, several additional data augmentation options as described in **appendix A** can be adapted which would act as having a larger dataset of images with varying conditions. This should cause training times to be much longer but in theory, could create a more capable model able to distinguish road signs from images in harsher situations.

6.3 Adapting Properly Overfitting Prevention Methods

Something that I have not successfully adapted to my project was the use of overfitting prevention methods. Although I had attempted to apply them throughout my training process their effects were either negligible or just not properly seen.

The correct application of these methods could have saved significant time as finding the optimal state of the Machine Learning model would have been much easier.

6.4 Additional Training Time

It is a prevalent issue using deep learning techniques, that training times can be significantly large. It had been clear that the consistent issue which I had from my chosen models was that I did not train them long enough. The models did take a considerable amount of time to reach that training point but given additional time the results should have been much better.

7. Conclusion

The goals of my project were on creating a prototype, capable of detecting and recognizing road signs using deep learning techniques. This aim has been successfully met, as I had created and tested several prototypes with the use of pre-trained models which I was then able to evaluate further.

Initially, finding an appropriate image dataset which I could process had proved to be more challenging than I had expected. There were plenty of available options but choosing on the right one was very important. The GTSRB dataset with its high number of low-resolution images was a huge convenience as I was not limited due to high memory consumption which the alternatives caused. It gave me the flexibility to train a very accurate model in a time-efficient manner. I was able to easily manipulate, trim, and process the dataset to a size that was much suitable for me.

Moreover, switching to a cloud-based alternative seemed to be a very important move as I was not limited to the capabilities of my system. I was able to carry out very computer-intensive processes in a much shorter period which allowed testing and exploring several different approaches and methods. With the benefit of using this highly developed TensorFlow Python Library I was able to test several different deep learning models with minor changes but significantly affecting their overall performance.

I compared the effect of using a smaller number of classes had in the performance of the model relative to training time. As was expected the SSD MobileNet model which was training in 13 classes rather than all 43 had performed substantially better with over 47% in higher accuracy. Additionally, I compared the effects of applying the dropout method which in the given time frame provided me with negative results. Furthermore, I had tested the effects of using a different deep learning model, the Faster-RCNN with ResNet. It was expected to have a significant benefit in performance with some effect in processing speed but with the results that I had gathered this seemed to not be the case. The minor increase in accuracy with the significant drop in image processing speed, just reaching 9FPS made it unsuitable for the applications of my project which are expected to be mostly high-speed environments.

Lastly, I had concluded that the SSD MobileNet trained on just 13 classes is a very suitable model for Road Sign detection applications. Its development approach of being used mostly on low-end systems with high processing speed was shown as it was the fastest amongst all of the models tested. With an accuracy of over 80% and enough processing speed to run on more than 30FPS this model meets all of the key requirements. In conclusion, the results which I have found, proved that creating a deep learning model that is capable of accurately and quickly, detecting road signs is plausible. Their overall performance shows that they can be used for real-time processing applications.

8. Reflection

This Final year project proved to be my most challenging endeavour as it dared me to explore a subject that I had never previously worked with. Making a choice for my final year project was relatively easy as I was very intrigued in the field of artificial intelligence, specifically machine learning. I was highly motivated to tackle a problem with a very steep learning curve because, as I had expected, it challenged me to truly put my mind to work. I had previously seen several applications of Artificial Neural Networks carrying out simple tasks but never to the extent of being implemented for real-life uses.

Properly managing my time had proved to be very hard. I underestimated the time each task would take which veered me significantly from my initial time plan, right from the very beginning. Additionally, the difficulty of the subject was not of any help. Since it was the first time working with this technology, making changes and bug fixing was very difficult and the process of training neural networks was very time-consuming. I had to wait for a considerable amount of time to see if the changes that I was making were of any benefit. Truly sitting down, researching, and learning the background I was then able to understand how to approach solving my problems.

Due to the recent pandemic and by following my government's repatriation process I found myself stuck in a hotel for 14 days where I had to quarantine before being allowed to go back to my home. Due to this restriction I found the opportunity to focus strictly on creating my final report. With no distractions I was much more productive. I was able to re-think the approach of my project, made some significant changes, and managed to get the results I was finally happy with. For me, this 2-week quarantine the best opportunity that I had, and it played a significant role in the outcome of my dissertation. Reflecting on the overall process of my time creating this project I should have been better at time management and task prioritization. The complexity of the background and the unfamiliar technologies was a difficult challenge, but seeing the outcome was all worth it.

