Vehicle Detection in Deep Learning

TEOH PER NIAN

A project report submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering (Honors) Electronic Engineering

Faculty of Engineering and Green Technology

University Tunku Abdul Rahman

May 2019

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature : _____

Name : Teoh Per Nian

ID No. : 14AGB03306

Date : 12 / 4 / 2019

APPROVAL FOR SUDMISSION

I certify that this project report entitled **Vehicle Detection in Deep Learning** was prepared by **TEOH PER NIAN** has met the required standard for submission in partial fulfillment of the requirements for the award of Bachelor of Engineering (Hons.) Electronic Engineering at University Tunku Abdul Rahman.

Approved by,

Signature : _____

Supervisor : Dr. Yap Vooi Voon

Date : 12 / 4 / 2019

ACKNOWLEDGEMENTS

I would like to thank everyone who had contributed to the successful completion of this project. I would like to express my gratitude to my research supervisor, Dr. Yap Vooi Voon for his invaluable advice, guidance and his enormous patience throughout the development of the research.

In addition, I would also like to express my gratitude to my loving parents and friends who had helped and given me encouragement, continuous support and motivation during the study.

Vehicle Detection in Deep Learning

ABSTRACT

Robust and efficient vehicle detection is an important feature to utilize in the smart transportation system. With the development of computer vision techniques and accessibility of large-scale traffic transport data, deep learning has been enabled to on-road vehicle detection algorithms. In addition, traffic transportation system involves death and life concern which requiring high accuracy to ensure safety, also, the detection system for autonomous driving requires real-time inference speed in order to guarantee prompt vehicle control.

In this report, a brief concept of training a deep CNN and how deep CNN works in object classification and localization is presented. The objective of this project is vehicle detection with deep learning, so, vehicles data set from highway, urban road and housing area had been collected and applied to the deep learning and computer vision algorithms. Due to the limited resources for training large-scale data set, the detecting classes will be limited to car, bicycle and motorcycle. Each class has roughly same amount of training images with each other. Some experiments have been conducted in this project to figure out which batch size performing well in the training process. Moreover, output of the convolutional layers has been visualizing for better understanding in CNN working principal. Finally, the result the vehicle detection performance in this project still have room from further improving, and a higher accuracy performance can be easily achieved by acquiring adequate data set and find the suited hyper-parameters to train the model.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Background

Deep learning is a class of state-of-the-art machine learning techniques that has sparked tremendous global interest in the recent years. It continued to be discovered and developed by most of mainstream company such as Nvidia, Mircosoft, Google, Amazon and Facebook etc. The deep learning field had been applied in many fields including from new internet services, like Google Assistant, have learned speech from sound to the deep learning self-driving cars to recognize and localize vehicles for making a better decision in avoiding obstacles.

Recent years, large amount of transportation data easily collected from multiple sources including road sensors, GPS, CCTV, incident reports and many more. Following the AI trends, this big transportation data can be utilize in intelligent traffic system which helps modern transportation making smart decision. An efficient smart decision support system has the potential to minimize incident response time, reduce congestion duration and enhance situation awareness.

However, processing and modelling traffic data are challenging because of the complexity of road structures, traffic patterns and spatial-temporal dependencies among them. Most of prediction systems used shallow traffic models which would not be able process and utilize such complex big data scenarios. Therefore, deep learning architecture become a rising research interest in the machine learning field. With sufficient amount of data information, the deep learning algorithms are able to learn complicated features and function from big data. The biggest advantage of deep learning over traditional methods is that the learning process is automated and

trained repeatedly to reach specific task's objective, without any human involvement. (Nguyen et al., 2018)

1.2 Artificial Intelligence Overview

Generally, the idea of AI is an intelligent machine that could possess human-like characteristics, including knowledge, reasoning, thinking and learning. These also known as "General AI" which is unrealistic-like to develop, at least until now. Therefore, most of the project or research are focusing on "Narrow AI". It is technologies that can accomplish specific task such as high-level strategic game (Alpha-Go), self-driving car, object recognition, data mining and virtual assistance.

1.2.1 Machine Learning

Machine learning is a set of algorithms that train computer from learning without being programmed in detail. The basic idea of machine learning is to program an algorithm that can statistically analyze the input data to predict an output. It required several disciplines working together such as statistic, probability, information theory and analysis. By using this technology, machine can be trained by manipulating large amount of input data and recognizing the data patterns in order to accomplish specific task.

Supervised learning requires a set of data to perform algorithms training. These labeled datasets contain variables, or attribute, the model going to analyze and use for prediction. Regression and classification are popular examples for supervised learning algorithms.

Unsupervised learning is learning the underlying structure or pattern from the given data. The data feed to unsupervised algorithm are not labeled, means that the algorithms are left to discover the data structures by themselves. K-means clustering and association rule are examples for unsupervised learning algorithms.

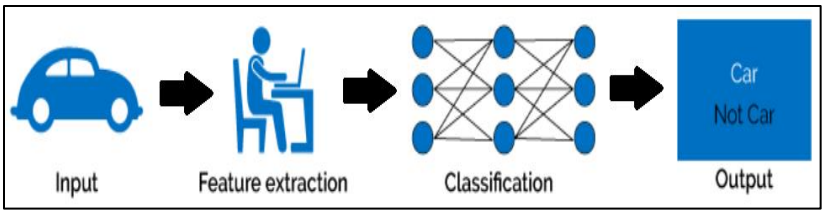


Figure 1.1: An example of how machine learning works

1.2.2 Deep Learning

Deep learning is a subset of machine learning concerned with artificial neural network algorithms. This model attempts to create meaningful representative of given data with multiple levels of abstraction by passing through multiple processing layers. Back-propagation algorithm make the machine "learn" from the dataset by changing neural network internal parameters from previous layer to improve fitting by reducing error prediction. Deep Neural network have greatest potential in areas such as processing images, audio signal and video to classified or recognize the information content in the image, signal or video.

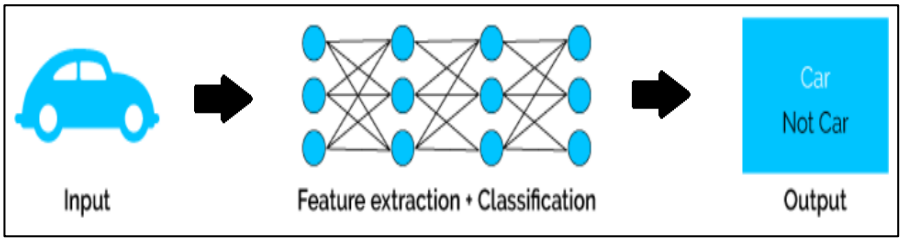


Figure 1.2: An example of how deep learning works

1.3 Problem statement

Deep learning is a way to solve the complicated task, but there have many types of deep neural networks architecture can be used to perform certain task. For example, architecture neural network (ANN) capable to learn underlying structure of input datasets based on the data attribute and model, to classified them in a meaningful manner. But ANN is not suitable to process image-based data due to

each neuron in a layer is fully connected to the neurons at next layer. Hence, when convert an image into a single vector and input to the ANN would lead to huge amount of parameter eventually consuming a lot of computation power. Deep convolution neural network (CNN) can be solved this problem due to window filters in convolution layer act as extractor to extract the feature by sliding entire the image, hence, the parameter can be reduced down when transfer to deeper layer. The deep CNN only classified different objects in different categories, but the object localization is important for vehicle detection. Thus, applying a suitable deep neural network which can detect, classify and localize the object is an important consideration to detect the vehicle on highway or urban road.

To train a deep learning model, it requires huge amount of data to make the model has a good generalization in detection. The more data used to train the model, the more features can be recognized or learn by model, hence, a better accuracy in detection. However, collecting and labeling a quality data is very expensive which usually consume a lot of time. Since, training a deep learning require a high computational power, while using low-performance computer system would take days or even weeks to complete the training and the outcome quality is largely depends on the hardware performance too. The latest GPU technologies nowadays such as Nvidia GTX or RTX can perform mathematical intensive computing operations parallelly and have enough virtual memories for feeding large amount of data to model per iteration. Unfortunately, a high-performance GPU required sub component to support it, hence, adding more cost to the hardware requirements in deep learning fields. Without the doubt, budget constraint will be taken into consideration from switching a low-performance computer machine to high-performance computer machine.

1.4 Aims and Objective

The objectives of the thesis are shown as following:

i. To understand the working principle of deep convolution neural network.

ii. To perform classification and localization on real life vehicle images.

iii. To perform real-time vehicle detection using deep learning.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

One of the applications of deep learning is in self-driving cars. Generally, self-driving car tends to be more reliable when putting it on the road surfaces which are well-maintained and line are marked clearly and accurately. Many automakers pursuing their product that can reduce driver work load and adding more safety features on their product. Most of the fully self-driving cars are equipped with expensive sensor such as LIDAR, high-precision GPS, and radar.

The commercial grade autonomous vehicles are equipped with radar, sonar and cameras. Detecting the car in long range required radar, while detecting nearby car can rely on sonar. The cameras act as computer vision which help to detect the lane as well as detecting redundant object at moderate distance. Radar working well for detecting vehicles, but has ambiguity to distinguish different metal object such as tin cans, thus, false decision would raise while moving on the road. Also, radar convey little orientation information and imperfect measurement on the lateral position of object, resulting difficulties for localization on sharp bends. Compared to radar and sonar, cameras contain more on road information, and it could serve as reliable addition sensor for autonomous car. The classic computer vision techniques are not reaching production grade on automation vehicles yet due to the techniques still not mature and require complicated modeling. (Huval et al., 2018)

Deep learning is alternative way for computer vision. Recently, it enjoys great accomplishment in image and video recognition (convolution neural network used in competition ILSVRC-2012 by Krizhevsky et al) become a solution to the

classic computer vision. Deep learning require huge amount of data and heavy computation, but minimal hand-engineering. Engineer can evaluate the deep neural networks with different driving environment and situation by input training data.

2.2 Real-time vehicle detection using deep learning.

In order to perform a real-time system, the detection system capable of operating at greater than 10Hz on laptop GPU. Also, the system used is able to detect vehicles more than 100m away, hence, the image resolution used will be higher. The Overfeat CNN detector, which is integrated object detection, localization and classification task all into one network. A “sliding window” will slide all over the image to detect the vehicle. Once an object is detected, a bounding box through regression is used to predict the object location. The classifier will ignore the object if it cannot distinguish an object within an image. This is undesirable prediction, which can only predict one object, as two different objects appear in an input view.

To ensure that all object in the input view can be classified, many contexts with different views are taken from the image by using skip gram kernels. Then, the classifier trained to recognize an object when it appears within its context view. The redundant bounding box location appears when multiple object appears, the network incorrectly predicting that there is a box between two merged bounding boxes to minimize the lost function. The network mistakenly decides that there is a third object between two merged bounding boxes. This is dangerous for autonomous car when running on the road, which falsely believes there is a car, actually it is not, and emergency breaking is falsely applied. This problem would become the bottleneck of a real-time system due to the merged bounding box is hardly parallelizable as the CNN (Huval et al., 2018).

2.3 Comparison between R-CNN and Faster R-CNN network.

The R-CNN network is made up of three modules. Firstly, it extracts region proposals in different aspect ratio. The network provide comparison with labelled work by using selective search methods. Secondly, it includes a convolution neural network, creating an attribute vector for each region with constant length. The convolution neural network will select the possible transformations of the proposed region and a bounding box regression used to improve the localization accuracy. Finally, it includes a linear support vector machine to classify the assortment of regions. Each produced attribute vector from convolution neural network is classified using the support vector machine. When there is high-intersection overlap for specific region, a non-maximum suppression will be applied to filter the overlap region and preserve the highest score region.

For the Faster R-CNN network is made up of two components. The first component is called Region Proposal Networks (RPN). The RPN produces multiple of rectangle object on an input image and each of the object will have an objectivity score. The RPN is designed with fully convoluted network. The second component is called Faster R-CNN detector which share a common set of convolution layers with RPN. The Faster R-CNN is a single, unified network used for detection network.

The Faster R-CNN has improved the regional proposal computation as a bottleneck by decreasing the process time of detection in R-CNN. Furthermore, RPN integrated with detection network shares full image convolution characteristics resulting free region proposals can be made. Faster R-CNN unlike R-CNN require external zone recommendation. In addition, the RPN provide an accurate district proposal, hence, improve the speed of detection and overall accuracy. An experiment made by Yilmaz, et al.(2018) stated that the mean average precision (mAP) for Faster R-CNN and R-CNN respectively, at approximately 0.76 and 0.65 values. For vehicle detection, results obtained from Faster R-CNN have higher detection accuracy compared with the R-CNN. In addition, the speed of object detection is become faster and more reliable through Faster R-CNN network.

2.4 Unified, Real-time object detection system YOLO.

Comparing to traditional object detection method, YOLO has extremely fast detection. This is because YOLO frame the detection as single regression problem, using a single convolutional network to predict multiple bounding boxes, class score, and objectiveness for those boxes simultaneously. The network runs on a Titan X GPU gains 45 fps with no batch processing and a fast version gains more than150 fps. This result reflecting the YOLO able to process a real-time streaming video with a very little of latency.

Furthermore, YOLO sees the entire image and making predictions globally based on the contextual information unlike receptive field and region proposal-based techniques. YOLO has a lesser background errors compared to Fast R-CNN. In addition, YOLO learns wide range of object representations. When trained on real life images and tested on artwork, YOLO gave better performance compare with top detection methods such as DPM and R-CNN by a wide margin. When applying a new or unexpected input, YOLO will less likely to break down due to its highly generalizable properties (Joseph, et al., 2015). Some performance and detection speed comparison between different method have made by Joseph, et al. (2015) as shown in Table. The Fast YOLO gave the highest fps among the other detectors and its mAP has doubled as any other real-time detector. YOLO has 10mAP higher than the fast version while still able to perform real-time detection as well.

Real-Time Detectors	Train	mAP	FPS
100Hz DPM [30]	2007	16.0	100
30Hz DPM [30]	2007	26.1	30
Fast YOLO	2007+2012	52.7	**155**
YOLO	2007+2012	**63.4**	45
Less Than Real-Time			
Fastest DPM [37]	2007	30.4	15
R-CNN Minus R [20]	2007	53.5	6
Fast R-CNN [14]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[27]	2007+2012	73.2	7
Faster R-CNN ZF [27]	2007+2012	62.1	18

Figure 2.1 : Real-Time Systems accurancy and detection speed on PASCAL V0C 2007. (Joseph, et al., 2015)

2.5 Benefits of adding batch normalization layer.

Theoretically in deep networks, high learning rate may cause gradients explode or vanish, as well as the network not learn at optimal. Batch Normalization tendency to speed up training without accuracy trade-off has become a favorite technique in deep learning. An experiment conducted by Johan, et al. (2018) have found that the Resnet model without batch normalization converge when the learning rate has be decreased to $\alpha = 0.0001$ and training takes 2400 epochs. To shows how batch normalize benefits related, they have trained a batch normalized network using a learning rate of $\alpha = 0.003$ and training takes 1320 epochs. These results further illustrated in Figure, the batch normalized network with low learning rate performs no better than an unnormalized networks. In addition, the batch normalized network with highest learning rate have the largest accuracy gap among the others.

i

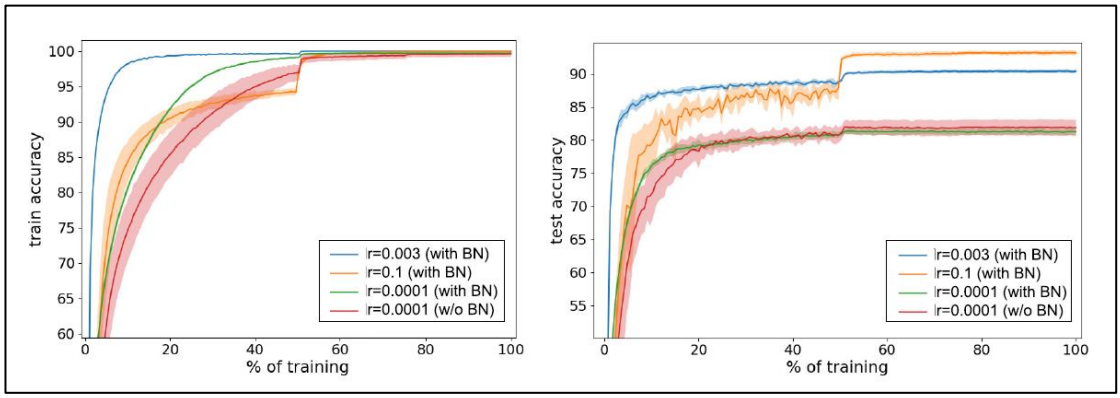


Figure 2.2: The training accuracy (left) and testing accuracy (right) through the training cycle. (Johan, et al., 2018)

The unnormalized networks tends to diverge for large rates in high learning rates and batch normalized networks perform well with higher learning rates. To investigate the reason behind, Johan, et al. (2018) have compared the gradient and distribution of comparable parameters between these two networks as shown in Figure. Through the histograms, the gradients are larger and distribution is heavy-tailed in unnormalized networks. Whereas for normalized networks, the gradients are centred around the mean and distribution is exponentially bounded.