9. Appendices

APPENDIX A

TensorFlow Object Detection Data Augmentation Options

- `NormalizeImage` `normalize_image` = 1;
- `RandomHorizontalFlip` `random_horizontal_flip` = 2;
- `RandomPixelValueScale` `random_pixel_value_scale` = 3;
- `RandomImageScale` `random_image_scale` = 4;
- `RandomRGBtoGray` `random_rgb_to_gray` = 5;
- `RandomAdjustBrightness` `random_adjust_brightness` = 6;
- `RandomAdjustContrast` `random_adjust_contrast` = 7;
- `RandomAdjustHue` `random_adjust_hue` = 8;
- `RandomAdjustSaturation` `random_adjust_saturation` = 9;
- `RandomDistortColor` `random_distort_color` = 10;
- `RandomJitterBoxes` `random_jitter_boxes` = 11;
- `RandomCropImage` `random_crop_image` = 12;
- `RandomPadImage` `random_pad_image` = 13;
- `RandomCropPadImage` `random_crop_pad_image` = 14;
- `RandomCropToAspectRatio` `random_crop_to_aspect_ratio` = 15;
- `RandomBlackPatches` `random_black_patches` = 16;
- `RandomResizeMethod` `random_resize_method` = 17;
- `ScaleBoxesToPixelCoordinates` `scale_boxes_to_pixel_coordinates` = 18;
- `ResizeImage` `resize_image` = 19;
- `SubtractChannelMean` `subtract_channel_mean` = 20;
- `SSDRandomCrop` `ssd_random_crop` = 21;
- `SSDRandomCropPad` `ssd_random_crop_pad` = 22;
- `SSDRandomCropFixedAspectRatio` `ssd_random_crop_fixed_aspect_ratio` = 23;

10. References

- [1] En.wikipedia.org (2020), Motor vehicle fatality rate in U.S. by year [online], Available at: https://en.wikipedia.org/wiki/Motor_vehicle_fatality_rate_in_U.S._by_year (Accessed 14 May 2020)
- [2] Bill Widmer (2019), 50+ Car Accident Statistics in the U.S. & Worldwide [online], Available at: <https://www.thewanderingrv.com/car-accident-statistics/> (Accessed 14 May 2020)
- [3] En.wikipedia.org (2019), Traffic-sign recognition [online], Available at: https://en.wikipedia.org/wiki/Traffic-sign_recognition (Accessed 14 May 2020)
- [4] TechLeer (2017) Google To Help Developers In Object Identification Using Tensorflow Object Detection API [online], Available at: <https://www.techleer.com/articles/123-google-to-help-developers-in-object-identification-using-tensorflow-object-detection-api/> (Accessed 14 May 2020)
- [5] Fred Lambert (2020), Tesla Autopilot crash rate increases, but still lower than without Autopilot – Electrek [online], Available at: <https://electrek.co/2020/01/16/tesla-crashes-autopilot-increase-better-without-autopilot/> (Accessed 14 May 2020)
- [6] Carscoops (2020) [Video] Available at: <https://www.youtube.com/watch?v=fKXztwtXaGo> (Accessed 14 May 2020)
- [7] Sweden Mapillary AB , Mapillary - Street-level imagery, powered by collaboration and computer vision [online], Available at: <https://www.mapillary.com/dataset/trafficsign> (Accessed 14 May 2020)
- [8] Benchmark.ini.rub.de (2012), German Traffic Sign Benchmarks [online] Available at: <http://benchmark.ini.rub.de/?section=gtsrb&subsection=news> (Accessed 14 May 2020)
- [9] Benchmark.ini.rub.de (2012), German Traffic Sign Benchmarks [online] Available at: <http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset> (Accessed 14 May 2020)
- [10] Christian Igel (2019), Public Archive: daaeac0d7ce1152aea9b61d9f1e19370 [online] Available at: <https://sid.erda.dk/public/archives/daaeac0d7ce1152aea9b61d9f1e19370/published-archive.html> (Accessed 14 May 2020)
- [11] Daffodil Software (2017), 9 Applications of Machine Learning from Day-to-Day Life Medium [online] Available at: <https://medium.com/app-affairs/9-applications-of-machine-learning-from-day-to-day-life-112a47a429d0> (Accessed 14 May 2020)
- [12] Facundo Bre (2017), Fig. 1. Artificial neural network architecture (ANN i-h 1-h 2-h n-o) [online] Available at: https://www.researchgate.net/figure/Artificial-neural-network-architecture-ANN-i-h-1-h-2-h-n-o_fig1_321259051 (Accessed 14 May 2020)
- [13] En.wikipedia.org (2019), TensorFlow [online] Available at: <https://en.wikipedia.org/wiki/TensorFlow> (Accessed 14 May 2020)
- [14] GitHub, tensorflow/models [online] Available at: https://github.com/tensorflow/models/tree/master/research/object_detection (Accessed 14 May 2020)