2.6 The impact of batch size towards the performance of CNN.

Improving the performance of a deep CNN is a difficult works. Adjusting the training parameters is the domain ways to obtain a relatively well performing deep CNN. An experiment conducted by Radiuk (2017) to investigate whether the batch size value affect the network training progress and testing accuracy. The deep CNN used in this experiment is not as deep as the CNN used in the project. Besides the database used in this experiment is different with the datasets used in this project. The MNIST database contain only gray scales hand written images and another database is CIFAR-10 which contain color images with 10 different classes. Although the deep network and datasets used in this experiment is different with the project used, the network training performance depends on the batch size value would be similar.

Table 2.1: The result of network testing accuracy on MNIST and CIFAR-10 datasets with different batch size value. (Radiuk, 2017)

Batch size	Testing accuracy, %		Batch size	Testing accuracy, %	
	MNIST	CIFAR-10		MNIST	CIFAR-10
16	97.42	54.05	150	98.58	68.33
32	97.78	59.38	200	98.84	68.71
50	98.44	63.59	250	98.70	67.32
64	98.39	64.96	256	98.22	68.97
100	98.75	67.67	512	98.93	70,80
128	98.51	65.24	1024	99.11	71.53

From the Table 2.1, the batch size of 16, 32, 50 and 64 giving poor testing accuracy. However, an average testing accuracy results obtained in batch sizes of 100, 128, 150, 200, 250 and 256. The best testing accuracy are obtained from the batch size value of 512 and 1024. Radiuk (2017) had concluded that the larger the batch size value, the better the CNN network performing image recognition.

CHAPTER 3

RESEARCH METHODOLOGY

3.1 Software Installation

The Tensorflow GPU framework requires adequate libraries and drivers to support. The hardware and software requirements are stated below.

Hardware:

NVIDIA GPU card with minimum CUDA Compute Capability 3.5

Software:

- NVIDIA GPU drivers (CUDA 9.0 requries 384.x or higher)
- CUDA Toolkit (TensorFlow supports CUDA 9.0)
- cuDNN SDK
- Python 3.6
- Anaconda

3.1.1 Step for installing CUDA Toolkit

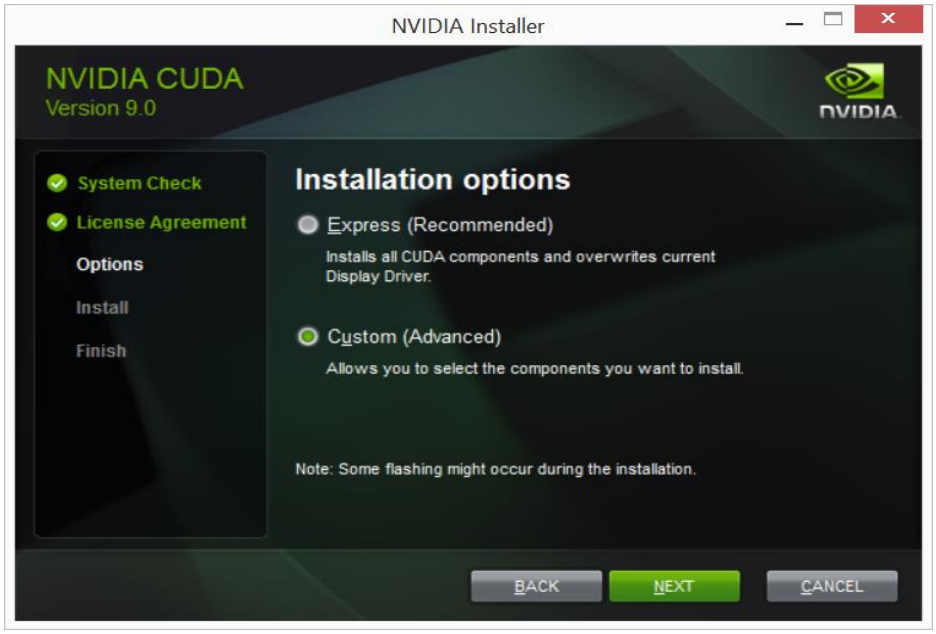


Figure 3.1: The CUDA v9.0 installation options

Figure 3.2: The CUDA v9.0 custom installation options

The CUDA Toolkit executive file was run immediately once the package was downloaded. The installation options are shown in Figure 3.1 and Figure 3.2. If the

CUDA toolkit is not allowed to install, make sure the installer version is compatible with pc platform. Another way would be re-installed or updated to latest NVIDIA driver.

3.1.2 Step for including cuDNN SDK into CUDA

The SDK file was downloaded through the link: https://developer.nvidia.com/rdp/cudnn-download. Choose a file compatible with the pc window OS. Sign up or login to membership of the NVIDIA Developer Program is required to get access to the pages.

After downloaded the file, following steps show how to build a cuDNN dependent program were gone through step by step:

1. The installed CUDA Toolkit directory path is referred to as : C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0

2. Navigate to the downloaded cuDNN directory and unzip the package.

3. Inside the package containing bin, include and lib file. Copy the files into the CUDA Toolkit directory accordingly:

    a) Copy cudnn64_7.dll in bin file to C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0\bin

    b) Copy cudnn.h in include file to C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0\include

    c) Copy cudnn.lin in lib file to C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0\lib\x64

3.1.3 Installing Python and Build tools

The minimum requirement for Python version and Build tools to support the Tensorflow GPU is provided as figure below. The Python version and build tools were installed via online. The required version to support tensorflow-gpu was followed from the list below.

Version	Python version	Compiler	Build tools	cuDNN	CUDA
tensorflow_gpu-1.12.0	3.5-3.6	MSVC 2015 update 3	Bazel 0.15.0	7	9
tensorflow_gpu-1.11.0	3.5-3.6	MSVC 2015 update 3	Bazel 0.15.0	7	9
tensorflow_gpu-1.10.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.9.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.8.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.7.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.6.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.5.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	7	9
tensorflow_gpu-1.4.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	6	8
tensorflow_gpu-1.3.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	6	8
tensorflow_gpu-1.2.0	3.5-3.6	MSVC 2015 update 3	Cmake v3.6.3	5.1	8
tensorflow_gpu-1.1.0	3.5	MSVC 2015 update 3	Cmake v3.6.3	5.1	8
tensorflow_gpu-1.0.0	3.5	MSVC 2015 update 3	Cmake v3.6.3	5.1	8

Figure 3.3: The require software and language version to support specific tensorflow-gpu version

3.1.4 Installing Anaconda

Anaconda provide a user-friendly distribution allows user to create a virtual environment and install packages needed for deep learning project. The Anaconda also come with conda, Jupyter Notebook and other open source packages. Conda is a package manager to manage virtual environment and allow user to install specific package version for project use without worrying about version conflicts.

The Anaconda package was downloaded from online and installed in the pc. Once successfully installed the packages, a virtual environment was created to store

the conda packages into target folder. Also, Tensorflow-gpu and some support packages were installed in the created virtual environment through the conda. Once the installation was done, some verify codes were run immediately to verify whether Tensorflow has recognize the pc graphic card. The terminal will prompt pc GPU information as shown in Figure when successfully installed the Tensorflow-gpu.

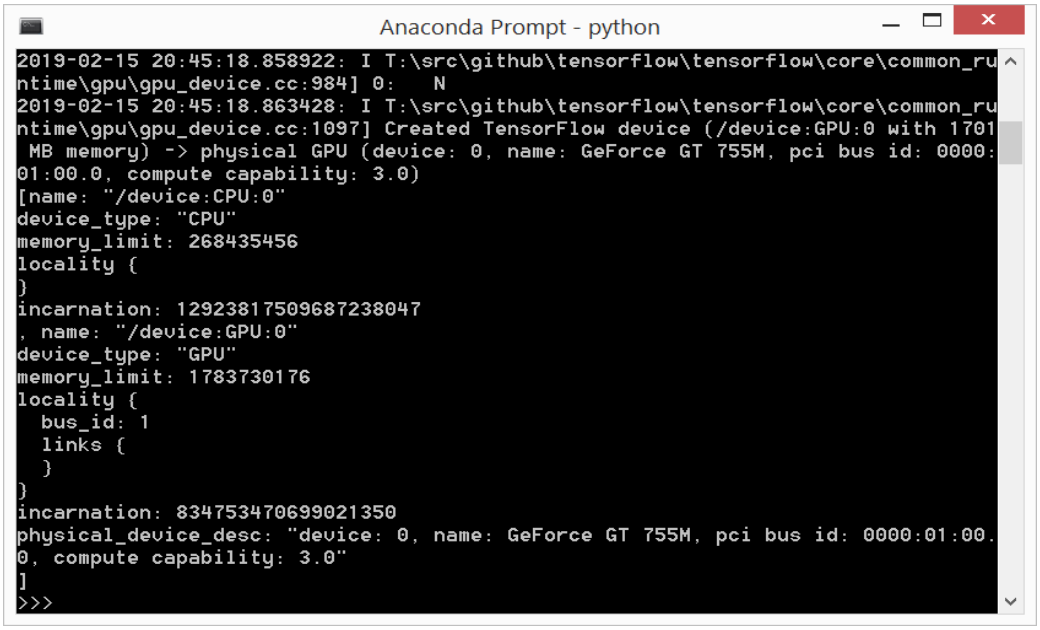


Figure 3.4: Anaconda terminal prompt the CPU and GPU information of the pc.

3.2 Dataset preparation

The first step to implement an object detection model is always acquire adequate datasets which are meaningful to the training and detection. In this project, the datasets not only an image itself, but the object annotations and labels are crucial for the neural network to learn the pattern of the detecting object and classifying them accordingly. Therefore, datasets creation must be carefully undertaken because the quality of the object detection is highly depending on the quality of the datasets provided.

Here are several methods of obtaining a dataset of images and annotation in this project:

Open source datasets: COCO, ImageNet and PascalVOC are providing huge amount of image and annotation for up to 80 classes. However, this project only detecting vehicle objects which are car, motorcycle and bicycle. Hence only required objects were extracted from the COCO datasets.

Filtering datasets: The datasets obtained from open source may not have captured properly and some of the annotations are missing or miss aligned. Those images which does not contain any related object in this detection were be removed and miss aligned object will be re-annotated.

Self-creating datasets: The datasets can be obtained through capturing real world vehicle objects. Varieties of object type were captured in multiple scenes with different angles and image resolution.

Dataset annotation: Image annotation is created by using image annotation tools to save the large volume of data stored after annotation. VOC Pascal format was used in this project as shows in Figure 3.5 below.

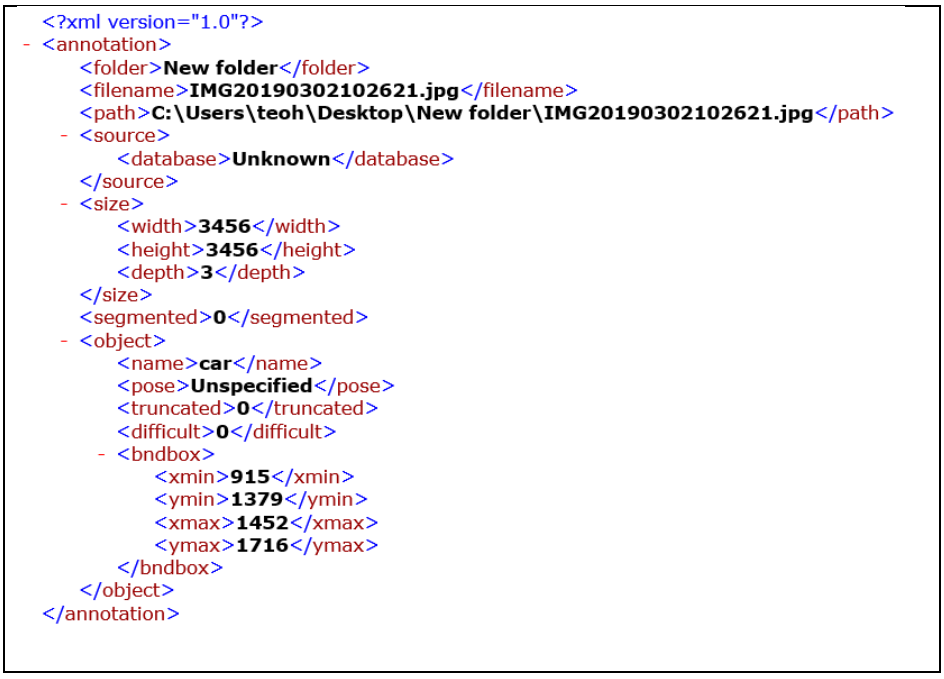


Figure 3.5: Image annotation information in Jason format

Dataset augmentation: Acquiring and annotating large number of images are time consuming and tiring, but not enough of datasets to train the deep learning network may lead to generalization issue. Dataset augmentation is a proven method to deal with the issue of limited amount of data. This technique generating a new training data from the existing dataset by applying image transformations, change lighting conditions or crop it differently.

The images and annotation files were collected by using the method mention above. There were only 3 classes contained in these images which were bicycle, car and motorcycle. Each of the class was split-ted equally among the total images. The total number of images used in this project were 7306 images for training and testing the model. These images then further split-ted into training images and testing images. The detail of these images is shown in Table below.

Table 3.1: The number of images used in training and testing process

Initial number of training images	Final number of training images	Number of testing images
6000	6656	650

3.3 Training process

Before starts to train the model, it is important to decide the model whether is trained in scratch or pre-trained weight. Model trained in scratch takes much longer times to get to recognize the dataset underlying pattern. Tone of training images needed and required a decent GPU to support the training process. However, the pre-trained weight had trained explicitly with tone of images by researcher, also known as fine-tuning. The second option usually bring much improvement for training process, hence, transfer learning was adopted in this project to train own dataset.

During the training process, some hyper-parameters will be varying according to the training results such as batch size, learning rate and number of epochs for the training.

Batch size is a number of images feed to the model and update the parameters once per iteration during training process. The relationship between epoch, iteration, batch size in training process are illustrated at Table 3.2 below. One epoch is when entire dataset has passed forward and backward through the model during training process. The iteration is the number of batches needed to complete one epoch.

Table 3.2: Relationship between Epoch, iteration and batch size.

Stage	1	2	3
Training images	6000	6000	6656
Epoch	1	1	1
Batch size	2	4	16
iteration	3000	1500	416

Learning rate is a hyper-parameter that controls the step for adjusting model's weights with respect the loss gradient. A low learning rate takes longer time to reach global minimum loss, but the loss converges steadily. A high learning rate takes shorter time to converge to lower loss, but the model loss may also bounce back to higher loss due to the model weights update step is large causes model miss from global minimal easily. Hence, choosing a right learning rate is based on trial and error. A low batch size needs a low learning rate to prevent from weight explode leads to high performance loss. In contrast, a large batch size can use a high learning rate to speed up the training process.

At initial training stage, the batch size was set to 4 dues to hardware constrain and trained for 100 epochs with total 6000 images. Then at the final training stage, the batch size was increased to 16 and continued training for 100 epochs with increase total training images to 6656 from initial 6000 images. The starting learning

rate was set to 0.001 and divided by 10 after epoch 30 and epoch 60 for both initial and final training stage.

3.4 Testing process

After certain epoch of training, the network will be evaluated its performance. A completely random images were input to the network to make the detection. These prediction and localization will be compared to the labels and ground truth provided by the annotations of testing set. Also, the model detection accuracy is evaluated using mean average precision metric (mAP).

3.5 Process flow

The whole project design and implementation flow are illustrated at the flow chart below:

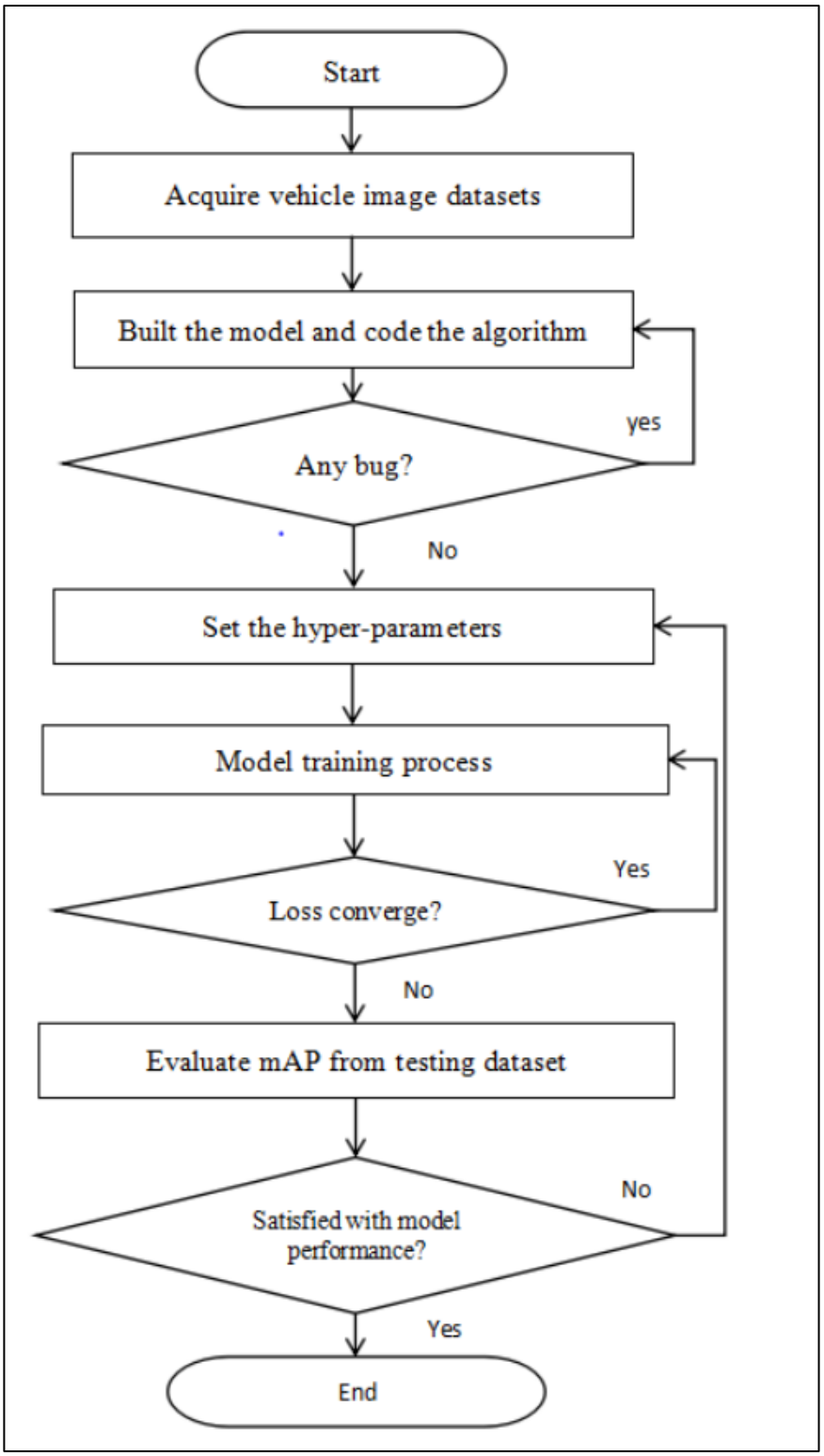


Figure 3.6: Whole project design and implement flow

CHAPTER 4

RESULT AND DISCUSSION

4.1 Model Architecture

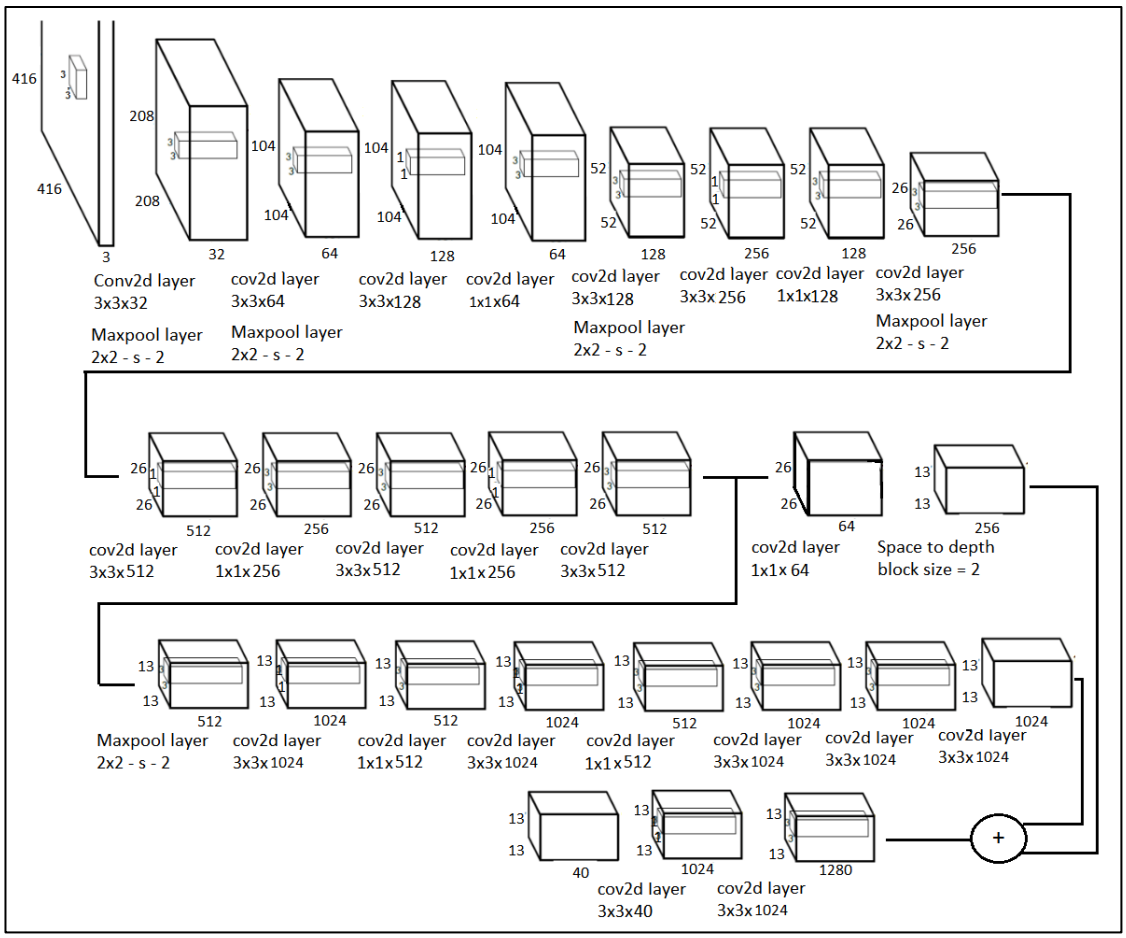


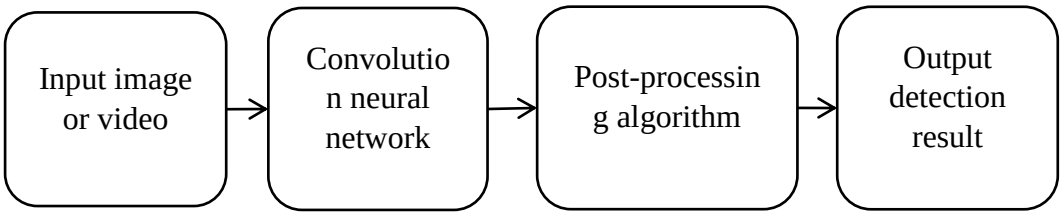
Figure 4.1: The model architecture (Darknet-23) demonstration. Total 23 convolution layer and 5 Max-pooling layers

The layout of convolution neural network used in this project is demonstrated in Figure 4.1 above. There are total 23 convolution layer where each of the layer will follow by a batch normalization layer and activation layer. The size of the filter window in these convolution layers are 3x3 and 1x1. At intermediate layer, a larger

output feature map will be concatenated with a smaller feature map to improve small object detection accuracy. Finally, 5 max-pooling layers are used in the model to reduce the input size from 416x416 to 13x13 at the output of the last convolution layer.

The flow of the detection system has shown in Figure 4.2 below. A random image or video with minimum 416x416x3 resolution will input to the model for making the prediction. After that, the output of the model is further post-processing and finally output the detection result which is understandable by human.

Figure 4.2: The flow of how the system perform vehicle detection



4.1.1 Convolutional layer

The convolution layer is the core layer of a convolution neural network that does most of the computational heavy lifting. Each of the convolution layer have a window filter which slide through entire images to multiply with the input image and output a feature map. Each of the window filter is responsible to extract different features from input feature map. Once a layer accurately recognizes those features, they’re fed to the next layer, which trains itself to recognize even more complex features.

The parameters of a convolution layer include: window size, depth, stride, zero-padding and filter quantity (Du, 2018). Window size and depth are the 3-dimension of the window filter. The window size is 3x3 or 1x1 and the window depth usually same as input depth. The 1x1 window size convolution mainly used to reduce input spatial dimensionality to remove computational bottlenecks and also limit the size of models (Szegedy, et al., 2014).

Stride is controlling the window filters shifting how many unit at a time. The window filters shift 1 unit per time in this model to convolve the input feature map in detail. Zero-padding pads zero values around the border of image. To ensure the spatial dimension of output convolution is remained as the input's spatial dimension after convolves by window filter, the zero-pad size is set to 1. The filter quantity is the number of filters to convolve the input data in a single convolution layer. Finally, the spatial dimension of the output convolution layer can be calculated using formula below (Du, 2018):

$$H_{out} = 1 + \frac{H_{in} + (2 \times p) - K_{height}}{s} \; ; \; W_{out} = 1 + \frac{W_{in} + (2 \times p) - K_{width}}{s} \qquad (4.1)$$

where
H_{out}, W_{out} = output spatial dimension
H_{in}, H_{out} = input spatial dimension
K_{height}, K_{width} = window filter size
p = zero - padding
s = stride

4.1.2 Batch Normalization Layer

Batch normalization layer can be treated as a data pre-processing layer which is applied after every convolution layer. The purpose of this layer is to normalize the input data by adjusting and scaling it to have a zero mean and unit variance (normal distribution). This will help to speed up convergence in result of greatly reduce the model's training time. Also, it makes the hidden layer continue learning on a stable input distributions of activation values that experience less internal covariate shift, hence, a dramatically acceleration on training process (Ioff and Szegedy, 2015).

Batch Normalization eliminates the need of dropout layer to prevent for model over-fitting and also allowing model train in much higher learning rates without being explicitly adjust the hyper-parameter and parameter initialization.

4.1.3 Activation layer

Activation layer is applied immediately after the batch normalization layer. This layer help enable model to learn from the complex arbitrary dataset and a non-linear complex function applied between the input signal and output signal. Non-linear functions are those which have higher degree and have a curvature properties when plotting these functions (Walia, 2017). Without applying the activation function in the convolution layer, the model would be simply a linear regression function which not powerful enough to learn from the complex dataset. The purpose of this layer is to make a non-linear result and set to solve the vanishing gradient problem. The vanishing gradient problem happens when the gradient is too small that will not update the weights and biases from last layers to initial layers effectively during back-propagation.

The activation function layer used in this model is Leaky ReLu. A comparison between ReLu and Leaky ReLu made by He, et al. (2015) stated that the Leaky ReLu improves overall training process with almost no computational cost. The ReLu may causes a dead neuron when a weight update during back propagation and Leaky ReLu solved this problem by introducing a small slope to prevent from dead neurons.

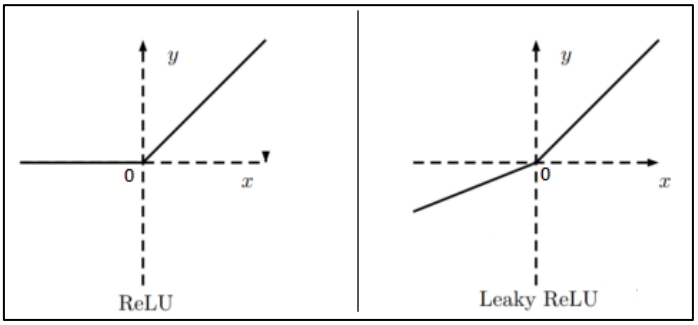


Figure 4.3: The activation function of ReLu and Leaky ReLu. (Xu, et al., 2015)

4.1.4 Pooling layer

Pooling layer down samples the dimension of the output from convolution layer. Max-pooling with window size 2x2 and stride of 2 is applied after some of the activation layer. It extracts highest value from pooling window when window slide through entire input feature map. This is because a high activation values from convolution layer is treated as meaningful feature and pooling layers filtered those low activation values to reduces the number of parameters and lessening the computational cost as well.

4.1.5 Fine-grained features

From the Figure 4.1 above, a 26x26x64 feature map from earlier layer is concatenated with 13x13x1024 feature maps producing a 13 x 13x1280 feature map. The purpose is to improve the model from detecting small object in a given image. A 13x13 resolution alone is sufficient in detecting large object but still a large room of improvement in detecting small object. Hence, earlier layer concatenates with later layer increase the mAP by 1% (Redmon and Farhadi, 2016).

4.2 Loss function

The purpose of loss function is measuring the quality of model's parameters (weight and bias) based on how well the induced scores matched with the ground truth labels in the training data. To compute the loss of the true positive, only the highest IoU among the all the prediction have made on an image will be selected and responsible to the object. This strategy improves the predictions among the bounding boxes at predicting certain sizes and aspect ratios.

YOLO architecture calculates the loss between the predictions and the ground truth by using sum-squared error method. Since vehicle detection involve

classification and localization, the loss function will involve classification loss, localization loss and confidence loss.

If an object appear and model successfully detected it in a grid cell, the classification loss for the particular cell is calculated as below:

$$\sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \tag{4.2}$$

where

$\mathbf{1}_i^{obj}$ is 1 when object detected in cell i, otherwise 0

$p_i(c)$ is the predicted conditional class probability for class c in cell i.

The localization loss measures the errors in the predicted boundary box coordinate and sizes. The measurement occurs when the bounding box in a cell is responsible for detecting the object:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2 \right] \tag{4.3}$$

where

$\mathbf{1}_i^{obj}$ is 1 when object detected in cell i, otherwise 0

x_i and y_i is the predicted coordinate for bounding box in cell i

w_i and h_i is the width and height of predicted boudning box in cell i

The bounding box width and height are squared root to reflect that the small deviation in large bounding boxes should matter less than in a small bounding box. The $\lambda cood$ is to emphasize the loss from bounding box prediction (Redmon et. al, 2016).

The confidence loss is measuring the objectness of the box:

$$\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_i^{obj}(C_i-\hat{C}_i)^2+\lambda_{noobj}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_i^{noobj}(C_i-\hat{C}_i)^2 \quad (4.4)$$

where

$\mathbf{1}_i^{obj}$ is 1 when object detected in cell i, otherwise 0

$\mathbf{1}_i^{noobj}$ is 1 when no object detected in cell i, otherwise 0

C_i is the predicted confidence score

$\hat{C}$ is IoU score between predicted bouding box and ground - truth

Most of the grid cell may not contain any object and this would cause the confidence loss imbalance. To solve this problem, the $\lambda noobj$ is applied to decrease the loss value in no object confidence loss (Redmon et. al, 2016).