- [15] GitHub, tensorflow/models [online] Available at: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md (Accessed 14 May 2020)
- [16] Sik-Ho Tsang (2018), Review: SSD — Single Shot Detector (Object Detection) [online] Available at: <https://towardsdatascience.com/review-ssd-single-shot-detector-object-detection-851a94607d11> (Accessed 14 May 2020)
- [17] Eddie Forson (2017), Understanding SSD MultiBox — Real-Time Object Detection In Deep Learning [online] Available at: <https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab> (Accessed 14 May 2020)
- [18] Jiwon Jeong (2019), Deep Dive into the Computer Vision World: Part 1 [online] Available at: <https://towardsdatascience.com/deep-dive-into-the-computer-vision-world-f35cd7349e16> (Accessed 14 May 2020)
- [19] Lunit Tech Blog (2017), R-CNNs Tutorial [online] Available at: <https://blog.lunit.io/2017/06/01/r-cnns-tutorial/> (Accessed 14 May 2020)
- [20] mc.ai (2019), Resnet architecture explained [online] Available at: <https://mc.ai/resnet-architecture-explained/> (Accessed 14 May 2020)
- [21] Nick Zeng (2018), An Introduction to Evaluation Metrics for Object Detection [online] Available at: <https://blog.zenggyu.com/en/post/2018-12-16/an-introduction-to-evaluation-metrics-for-object-detection/> (Accessed 14 May 2020)
- [22] Supervise (2020), Precision and Recall - Custom Plugin [online] Available at: <https://supervise.ly/explore/plugins/precision-and-recall-75278/overview> (Accessed 14 May 2020)
- [23] (2013), Object Detection Using SURF and Superpixels [online] Available at: https://www.researchgate.net/figure/Illustration-of-meaning-of-false-positive-Fp-false-negative-Fn-and-true-positive_fig1_273687795 (Accessed 14 May 2020)
- [24] Koo Ping Shung, Accuracy, Precision, Recall or F1? [online] Available at: <https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9> (Accessed 14 May 2020)
- [25] Research.google.com, Colaboratory – Google [online] Available at: <https://research.google.com/colaboratory/faq.html> (Accessed 14 May 2020)
- [26] Dat Tran (2017), datitran/raccoon_dataset [online] Available at: https://github.com/datitran/raccoon_dataset/blob/master/generate_tfrecord.py (Accessed 14 May 2020)
- [27] GitHub, tensorflow/models [online] Available at: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/installation.md (Accessed 14 May 2020)

- [28] Jonathan Hui (2018), Object detection: speed and accuracy comparison (Faster R-CNN, R-FCN, SSD, FPN, RetinaNet and YOLOv3) [online] Available at: https://medium.com/@jonathan_hui/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcn-ssd-and-yolo-5425656ae359 (Accessed 14 May 2020)
- [29] Piotr Skalski (2018), Preventing Deep Neural Network from Overfitting [online] Available at: <https://towardsdatascience.com/preventing-deep-neural-network-from-overfitting-953458db800a> (Accessed 14 May 2020)
- [30] Amar Budhiraja (2016), Dropout in (Deep) Machine learning [online] Available at: <https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5> (Accessed 14 May 2020)
- [31] Konstantin (2017), Four Years Remaining » Blog Archive » The Mystery of Early Stopping [online] Available at: <http://fouryears.eu/2017/12/06/the-mystery-of-early-stopping/comment-page-1/> (Accessed 14 May 2020)
- [32] TensorFlow, tf.estimator.experimental.stop_if_higher_hook [online] Available at: https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/estimator/experimental/stop_if_higher_hook (Accessed 14 May 2020)
- [33] Harrison Kinsley (2017) , Streaming Object Detection Video - Tensorflow Object Detection API Tutorial [online] Available at: <https://pythonprogramming.net/video-tensorflow-object-detection-api-tutorial/?completed=/introduction-use-tensorflow-object-detection-api-tutorial/> (Accessed 14 May 2020)