Finally, all the losses combined and become the final loss function:

$$\begin{aligned}
&\lambda_{\text{coord}}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{\text{obj}}\left(x_i-\hat{x}_i\right)^2+\left(y_i-\hat{y}_i\right)^2\\
&+\lambda_{\text{coord}}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{\text{obj}}\left(\sqrt{w_i}-\sqrt{\hat{w}_i}\right)^2+\left(\sqrt{h_i}-\sqrt{\hat{h}_i}\right)^2\\
&+\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{\text{obj}}\left(C_i-\hat{C}_i\right)^2\\
&+\lambda_{\text{noobj}}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{\text{noobj}}\left(C_i-\hat{C}_i\right)^2\\
&+\sum_{i=0}^{S^2}\mathbb{1}_{i}^{\text{obj}}\sum_{c\in\text{classes}}\left(p_i(c)-\hat{p}_i(c)\right)^2
\end{aligned} \quad (4.5)$$

4.3 Optimizer

Loss function in previous section measuring how wrongly the prediction have made for model. High value of losses meaning the model is not predicting accurately and the root cause the wrongly prediction is model's parameters (weight and bias). The model's prediction improves when parameters able to process the input data and output a meaningful information which truly related to the objective task. Hence, these parameters have to be updated repeatedly throughout the training process in order to get to a lower loss value.

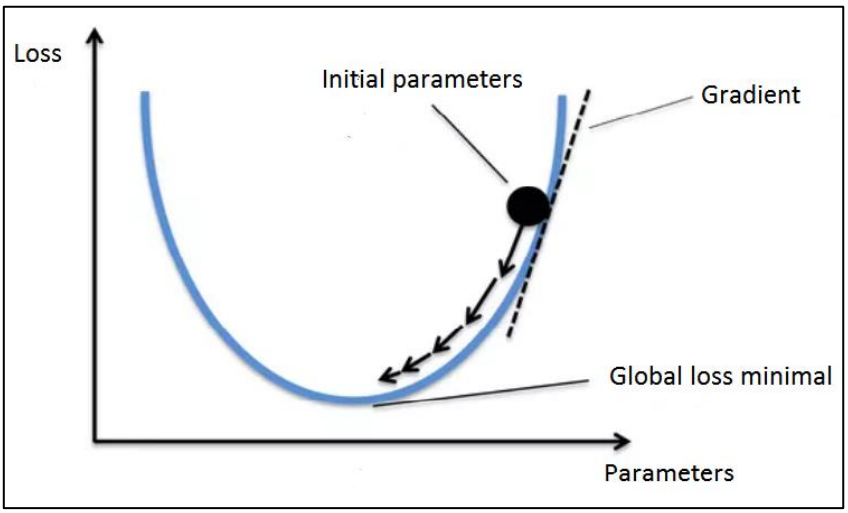


Figure 4.4: Initial parameters are optimized and tends to converge to global minimum loss. This method also known as gradient descent.

The goal of optimizer is to modified the parameters in order to minimizes the loss value. The optimizer computes the gradient of loss function, the procedure of evaluating the gradient repeatedly and performing a parameter update. The Adaptive Moment Estimation also called as Adam optimizer has been widely used in most of the deep learning architectures. Many experiments conducted by expert demonstrate that Adam works well in practice and outperforms other optimizers. Adam optimizer compute individual adaptive learning rates which provide heuristic approach without explicitly tuning hyper-parameters for the learning rate schedule manually. Moreover, Adam require lesser of memory to perform back-propagation for updating the parameters. Performing back-propagation almost double up the memory usage and this is the reason why so much memory needed during training.

4.4 Grid cell

After an input images propagate through the model, the input images is divided into S x S grid cells. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. The grid cells are demonstrated in Figure 4.5.

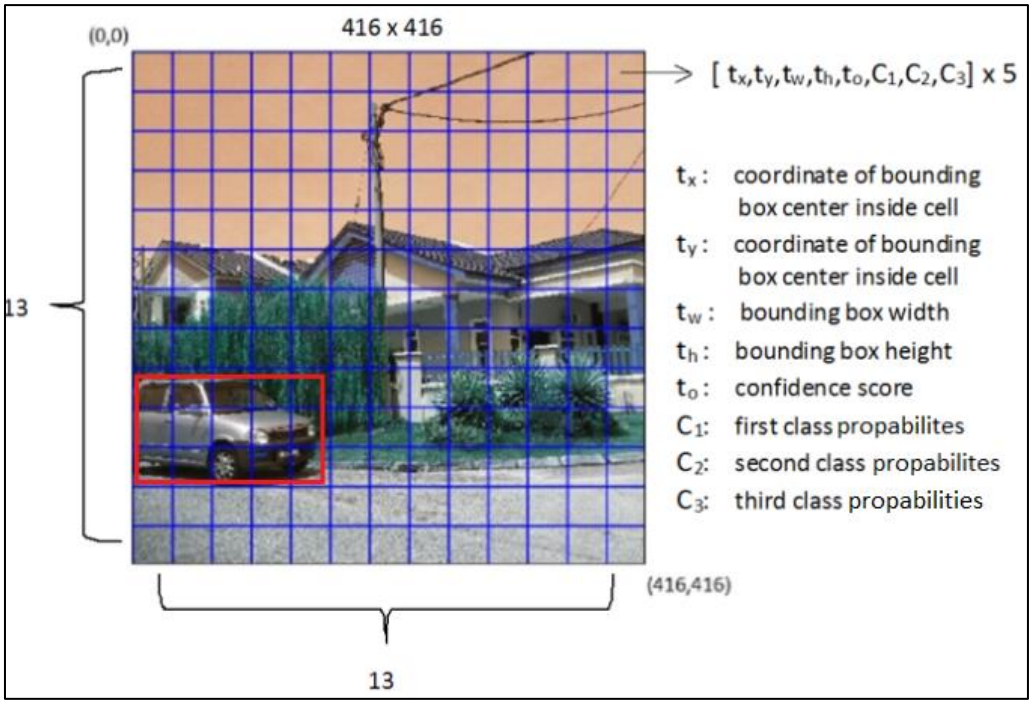


Figure 4.5: An illustration of 416 x 416 image divided into 13 x 13 grid cells and a ground truth box (red).

Basically, each grid cell consist 5 bounding boxes and each bounding box contain of 5 components: t_x, t_y, t_w, t_h and t_o. The (t_x, t_y) coordinates denote the center of the bounding box in a grid cell. The grid cell is not responsible for those centers of the bounding boxes are not fall inside it. The t_w and t_h are the width and height of the predicted bounding box respectively.

The confidence score, t_o indicates the confidence level of the model predict whether there is an object in particular grid cell and also reflects how accurate its prediction. The score should be 0 if there is no object appear in the cell (Redmon, et al., 2016). The IoU confidence score is calculated as equation below:

$$\Pr(object) x IoU_{truth}^{predict} \tag{4.6}$$

Logically, the model will give a confidence score zero if there is no object in that particular cell. Otherwise, the confidence score tends to equal to the IoU between the predicted box and the ground truth box.

Finally, each grid cell also predicts C conditional class probabilities, $\Pr(Class_i | Object)$. These probabilities are conditioned on the grid cell containing an object. Each bounding box predicts a set of class probabilities, C_1, C_2 and C_3. In this project only predicts 3 classes which are bicycle, car and motorcycle. If there is an object predicted by model in the cell, the model will also predict which class the object belongs to. The class probabilities score indicates how confident the model classifies the object into particular class, from 0 the least confident to 1 the most confident.

4.5 Anchor box

Anchor Box is a predefined box used to make model training more stable by fixing the shapes and sizes of bounding boxes. Instead of directly predicting the anchor box size, the anchor box is determined via k-Means clustering on given dataset. After generating anchor boxes using K-Means clustering, it turns out that most bounding boxes have certain height-width ratios for certain objects. For example, as shown in Figure 4.6 below, car have a horizontal bounding box, whereas, person on bicycle has a vertical bounding box.

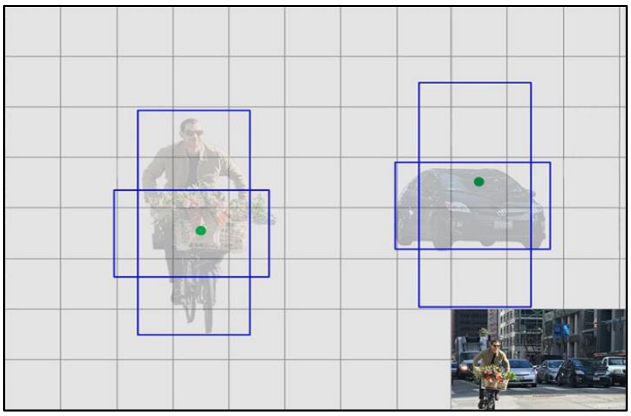


Figure 4.6: An example of diverse guesses for real-life objects. (Hui, 2018)

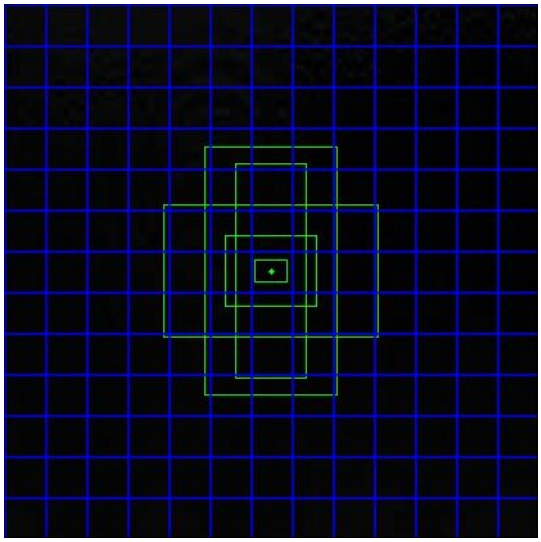


Figure 4.7: Predefined 5 anchor box with different size at center of a cell.

Instead of predicting arbitrary boundary boxes, this project predicts offsets to each of the anchor boxes. As illustrated in Figure 4.7 above, each anchor box is fixed for particular aspect ratio and size. The location of anchor boxes is the center of the cell in which the center of the ground truth box falls (Yadav, 2017). The ground-truth box and anchor box usually are highly overlap to each other due to the size of anchor box is generated based on the mean size of the object in the given dataset. Hence, the predicted bounding box with highest IoU between the ground-truth box and anchor box will be selected as appropriate bounding box.

4.6 Direct location prediction

During testing stage, the model predicts 5 bounding boxes at each cell in the output of last convolution layer feature map. As mention in grid cell section, the model predicts 5 coordinates for each bounding box, t_x, t_y, t_w, t_h, and t_o. The values of these 5 coordinates values are meaningful for model but not the actual results yet. The center coordinates (t_x, t_y) applies to activation function to constrain the model's predictions to fall between 0 and 1 so that the center coordinates are bounded within respective cell. The actual bounding box coordinates calculation is shown in Figure 4.8 below.

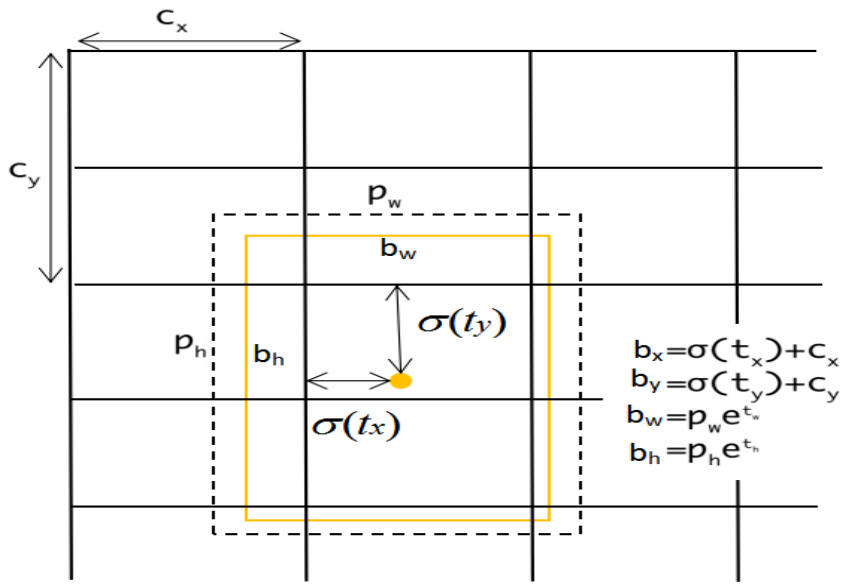


Figure 4.8: Predicted bounding box size adjusted through anchor box to form an actual bounding box (yellow). (Redmon, et al., 2016)

The (c_x, c_y) is the distance margin from top left corner of 1st grid cell to the top left corner of target cell and (p_w, p_h) is the width and height of anchor box. The width and height of actual bounding box are adjusted nearest to size of anchor boxes. The size of the anchor boxes is the size with most of the object going to be detected. Thus, the model is not directly predicting the final size of the bounding box, but adjusted to more similar to the object size which improve overall detection performance.

4.7 Intersection over Union (IoU)

Intersection of Union is a metric for comparing the overlapping between two arbitrary shapes. It is used to measure how accurate an object detection base on the provided dataset. To calculate the IoU ratio, a ground-truth bounding boxes (drawn by hand) and predicted bounding boxes from output model are needed. These two bounding boxes will be used to determine the IoU ratio via:

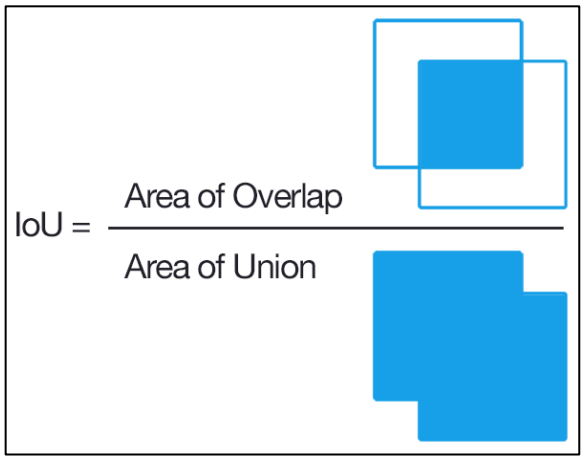


Figure 4.9: Computing the IoU by calculate the overlap area of two bounding boxes, and divides it by union area of two bounding boxes. (Rosebrock, 2016)

In the IoU equation, the numerator is the area of overlap between ground-truth bounding boxes and predicted bounding boxes. The denominator is the area of union by both the ground-truth bounding boxes and predicted bounding boxes. The area of overlap divided by the area of union yields output Intersection over Union (IoU).

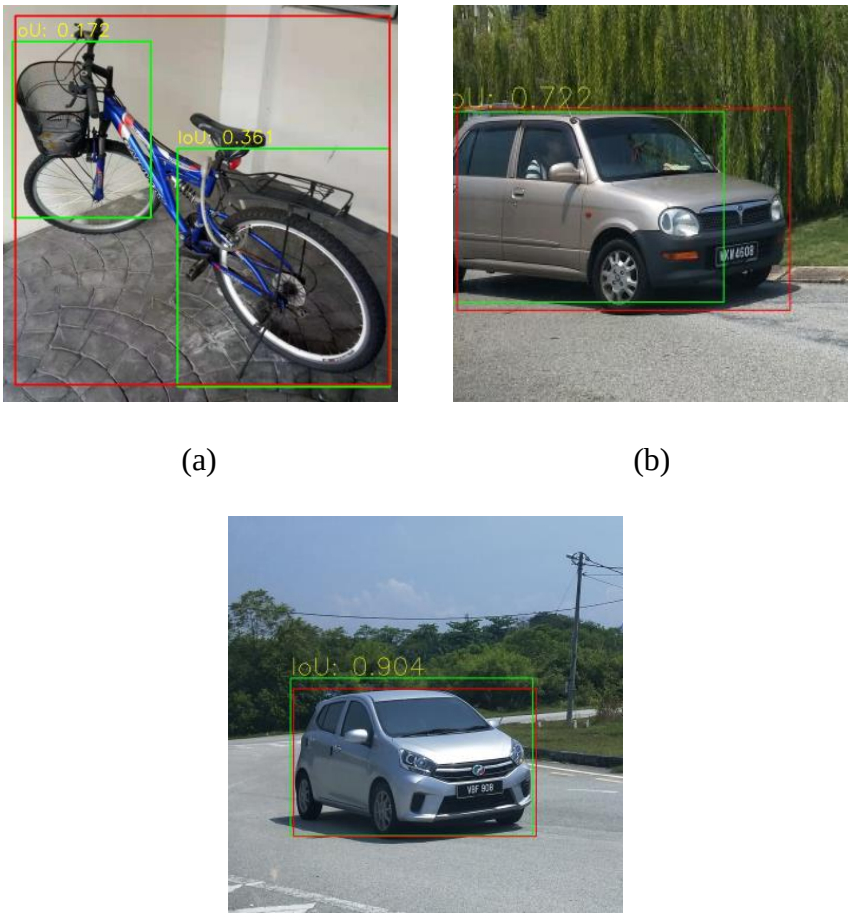
$$\text{Area of Overlap} = A \cap B \tag{4.7}$$

$$\text{Area of Union} = A \cup B \tag{4.8}$$

$$\text{IoU} = \frac{A \cap B}{A \cup B} \tag{4.9}$$

where
$A = \text{Ground - truth bounding box}$
$B = \text{Predicted bounding box}$
$IoU = \text{Intersection of Union}$

The IoU is a number between 0 and 1, if IoU score approaches to 1 meaning that the size and location of predicted bounding box tends to similar with ground truth hence better prediction has made by the model. In training process, if the IoU score is below threshold value 0.6 will penalize the network and confidence of the boxes with low IoU. Ideally, the predicted box and the ground-truth have an IoU score of 1 but in practice anything over 0.5 is considered as a good prediction (Rosebrock, 2016). For example, the Figure shown below demonstrate a ground-truth (red) along with the predicted bounding boxes (green) from the custom object.



(a) (b)

(c)

Figure 4.10: Computing IoU score with different bounding boxes (a) poor IoU score (b) good IoU score (c) excellent IoU score

In Figure 4.10 (a) two predicted box with IoU score 0.172 and 0.361 are covered part of the object which fairly localize the whole object as one prediction and each of the predicted box (green) is not large enough to cover the ground truth

bounding box (red). However, in Figure 4.10 (b) the predicted box with IoU score 0.722 is bounded within the ground truth, but predicted box not wider enough to cover front part of the object. In Figure 4.10 (c), the predicted box with IoU score 0.904 is entirely cover the object and almost fully overlaps with the ground-truth bounding box.

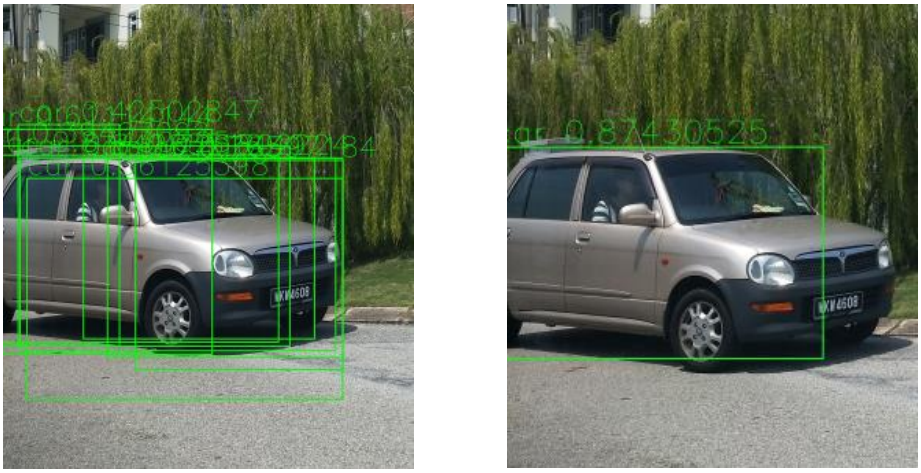
4.8 Non-max suppression (NMS)

NMS is post-processing algorithms to filter redundant bounding boxes. NMS makes selections based on the Intersection over Union (IoU) between detection boxes. As mentioned previously, IoU measures the ratio of the overlapped area over the union area between two boxes. NMS can be simplified into two steps (Hosang, et al., 2017):

1) In the same vehicle category, all the predicted bounding boxes are sorted based on their box confidence scores from highest to lowest.

2) NMS selects the box which has the highest box confidence score as the reference box, and then it discards other predicted bounding boxes whose IoU value with the reference box is beyond the threshold value.

Then, NMS repeats the above two steps for the remaining predicted boxes until the last predicted bounding box in the sorted list.



(a) (b)

Figure 4.11: The output of model prediction (a) without NMS (b) after NMS

From the Figure 4.11 (a) shown above, multiple bounding boxes prompted around the vehicle and each of the bounding box correspond to a confidence score. Each of the bounding boxes giving true positive result meaning that the model is working as expected, but only one bounding box is needed to localize the vehicle. After implementing the non-max suppression algorithm, the result shown in Figure 4.11 (b) has successfully eliminate the redundant bounding boxes.

The NMS algorithm also called as Greedy NMS because algorithm greedily selects bounding box with high confidence score and eliminate close-by less confidence score neighbors since they are likely to cover the same category. The ideal result from this algorithm if:

1) Bounding boxes with high confidence score triggered by same object will always be suppressed.

2) Bounding boxes with high confidence score of the next closest object will never be suppressed.

The greedy NMS usually works well when objects are far apart, but there is a tension for suppression when objects with same category are too close to each other or crowded scenes (Hosang, 2017). In other words, one object per image

post-processing the NMS problem is insignificant, but nearby objects require a better NMS algorithm.

(a) (b)

Figure 4.12: Different size of bounding boxes triggered by objects (a) before suppression and (b) after suppression

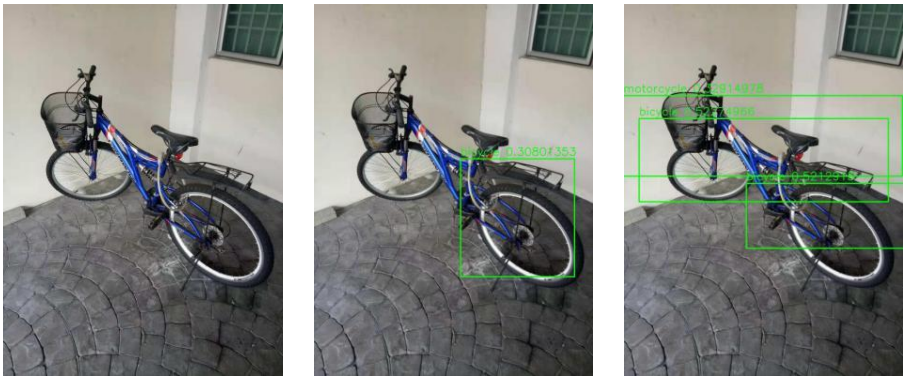
The problem mention above is demonstrated in Figure 4.12. From the Figure 4.12 (a), the model was successfully predicted multiple motorcycles and different size of bounding boxes was covering the motorcycles correspondingly. After post-processing it as shown in Figure 4.12 (b), the motorcycles with far apart from the other two is correctly bounded with a bounding box, but the other two motorcycles which are close to each other is simply bounded with one box. This is because NMS has taking the bounding boxes with high confidence score and eliminating the neighbor bounding box which are exceed the suppression threshold.

4.9 Real life vehicle detection and observation

(a) (b) (c)

Figure 4.13: The result of model's output detection on housing area image at (a) epoch 10 (b) epoch 100 and (c) epoch 250

From Figure 4.13 above, the model started to detect the cars in the given image while training process goes through. Initially, the model unable to detect any cars inside the image. After few hundreds of training, the model starts to detect some of the cars inside the image which is shown in Figure 4.13 (b), but the white car on the left still unable to detect it even though the white car is occupy large area in the image. Observing in Figure 4.13 (c), the model detected most of the car inside the image and the size of bounding boxes are bounded with the cars. Some of the bounding box does not match with exact size and location, but the overall performance is quite accurate.



(a) (b) (c)

Figure 4.14: The result of model's detection on a bicycle image at (a) epoch 10 (b) epoch 100 and (c) epoch 250

The model also tested with only one bicycle in an image. From the Figure 4.14 (a) above, unsurprisingly the model is not able to detect bicycle inside the image and none of the bounding box appear in image. In Figure 4.14 (b), the model able to recognize small part of the target object, but detection is not accurate enough to detect a complete object. In Figure 4.14 (c), the model is starting to detect most part of the object compares with Figure 4.14 (b), but the bounding box is not confidence and large enough to cover the target object into one box. The overall performance for detecting one bicycle in this image is poor.

(a) (b)

(c)

Figure 4.15: The result of model's output detection on traffic image at (a) epoch 10 (b) epoch 100 and (c) epoch 250

The model also tested in real life traffic road. The model started to detect some of the vehicles at training epoch 100. From Figure 4.15 (b), most of the car which are very obvious on the image are able to detect correctly, but the motorcycle

on the right most is not classify correctly and motor on the left is not detected. At left of the image is covered with multiple bounding boxes indicate that the model is not confidence enough to localize the vehicle. Also, the motorcycle is wrongly classified as car although the size of the bounding box is perfectly bounded with the motorcycle. After few more hundred epoch of training, the model detection result shows in Figure 4.15 (c) is look more pleasant compare with Figure 4.15 (b). All the vehicles appear inside the image are correctly classified and localized.

4.10 Training loss performance

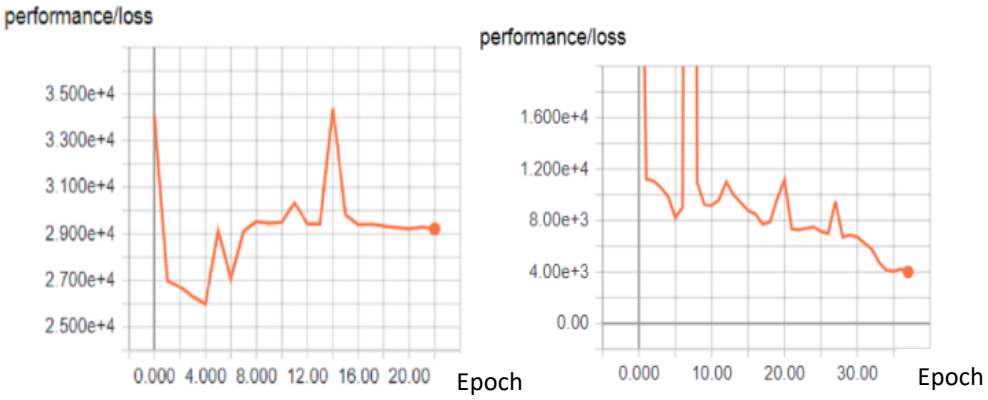


Figure 4.16: The graph of training loss against the epohcs with (a) 2 batch size and (b) 8 batch size.

Initially, the model was trained on a graphic card GT755m 2GB vram can process 2 images per iteration. The model was let trained for a day and the training performance is shown in Figure 4.16 (a). From the graph, the loss is converge for early epoch, but bouncing back to higher loss and hardly to converge to smaller loss for rest of the epoch. As illustrated from this graph, the model found very hard to learn from such small batch size especially at a deep model that contain 23 convolution layers. In this case, the model may take longer time to learn from all data given or in worst case the model may not able to learn from the dataset no matter how many epochs have gone through.

The second model had trained on a graphic card GTX1050 4gb vram can process 4 images per iteration. The model was let trained for few hours and the training performance is shown in Figure 4.16 (b). From the graph, the overall loss slowly converges though the training process. There is a sudden grow of loss in epoch 8, but return to normal at following epoch. Comparing both graphs above, the model has a stable loss converge and better training performance after increasing the batch size. Besides, a better GPU with larger virtual ram also stand an important factor in quality training process and greatly reducing the time required to train a model.

Since the result shown in Figure 4.16 (b) is in correct trend, the training process was continue running for 2 days until reach the final epoch 100 which was set initially. The complete loss performance graph has shown in the Figure 4.17 below:

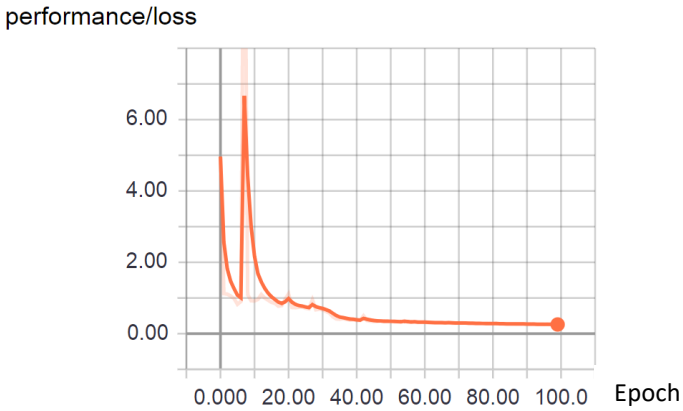


Figure 4.17: The performance loss graph for 100 epochs (noted that the loss values had converted to average loss)

From the graph in Figure 4.17, the loss barely converges and the line moving steadily after epoch 40. At epoch 100, the weight parameters were saved and had been evaluated the performance. The evaluation result (shown in Figure 4.19 at epoch 100th) is not satisfied, hence, a lower loss is required in this case meaning that a larger batch size and a decent GPU are necessary for better training performance.

This time, the model was trained on a graphic card RTX2040 with 8gb vram. The weight files obtained initially was loaded and further trained in this training

process. The batch size was increased to 16 images per iteration and learning rate was fixed at 0.0001. Another reason that may cause low mAP performance is the number of images to train the model is not enough, hence, the total number of images was increased from 6000 images to 6656 images. After 13 hours of training, the model was trained in 100 epoch and the loss performance graph has shown below:

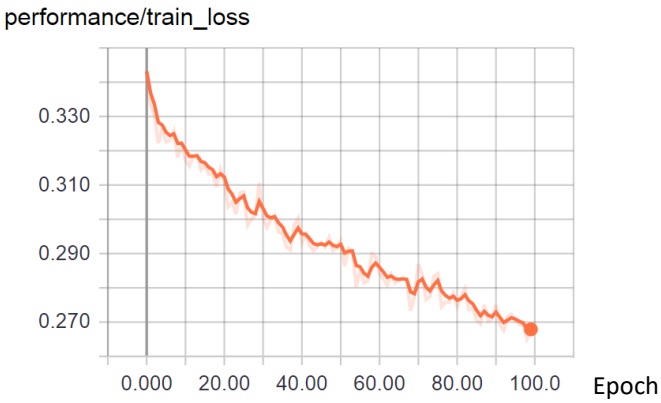


Figure 4.18: The performance loss graph for 100 epochs with increase batch size, number of training images and learning rate.

From the graph shown in Figure 4.18, the loss converged slowly through the epochs. The moving pattern of the graph showing that the loss can be further converged as number of training epoch extended. To prevent from over fitting issues, the weight parameters at epoch 50 (epoch 150 at graph 4.19) and epoch 100 (epoch 200 at graph 4.19) were used to evaluate mAP. The mAP obtained for epoch 50 and epoch 100 are 0.3, 0.23 and 0.39, 0.32 for IoU = 0.3, 0.5 respectively. The mAP has significant grow compared with previous training process. From the observation, a larger batch size can help the model converge to a lower loss. Also, the more training images feed to model for training, the better the detection performance. Most importantly, a better GPU is benefits from the training speed and training quality.

4.11 Model accuracy evaluation

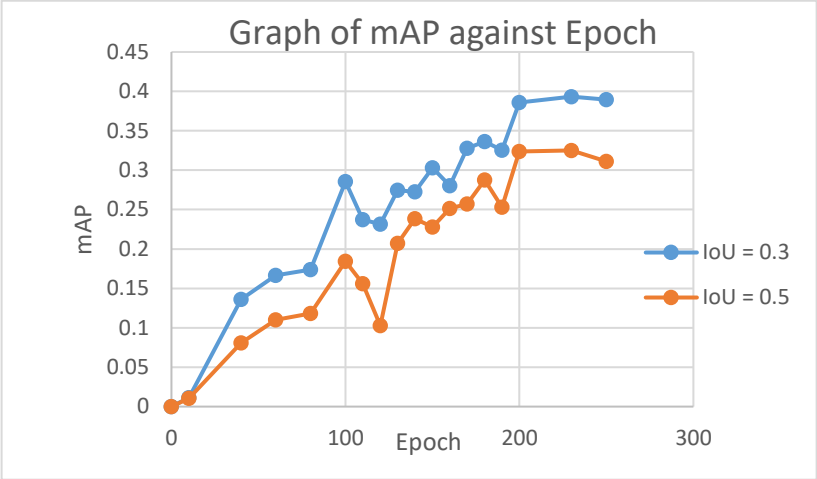


Figure 4.19: mAP evaluation for 250 epochs

Mean Average Precision (mAP) is a metric in measuring the detection accuracy of an object detection system. Since the object detection involves in classification and localization, hence, correctly classified the target object and bounding box overlap with ground truth exceed predefined IoU threshold value will improve the overall mAP values. Similarly, target object wrongly classified or below IoU threshold or miss detection will decrease the mAP as well. The higher the mAP value, the better the detection performance.

From the mAP graph in Figure 4.19, the mAP improved as training goes through hundreds of epochs. The highest mAP obtained are 0.39, 0.33 for IoU 0.3 and 0.5 respectively. Again, from the graph clearly shows that IoU=0.3 has higher mAP than IoU=0.5 because a smaller box which poorly overlap with the ground truth boxes also considered as correct detection as long as correctly classify the particular object (A clear explanation about IoU in Section 4.7). The mAP improvement with IoU=0.3 may interpret that the model has successfully detected there is an object appear in the input image and classified it correctly, but the model localization is not fine enough to cover the entire object into one box.

The overlapping area between predicted bounding boxes and ground-truth which have IoU score greater or equal to 0.5 are considered as good detection as

mention in previous section. Hence, improve the mAP with IoU=0.5 set as a destination to obtained a good object detection system.

4.12 Loss performance with different larger batch size

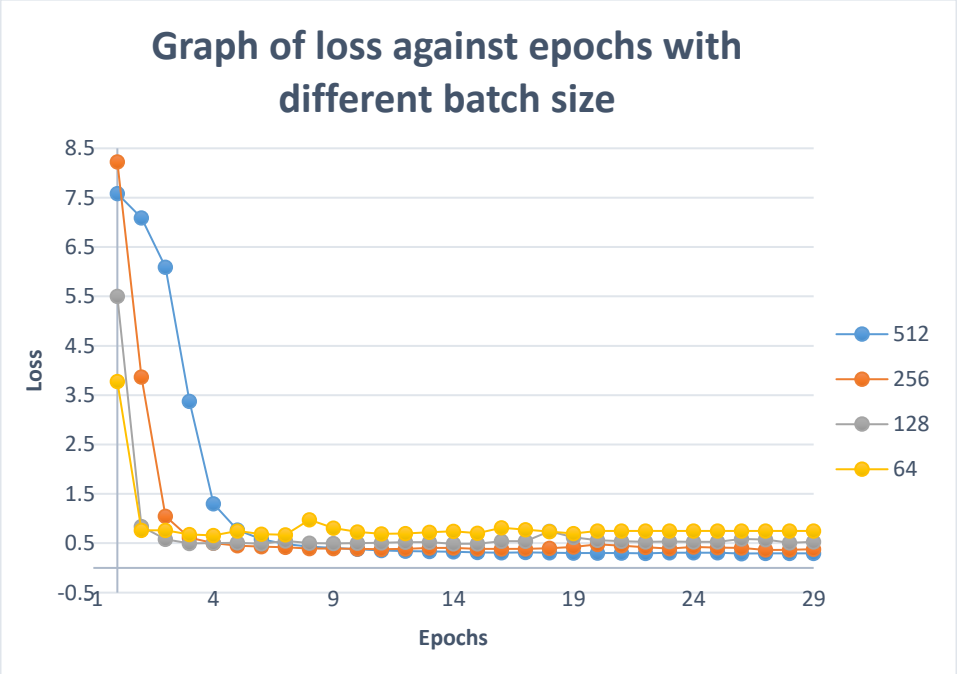


Figure 4.20: The model loss performance with different batch size for 30 epochs and fixed learning rate at 0.001.

To study how the batch size affect model training loss performance with only 6656 images. The experiment was conducted with different batch size which are 64, 128, 256 and 512. The training process was fixed at high learning rate 0.001. The model was trained on graphic card Tesla K80 with 11 GB vram (free cloud GPU from Google Colab), but the GPU only available continuous running for 12 hours and will be force-disconnected by system after the time constrain. Due to the limited time resources to train on a free GPU, the training epoch was set to 30 epochs instead of 100 epochs. The purpose of this experiment is to figure out the effect of batch size at high learning rate on training performance for current number of images.

From Figure 4.20, the loss tends to decrease gradually with a larger batch size and a steeper decrease with a smaller batch size at the beginning of training epochs. This implies that a smaller batch size saturate faster than larger batch size. After a few epochs of training, the loss decreased slowly and steadily for batch size 256 and 512, but small fluctuate without clear decreasing sign for batch size 64 and 128. This can be further encoding that; larger batch size guarantees a lower loss or minimal optima compare with smaller batch size.

The experiment shows that using a smaller batch sizes have faster convergence to "good" solutions. A smaller batch size will update model's weight parameters more frequently and this intuitively explained by the fact that the model start learning vigorously before seeing entire data. The disadvantages model trained in smaller batch is not guaranteed to reach global optima (lowest loss with highest test accuracy). Also, using too small batch size will apply too much noise during model training process leads to poor model performance (As proven in Section 4.10 with batch size 2 and 4). Many researchers usually use a larger batch size to train their model as it allows computational speedups from multiple GPUs in parallel. However, it is well known that too large of a batch size will lead to poor generalization (cannot adapt new data properly).

From the investigation above, it is not easy to find a optimal batch size to obtain a well performing model. The batch size also largely depending on the number of images used to train the model. Huge number of images hardly leads to poor generalization since variety of samples can be observed by the model, hence, a larger batch size would do a better job in this case. The dataset amount used in this project is incomparable with the dataset amount used by researcher and developer. They trained the model in millions of images which allowed them to use a larger batch size yet hardly over-fitting the model cause poor generalization.

An experiment conducted by Smith, et al. (2018) states the model benefits from increasing batch size during training instead of decaying the learning rate. Also, Hoffer, Hubara and Soudry (2018) claim that initial high-learning rate training helps model find wider local minimal and no inherent generalization issue with training using large batch size. If without computational constrain and time constrain, it is

advisable to train the model starting with small batch size and varies the batch size with steady grows through training process. At the same times, evaluate whether the model detection performance is improved from guaranteed loss converge as the batch size increase.

4.13 Convolutional layer output evaluation

Other than focusing on loss performance or mAP accuracy graph analysis, there are so much more about CNN can be visualized to help for better understanding. Visualizing output of each convolution layer through the training process gain a better insight of CNN.

For instance, the following Figures showing the output of intermediate layer of trained model when provided multiple vehicles image in Figure 4.21.

Figure 4.21: Original 480x640x3 resolution image before propagate through the model.

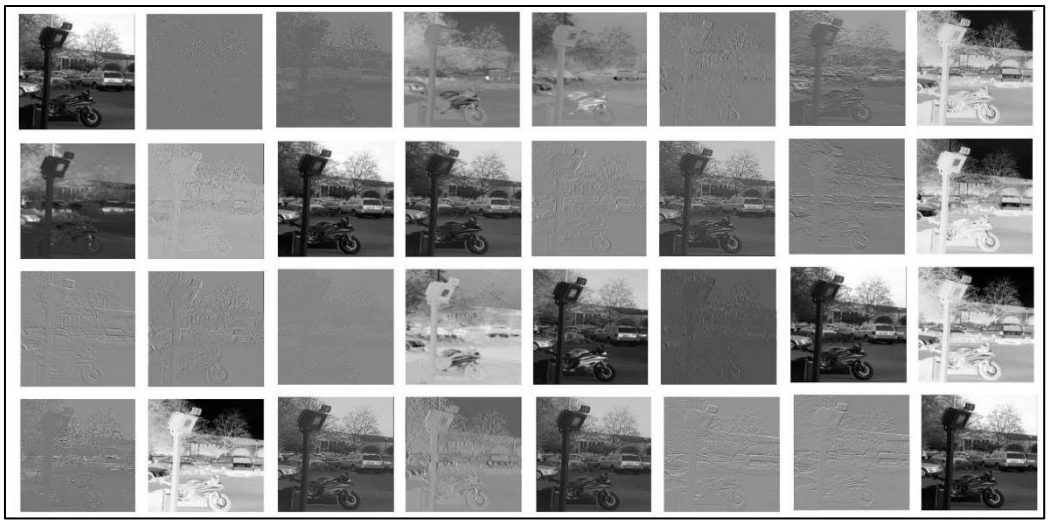


Figure 4.22: Total 32 output feature map of 1st convolution layer with 416x416 resolution each.

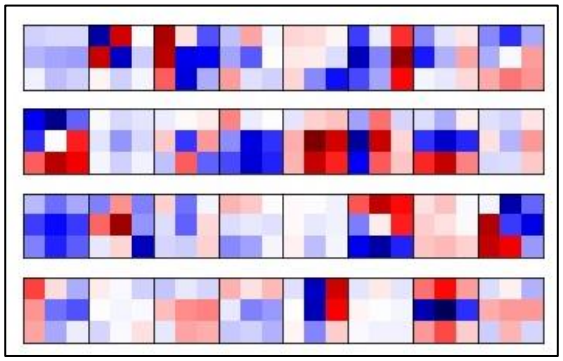


Figure 4.23: 3x3 filters for each of the feature map respectively in 1st convolution layer. (weight value is interpreted in color intensity so that can be easily compared, positive weights are red and negative weights are blue)

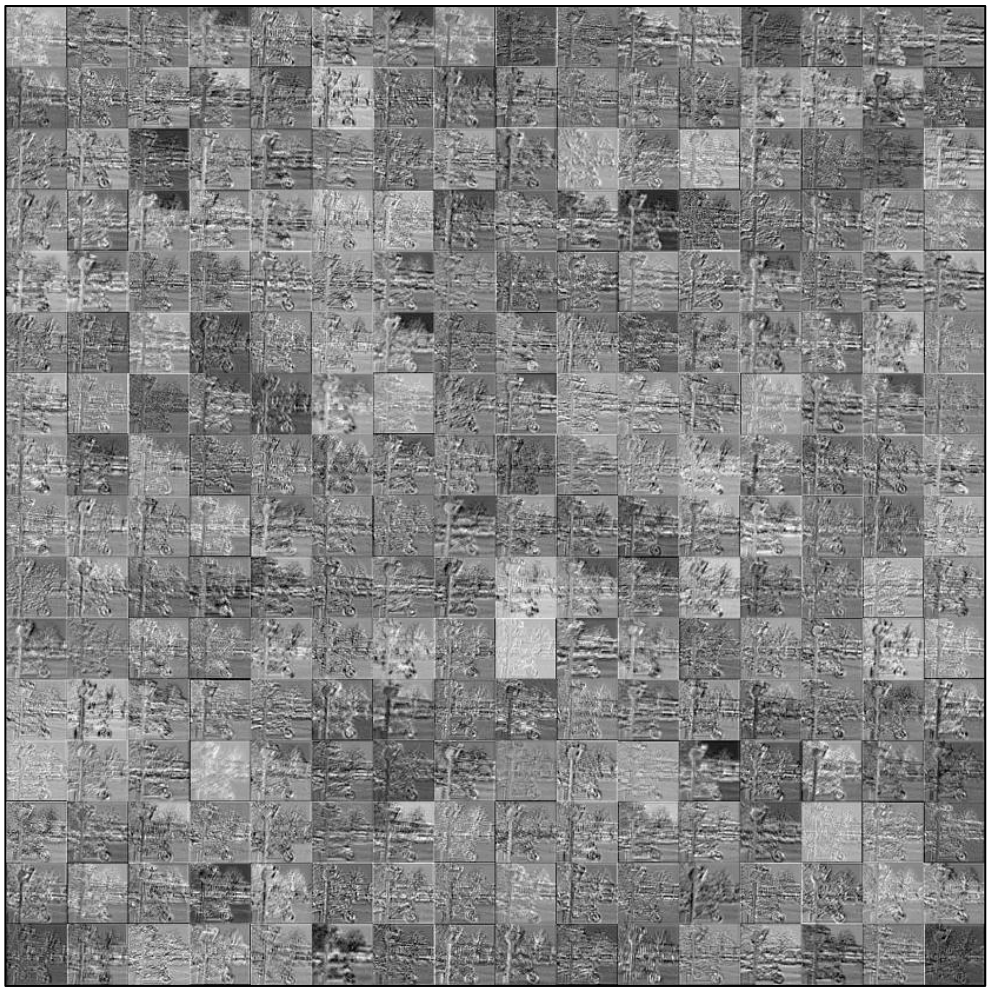


Figure 4.24: Total 256 output feature map of 6th convolution layer with 52x52 resolution each.

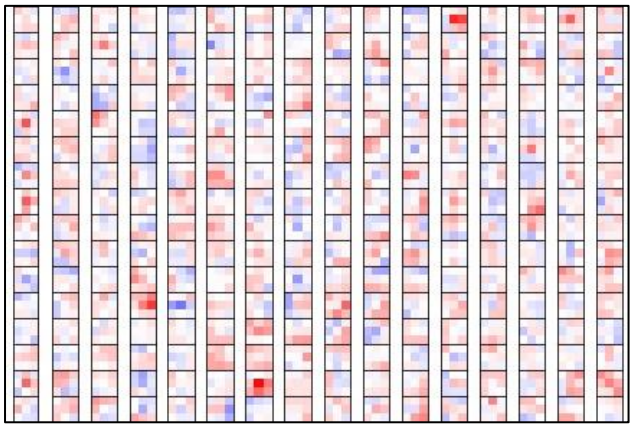


Figure 4.25: 3x3 filters for each of the feature map respectively in 6th convolution layer.

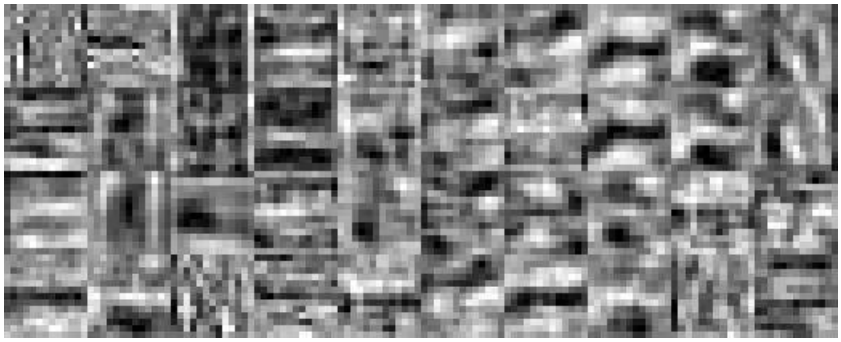


Figure 4.26: Total 40 output feature map of last convolution layer with 13x13 resolution each.

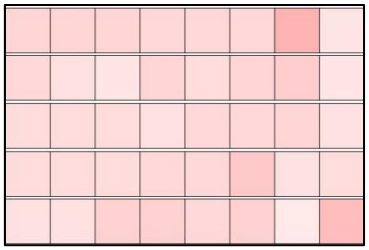


Figure 4.27: 1x1 filters for each of the feature map respectively in last convolution layer.

The output of first convolution layer shown in Figure 4.22 retaining most of the information of the original image. The features that are important for discrimination are usually retained from higher layers of CNN, while irrelevant variations are suppressed. Each of the filter in Figure 4.23 containing 3x3 weight parameter values and each responsible to slide through the entire original image and produce feature maps. Each of the filter extract different features from original image as the result each of the feature map is different to each other.

As the feature map propagates to deeper layers, the output become more abstract and less visually understandable as shown in Figure 4.24. This is due to the combination of convolution layer, activation layer and polling layer have extracted, activated and dropped the input feature map and output to another layer. Hence, the deeper the layer, the less information contain about the image, and increasingly more information which understand by the model.

Finally, at the last layer as shown in Figure 4.26, the output feature map completely distorted by model. This information will be post-processed before presented to human use.

4.14 Benchmark GPU

A fast GPU allowing for rapid gain in practical experience of deep learning which help in responds to new problem arise quickly or monitoring model's learning condition while training. Without these rapid feedbacks, it would take long time to realize a mistake or do some changes based on the result. In long term training and evaluating, it can be discouraging and frustrating to continue with deep learning. To emphasize how important a fast GPU for doing deep learning, table below shows the comparison between different GPU's performance for training a deep CNN.

Table 4.1: GPU speed comparison in term of total time usage to complete a training.

GPUs	vram(GB)	batch size	Total images	1 epoch	100 epochs	image /min	image /sec
GT 755M	2	2	6000	80 min	8000 min	75	1
GTX1050	4	4	6000	39 min	3239 min	153	2
Tesla K80	12	16	6656	14 min	1410 min	475	7
RTX2040	8	16	6656	8 min	831 min	832	13

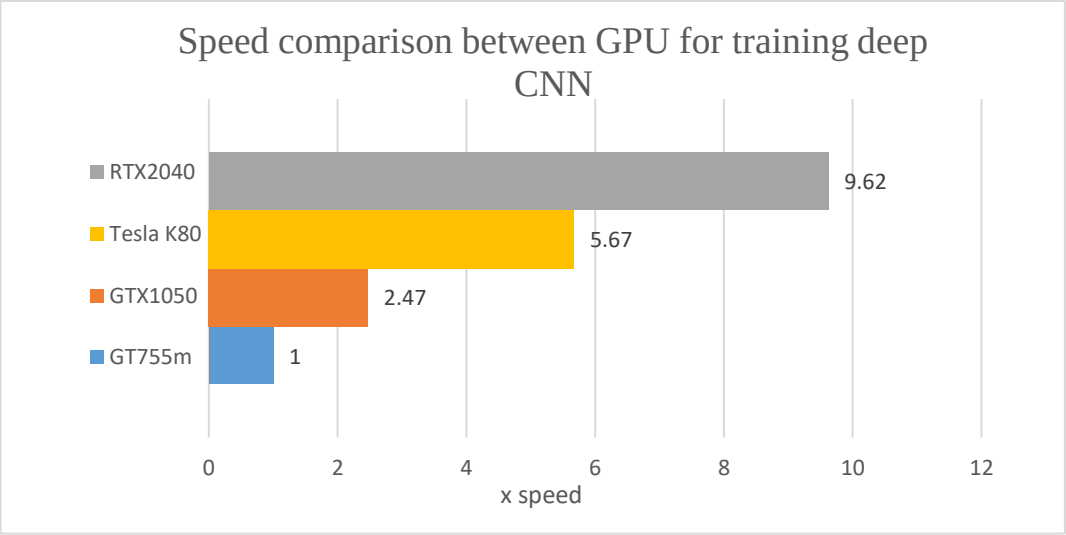


Figure 4.28: Speed comparison between GPUs had been used for training the model (Takes GT755m as reference).

From Table 4.1, a normal GPU GT 755M estimated takes 5 days and 12 hours to complete 100 epoch training. Also, the GPU's virtual memory is not enough room for a higher batch size other than maximum of 2 batch size. The batch size is too small for making the model's loss converge, in fact, the model found very hard from learning the dataset. However, a gaming GPU GTX1050 setting as the minimum requirement to train a deep learning algorithm. Although the time usage 2 days and 6 hours is considered long training time period, the batch size of 4 just right enough to makes the model's loss converge. Google provides a free cloud GPU for any user to do the research in deep learning. The cloud GPU Tesla k80 provide a fairly fast enough speed for deep learning learner to run their algorithm. The downside of cloud GPU is maximum access time connected to particular GPU is 12 hours and user have to wait for next access time after force disconnected by system. Hence, it is very hard to do any real time training evaluation without interruption. Lastly, a decent GPU RTX 2040 showing a pleasant performance in training speed with a good batch size of 16. In Figure 4.28 compares how many times of speed reduced between a normal used GPU and higher spec GPU.

4.15 Compare with existing product

Tesla had deployed a new neural net for Autopilot and it claimed to be massively bigger neural net with impressive new capabilities compare to previous version. The new computer vision had to be significantly updated by such massive network. The network handling all 8 cameras to detect and track vehicles and other objects all around the car. These features add up requires tremendous amount of data to fully utilize all neural network parameters and huge computational processing power. The Tesla CEO Elon Musk claimed that the neural network improvement has nearly 400% increase in useful operation per second due to enabling integrated GPU and further utilize of discrete GPU. Compare with such advance in neural network architecture and GPU processing power, the neural network in this project would be just a learning purpose.

CHAPTER 5

CONCLUSION

The objective of this project is to perform vehicle detection using deep learning. The model used for vehicle detection was originated from You Only Look Once (YOLOv2) architecture which have 23 layers and it is deep enough to recognize multiple classes with good accuracy and fast enough to use in real-time system. It had been trained by own dataset and successfully perform vehicle detection.

In conclusion, the highest mAP obtained were 0.39, 0.33 for IoU 0.3 and 0.5 respectively. The detection accuracy is limited by various factors which has discussed before. Firstly, the greedy NMS algorithm was not smart enough to filter the redundant bounding box from crowded scene causing some overlapped object bounded with only a box instead of multiple discrete boxes. Also, the number of training images used to train the deep model were not enough to improve the model's generalization. Major accuracy improvement in increasing the amount of training dataset and minimum batch size had proven in Section 4.10.

Moreover, the model training loss performance had been tested by larger batch size in Section 4.12. The loss saturated faster with smaller batch size, whereas, larger batch size having a steady converge after saturation compare with smaller batch size. Unfortunately, too low in loss value may interpret the model over-fitting from dataset cause poor detection in new input dataset. Hence, training a model is an state of art which does not have any fix steps to train the model. Instead, training a model require trial and error for tuning the hyper-parameters or adding more layers in order to approach to global minimal where the idea loss with highest testing accuracy.

Finally, all those training and result analysis are unable to perform effectively without a fast GPU. Training deep learning is computationally intensive, hence, acquiring a better GPU will improve in both practical experience and quality result as well.

To help further improve in the model's training performance and vehicle detection accuracy, following recommendation are proposed:

5.1 A diverse large-scale dataset

The deep learning is data centric and requires large data sets that represent all possible driving environments and scenarios in vehicle transport system. Explicitly collecting relevant dataset is expensive and time consuming. An effective way to counter with limited dataset resources is data augmentation. Modifying the existing image to different color space, size, viewpoint, orientation, and transformation would double the number of images with no duplicate data resources. This could helps the model recognize in a variety of conditions, hence, improve in generalization.

5.2 Soft non-max suppression (Soft-NMS)

A post-processing algorithm improve from greedy non-max suppression issues. Greedy NMS easily suppress close-by vehicle's bounding boxes and only retain highest confidence score box causes overlapped vehicle miss detected. Soft-NMS has mitigated this problem by decaying the detection scores of all other vehicles as a continuous function, lower confidence box still be ranked in the list. (Bodla, et al., 2017)

5.3 Deeper and more layer concatenated CNN

The more convolution layer, the more complex structure and feature can be recognized by the model, hence, more complicated works can be resolved or the model can accurately perform the specific task such as classification and localization.

Moreover, concatenate higher layers with lower layers help in detecting smaller object because higher layers usually retain the most information from the data given, hence, some detail information can be further extracted at the lower layers. Many well performing object detection networks such as GoogleNet, Yolov3, Inception Net are built in very deep layers, however, the downside is it requires much more complexity big-scale data to utilize all the layers in the model and the memory size for training the network is increasing dramatically.

5.4 Train in medium batch size

A very large batch size would lead to poor generalization, also, a very small batch size would lead to under-fitting and unable to converge the loss value. Finding a fitted batch size should be in between very large batch size and very small batch size. A suitable one is tightly depending on the amount of dataset with many trial and error evaluation in order to make the model's loss reach or approach to global loss minimal.

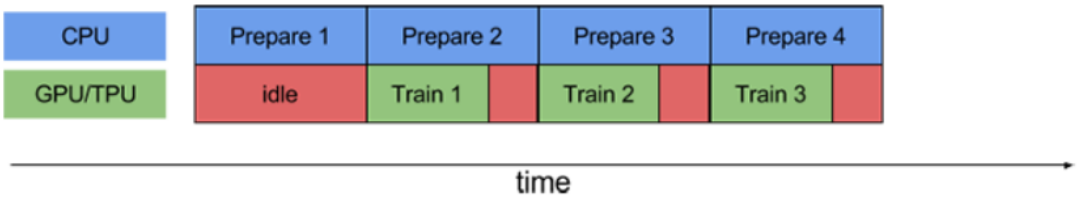
5.5 Fast GPU

Graphic card is designated to handle large-scale mathematical intensive task in parallel especially like deep learning algorithm contain massive of parameters value. A fast GPU only bring benefits in exploring deep learning fields, but the costing would block many learners to afford a decent unit for just further study in this field. Luckily, Google has providing a free online cloud GPU which save them from limited resources for going deeper into deep learning studies, but the cloud GPU have limited time access which would have some minor availability constrain for training a model.

5.6 Data Input Pipeline

To execute a training step, first must pre-processing the data and then input to a model running on a GPU. However, in a naive implementation way would be the CPU is preparing the data and the GPU is sitting idle at the same time. Similarly, while the GPU is training the model, the CPU is sitting idle. Hence, the training step time is sum of both CPU pre-processing time and the GPU training time. To utilize the hardware resources, an efficient data pipe-lining could lessen the resources in idle mood. Tensorflow's data API provides users with building blocks to design input pipelines that effectively. The pipe-lining overlaps the pre-processing and model execution of a training step. Figures show below illustrate the effect of total execution time with and without pipe-lining

(a)



(b)

Figure 6.1: The execution process running (a) without pipe-lining (b)with pipe-lining

REFERENCES

Nguyen, H., Kieu, L., Wen, T. and Cai, C. (2018). Deep learning methods in transportation domain: a review. *IET Intelligent Transport Systems*, 12(9), pp.998-1004.

Yilmaz, A., A., Guzel, M.,S., Askerbeyli, I., Bostanci, E., 2018. *A Vehicle Detection Approach using Deep Learning Methodologies.* [ONLINE] Available at: *https://arxiv.org/ftp/arxiv/papers/1804/1804.00429.pdf* [Accessed 17 February 2019].

Huval, B., Wang, T., Tandon, S., Kiske, J., Song, W., Pazhayampallil, J., Andriluka, M., Rajpurkar, P., Migimatsu, T., Cheng-Yue, R., Mujica, F., Coates, A. and Ng, A. (2019). *An Empirical Evaluation of Deep Learning on Highway Driving*. [online] arXiv.org. Available at: https://arxiv.org/abs/1504.01716 [Accessed 7 Apr. 2019].

Joseph, R., Santosh, D., Ross, G., Ali, F., 2015, *You Only Look Once: Unified, Real-Time Object Detection.* [ONLINE] Available at: https://arxiv.org/pdf/1506.02640.pdf [Accessed 19 February 2019].

Johan, B., Carla, G., Bart, S., Kilian Q.,W., 2018. *Understanding Batch Normalization*. [ONLINE] Available at: https://arxiv.org/pdf/1806.02375.pdf [Accessed 17 February 2019].

Radiuk, P. (2017). *Impact of Training Set Batch Size on the Performance of Convolutional Neural Networks for Diverse Datasets*. *Information Technology and Management Science*, 20(1).

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A. (2014). *Going Deeper with Convolutions*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1409.4842> [Accessed 1 Apr. 2019].

Ioff, S. and Szegedy, C. (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 9, pp.1,4.

Xu, B., Wang, N., Chen, T. and Li, M. (2015). *Empirical Evaluation of Rectified Activations in Convolution Network*, 5, pp.2,3.

He, K., Zhang, X., Ren, S. and Sun, J. (2015). *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, 11, p.8.

Walia, A. (2017). *Activation functions and it's types-Which is better?*. [online] Towards Data Science. Available at: <https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f> [Accessed 1 Apr. 2019].

Redmon, J. and Farhadi, A. (2016). *YOLO9000: Better, Faster, Stronger*, 9, pp.3,4.

Du, J. (2018). Understanding of Object Detection Based on CNN Family and YOLO. *Journal of Physics: Conference Series*, 1004, p.012029.

Menegaz, M. (2018). *Understanding YOLO*. [online] Hacker Noon. Available at: <https://hackernoon.com/understanding-yolo-f5a74bbc7967> [Accessed 28 Mar. 2019].

Redmon, J., Divvala, S., Girshick, R. and Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. 10, p.2.

Yadav, V. (2017). *(Part 1) Generating Anchor boxes for Yolo-like network for vehicle detection using KITTI dataset.*. [online] Medium. Available at: <https://medium.com/@vivek.yadav/part-1-generating-anchor-boxes-for-yolo-like-network-for-vehicle-detection-using-kitti-dataset-b2fe033e5807> [Accessed 29 Mar. 2019].

Hui, J. (2018). *Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3*. [online] Medium. Available at: <https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088> [Accessed 29 Mar. 2019].

Rosebrock, A. (2016). *Intersection over Union (IoU) for object detection - PyImageSearch*. [online] PyImageSearch. Available at: <https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/#> [Accessed 22 Mar. 2019].

Hosang, J., Benenson, R. and Schiele, B. (2017). *Learning non-maximum suppression*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1705.02950> [Accessed 25 Mar. 2019].

Hui, J. (2018). *mAP (mean Average Precision) for Object Detection*. [online] Medium. Available at: https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173 [Accessed 1 Apr. 2019].

Smith, S., Kindermans, P., Ying, C. and Le, Q. (2018). *Don't Decay the Learning Rate, Increase the Batch Size*. [online] arXiv.org. Available at: https://arxiv.org/abs/1711.00489 [Accessed 4 Apr. 2019].

Hoffer, E., Hubara, I. and Soudry, D. (2018). *Train longer, generalize better: closing the generalization gap in large batch training of neural networks*. [online] arXiv.org. Available at: https://arxiv.org/abs/1705.08741 [Accessed 4 Apr. 2019].

Bodla, N., Singh, B., Chellappa, R. and Davis, L. (2017). *Soft-NMS -- Improving Object Detection With One Line of Code*. [online] arXiv.org. Available at: https://arxiv.org/abs/1704.04503 [Accessed 8 Apr. 2019]

Appendix A Dataset Annotation Extractor

```
import sys
import cv2
import os
import pickle
import numpy as np
path = os.getcwd()
sys.path.insert(0, path)
# extract annoatation information and arrange into a list according to the dataset's filename
from preprocessing import parse_annotation
LABELS = ['bicycle','car','motorcycle']
train_annot_folder = os.path.join(path,'output','train','')  # training annotation file
train_image_folder = os.path.join(path,'images','train2017','') # training images file
valid_image_folder = os.path.join(path,'images','val2017','') # testing images file
valid_annot_folder = os.path.join(path,'output','val','') # testing annotation file

# training dataset list and saved into a file
train_imgs, seen_train_labels = parse_annotation(train_annot_folder, train_image_folder, labels=LABELS)
with open(os.path.join(path,'train_imgs'), 'wb') as fp:
    pickle.dump(train_imgs, fp)
# testing dataset list and saved into a file
valid_imgs, seen_valid_labels = parse_annotation(valid_annot_folder, valid_image_folder, labels=LABELS)
with open(os.path.join(path,'valid_imgs'), 'wb') as fp:
    pickle.dump(valid_imgs, fp)
```

Appendix B Vehicle Detection Train Coding

```
import sys
import cv2
import os
import tensorflow as tf
import numpy as np # numpy array
import pickle # list saver and loader
from random import shuffle
from tqdm import tqdm # a progress bar indicator
dirpath = '/content/drive/My Drive/yolo'
sys.path.insert(0, dirpath)
from preprocessing import BatchGenerator # data pre-processing and arrange in batches

LABELS = ['bicycle','car','motorcycle'] # classes going to detect in this project

# hyper-parameters of the model
start_lr = 1e-5  # starting learning rate
hm_epoch = 100   #number of epochs
IMAGE_H, IMAGE_W = 544, 544  # input width and height
GRID_H,  GRID_W  = 17 , 17   # grid cell
BOX              = 5         # number of bounding box
CLASS            = len(LABELS)
CLASS_WEIGHTS    = np.ones(CLASS, dtype='float32')
OBJ_THRESHOLD    = 0.3  # minimum object confidence level
NMS_THRESHOLD    = 0.3  # minimum non-max suppression threshold
NO_OBJECT_SCALE  = 1.0
OBJECT_SCALE     = 5.0
COORD_SCALE      = 1.0
CLASS_SCALE      = 1.0
BATCH_SIZE       = 32
minibatch_size   = 16
TRUE_BOX_BUFFER  = 50
# achor box width and height values
ANCHORS =  [0.57273, 0.677385,
            1.87446, 2.06253,
            3.33843, 5.47434,
            7.88282, 3.52778,
            9.77052, 9.16828]

# training image and testing image folder path,
# weight save folder path, data summaries foler path,
# and weight restore folder path
train_imgs_path = os.path.join(dirpath,'train_imgs')
valid_imgs_path = os.path.join(dirpath,'valid_imgs')
parameter_path_file = os.path.join(dirpath,'data','parameter','tiny')
summary_path = os.path.join(dirpath,'data','summary','trial1','')
weight_reader = os.path.join(dirpath,'data','parameter_1','tiny-15')
# graph and parameters initializer
W_initializer = tf.contrib.layers.xavier_initializer()

# dynamic dataset holder before input to the model
with tf.name_scope('input'):
    x = tf.placeholder('float',[None,IMAGE_H,IMAGE_W,3],name = 'x')
    y = tf.placeholder('float',name = 'y')
    true_boxes = tf.placeholder('float',shape = (None,1, 1, 1, TRUE_BOX_BUFFER , 4),name ='true')
    is_train = tf.placeholder(tf.bool, name="is_train")

# varaibles to hold the output loss every epochs and update to summaries file for storing purpose
with tf.name_scope('performance'):
    tf_total_loss = tf.placeholder(tf.float32,shape=None,name='total_loss')
    tf_loss_summary = tf.summary.scalar('train_loss', tf_total_loss)
tf_update = tf.summary.merge([tf_loss_summary,tf_decay,tf_valid_summary])
```

```
def normalize(image):
    return image / 255.

generator_config = {
    'IMAGE_H'         : IMAGE_H,
    'IMAGE_W'         : IMAGE_W,
    'GRID_H'          : GRID_H,
    'GRID_W'          : GRID_W,
    'BOX'             : BOX,
    'LABELS'          : LABELS,
    'CLASS'           : len(LABELS),
    'ANCHORS'         : ANCHORS,
    'BATCH_SIZE'      : BATCH_SIZE,
    'TRUE_BOX_BUFFER' : 50,
}

with open(train_imgs_path, 'rb') as fp:  # load the saved dataset list into a varaible
    train_imgs = pickle.load(fp)
```

```
def model(x):
    #layer 1
    x = tf.layers.conv2d(x,32,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_1')
    x = tf.nn.leaky_relu(x)
    x = tf.nn.max_pool(x,[1,2,2,1],[1,2,2,1],'SAME')
    #layer 2
    x = tf.layers.conv2d(x,64,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_2')
    x = tf.nn.leaky_relu(x)
    x = tf.nn.max_pool(x,[1,2,2,1],[1,2,2,1],'SAME')
    #-----------------------------------------------------
    #layer 3
    x = tf.layers.conv2d(x,128,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_3')
    x = tf.nn.leaky_relu(x)
    #layer 4
    x = tf.layers.conv2d(x,64,(1,1),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_4')
    x = tf.nn.leaky_relu(x)
    #layer 5
    x = tf.layers.conv2d(x,128,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_5')
    x = tf.nn.leaky_relu(x)
    x = tf.nn.max_pool(x,[1,2,2,1],[1,2,2,1],'SAME')
    #-----------------------------------------------------
    #layer 6
    x = tf.layers.conv2d(x,256,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_6')
    x = tf.nn.leaky_relu(x)
    #layer 7
    x = tf.layers.conv2d(x,128,(1,1),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_7')
    x = tf.nn.leaky_relu(x)
    #layer 8
    x = tf.layers.conv2d(x,256,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_8')
    x = tf.nn.leaky_relu(x)
    x = tf.nn.max_pool(x,[1,2,2,1],[1,2,2,1],'SAME')
    #-----------------------------------------------------
    #layer 9
    x = tf.layers.conv2d(x,512,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_9')
    x = tf.nn.leaky_relu(x)
    #layer 10
    x = tf.layers.conv2d(x,256,(1,1),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_10')
    x = tf.nn.leaky_relu(x)
    #layer 11
    x = tf.layers.conv2d(x,512,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_11')
    x = tf.nn.leaky_relu(x)
    #layer 12
    x = tf.layers.conv2d(x,256,(1,1),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_12')
    x = tf.nn.leaky_relu(x)
    #layer 13
    x = tf.layers.conv2d(x,512,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_13')
    x = tf.nn.leaky_relu(x)
    skip_connection = x
    x = tf.nn.max_pool(x,[1,2,2,1],[1,2,2,1],'SAME')
    #-----------------------------------------------------
    #layer 14
    x = tf.layers.conv2d(x,1024,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_14')
    x = tf.nn.leaky_relu(x)
    #layer 15
    x = tf.layers.conv2d(x,512,(1,1),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_15')
    x = tf.nn.leaky_relu(x)
    #layer 16
    x = tf.layers.conv2d(x,1024,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_16')
    x = tf.nn.leaky_relu(x)
    #layer 17
    x = tf.layers.conv2d(x,512,(1,1),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_17')
    x = tf.nn.leaky_relu(x)
    #layer 18
    x = tf.layers.conv2d(x,1024,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_18')
    x = tf.nn.leaky_relu(x)
    #layer 19
    x = tf.layers.conv2d(x,1024,(3,3),strides=(1,1),padding='same',use_bias=False,kernel_initializer=W_initializer)
    x = tf.layers.batch_normalization(x, training=is_train,name = 'norm_19')
    x = tf.nn.leaky_relu(x)
    #layer 21
    skip_connection = tf.layers.conv2d(skip_connection,64,(1,1),strides=(1,1),padding='same',use_bias=False
                                       ,kernel_initializer=W_initializer)
    skip_connection = tf.layers.batch_normalization(skip_connection, training=is_train,name = 'norm_21')
    skip_connection = tf.nn.leaky_relu(skip_connection)
    skip_connection = tf.space_to_depth(skip_connection,block_size=2)
    x = tf.concat([skip_connection,x],axis=-1)
    #layer 22
```

```
def custom_loss(y_true, y_pred):
    mask_shape = tf.shape(y_true)[:4]

    cell_x = tf.to_float(tf.reshape(tf.tile(tf.range(GRID_W), [GRID_H]), (1, GRID_H, GRID_W, 1, 1)))
    cell_y = tf.transpose(cell_x, (0,2,1,3,4))

    cell_grid = tf.tile(tf.concat([cell_x,cell_y], -1), [minibatch_size, 1, 1, 5, 1])

    coord_mask = tf.zeros(mask_shape)
    conf_mask  = tf.zeros(mask_shape)
    class_mask = tf.zeros(mask_shape)

    seen = tf.Variable(0.,trainable = False)
    total_recall = tf.Variable(0.,trainable = False)

    """
    Adjust prediction
    """
    ### adjust x and y
    pred_box_xy = tf.sigmoid(y_pred[..., :2]) + cell_grid

    ### adjust w and h
    pred_box_wh = tf.exp(y_pred[..., 2:4]) * np.reshape(ANCHORS, [1,1,1,BOX,2])

    ### adjust confidence
    pred_box_conf = tf.sigmoid(y_pred[..., 4])

    ### adjust class probabilities
    pred_box_class = y_pred[..., 5:]

    """
    Adjust ground truth
    """
    ### adjust x and y
    true_box_xy = y_true[..., 0:2] # relative position to the containing cell

    ### adjust w and h
    true_box_wh = y_true[..., 2:4] # number of cells accross, horizontally and vertically

    ### adjust confidence
    true_wh_half = true_box_wh / 2.
    true_mins    = true_box_xy - true_wh_half
    true_maxes   = true_box_xy + true_wh_half

    pred_wh_half = pred_box_wh / 2.
    pred_mins    = pred_box_xy - pred_wh_half
    pred_maxes   = pred_box_xy + pred_wh_half

    intersect_mins  = tf.maximum(pred_mins,  true_mins)
    intersect_maxes = tf.minimum(pred_maxes, true_maxes)
    intersect_wh    = tf.maximum(intersect_maxes - intersect_mins, 0.)
    intersect_areas = intersect_wh[..., 0] * intersect_wh[..., 1]

    true_areas = true_box_wh[..., 0] * true_box_wh[..., 1]
    pred_areas = pred_box_wh[..., 0] * pred_box_wh[..., 1]

    union_areas = pred_areas + true_areas - intersect_areas
    iou_scores  = tf.truediv(intersect_areas, union_areas)

    true_box_conf = iou_scores * y_true[..., 4]

    ### adjust class probabilities
    true_box_class = tf.argmax(y_true[..., 5:], -1)

    """
    Determine the masks
    """
    ### coordinate mask: simply the position of the ground truth boxes (the predictors)
    coord_mask = tf.expand_dims(y_true[..., 4], axis=-1) * COORD_SCALE

    ### confidence mask: penelize predictors + penalize boxes with low IOU
    # penalize the confidence of the boxes, which have IOU with some ground truth box < 0.6
    true_xy = true_boxes[..., 0:2]
    true_wh = true_boxes[..., 2:4]

    true_wh_half = true_wh / 2.
    true_mins    = true_xy - true_wh_half
    true_maxes   = true_xy + true_wh_half

    pred_xy = tf.expand_dims(pred_box_xy, 4)
    pred_wh = tf.expand_dims(pred_box_wh, 4)

    pred_wh_half = pred_wh / 2.
    pred_mins    = pred_xy - pred_wh_half
    pred_maxes   = pred_xy + pred_wh_half
    intersect_mins  = tf.maximum(pred_mins,  true_mins)
    intersect_maxes = tf.minimum(pred_maxes, true_maxes)
    intersect_wh    = tf.maximum(intersect_maxes - intersect_mins, 0.)
    intersect_areas = intersect_wh[..., 0] * intersect_wh[..., 1]

    true_areas = true_wh[..., 0] * true_wh[..., 1]
    pred_areas = pred_wh[..., 0] * pred_wh[..., 1]

    union_areas = pred_areas + true_areas - intersect_areas
    iou_scores  = tf.truediv(intersect_areas, union_areas)

    best_ious = tf.reduce_max(iou_scores, axis=4)
    conf_mask = conf_mask + tf.to_float(best_ious < 0.6) * (1 - y_true[..., 4]) * NO_OBJECT_SCALE

    # penalize the confidence of the boxes, which are reponsible for corresponding ground truth box
    conf_mask = conf_mask + y_true[..., 4] * OBJECT_SCALE

    ### class mask: simply the position of the ground truth boxes (the predictors)
    class_mask = y_true[..., 4] * tf.gather(CLASS_WEIGHTS, true_box_class) * CLASS_SCALE

    #Loss function
    nb_coord_box = tf.reduce_sum(tf.to_float(coord_mask > 0.0))
    nb_conf_box  = tf.reduce_sum(tf.to_float(conf_mask  > 0.0))
    nb_class_box = tf.reduce_sum(tf.to_float(class_mask > 0.0))
```

```
    loss_xy    = tf.reduce_sum(tf.square(true_box_xy-pred_box_xy)     * coord_mask) / (nb_coord_box + 1e-6) / 2.
    loss_wh    = tf.reduce_sum(tf.square(true_box_wh-pred_box_wh)     * coord_mask) / (nb_coord_box + 1e-6) / 2.
    loss_conf  = tf.reduce_sum(tf.square(true_box_conf-pred_box_conf) * conf_mask)  / (nb_conf_box  + 1e-6) / 2.
    loss_class = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=true_box_class, logits=pred_box_class)
    loss_class = tf.reduce_sum(loss_class * class_mask) / (nb_class_box + 1e-6)

    loss = loss_xy + loss_wh + loss_conf + loss_class
    return loss
```

```
prediction = model(x) # convolution neural network model
loss = custom_loss(y,prediction) # loss function
opt = tf.train.AdamOptimizer(learning_rate=start_lr)  # Adam optimizer
op_update = tf.get_collection(tf.GraphKeys.UPDATE_OPS) # collect all the trainable varaibles for updating purpose
tvs = tf.trainable_variables()
# 1) Create placeholders for the accumulating gradients we'll be storing
accum_vars = [tf.Variable(tv.initialized_value(),trainable=False) for tv in tvs]
# 2) Operation to initialize accum_vars to zero
zero_ops  = [tv.assign(tf.zeros_like(tv)) for tv in accum_vars]
# 3) Operation to compute the gradients for one minibatch
gvs = opt.compute_gradients(loss)
# 4) Operation to accumulate the gradients in accum_vars
accum_ops = [accum_vars[i].assign_add(gv[0]) for i, gv in enumerate(gvs)]
# 5) Operation to perform the update (apply gradients)
saver = tf.train.Saver()
with tf.Session() as sess:
    train_op = opt.apply_gradients([(accum_vars[i], tv) for i, tv in enumerate(tvs)])
    sess.run(tf.global_variables_initializer())
    saver.restore(sess,weight_reader)
    #train_writer = tf.summary.FileWriter(summary_path,sess.graph)
    for epoch in range(hm_epoch):
        shuffle(train_imgs) # shuffle the dataset
        train_batch = []
        train_batch = BatchGenerator(train_imgs, generator_config, norm=normalize) # pre-processing the dataset and arrange into batches
        epoch_loss,valid_loss, mini_loss = 0, 0, 0
        #train_writer.add_summary(sess.run(tf_decay),epoch)
        for batch in tqdm(train_batch,desc = 'Iteration: ',leave =False):
            sess.run(zero_ops)
            mini_loss = 0
            for j in range(int(len(batch[1])/minibatch_size)):
                _,c,k = sess.run([accum_ops,loss,op_update],feed_dict={x: batch[0][0][j*minibatch_size:(j+1)*minibatch_size],
                                                       true_boxes:batch[0][1][j*minibatch_size:(j+1)*minibatch_size],
                                                       y: batch[1][j*minibatch_size:(j+1)*minibatch_size],
                                                       is_train:True})
                mini_loss += c
            mini_loss = mini_loss / (len(batch[1])/minibatch_size)
            sess.run(train_op)
            epoch_loss += mini_loss
        epoch_loss = epoch_loss / len(train_batch)

        print('Epoch', epoch,'--->','loss:',epoch_loss,'current_lr:', start_lr)

        if ((epoch+1)%6== 0): # learning rate decay 0.97 every 6 epoch
            start_lr = start_lr*0.97
            saver.save(sess,parameter_path_file,global_step= epoch+1,write_meta_graph=False)
        if ((epoch+1)%10 == 0): # parameters saved every 10 epoch
            saver.save(sess,parameter_path_file,global_step= epoch+1,write_meta_graph=False)
            train_writer.flush() # flush all the pending event file into summaries file
        # update the event file
        summ_overall = sess.run(tf_update,{tf_total_loss:epoch_loss,tf_valid_loss:valid_loss})
        train_writer.add_summary(summ_overall,epoch)

    print('Training is completed!!!')
    sess.close()
```

Appendix C Vehicle Detection Test Coding

```
import cv2
import sys
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from tqdm import tqdm_notebook
sys.path.insert(0, r'C:\Users\teoh\Desktop\hi')
from utils import decode_netout, compute_overlap, draw_boxes,BoundBox

Image_h ,Image_w = 544 ,544
anchors =  [0.57273, 0.677385, 1.87446, 2.06253, 3.33843, 5.47434, 7.88282, 3.52778, 9.77052, 9.16828]
obj_threshold = 0.3
nms_threshold = 0.3
nb_class = 3
LABELS = ['bicycle', 'car', 'motorcycle']
video_inp = r'C:\Users\teoh\Desktop\New folder (2)\video\VID20190315112920.mp4' # video source folder path
video_out = r'C:\Users\teoh\Desktop\New folder (2)\test3_544.mp4'               # video saved folder path

# Arrange the video into sequence frame and resize each frame before feed to model
video_reader = cv2.VideoCapture(video_inp)
nb_frames = int(video_reader.get(cv2.CAP_PROP_FRAME_COUNT)-10)
frame_h = int(video_reader.get(cv2.CAP_PROP_FRAME_HEIGHT))
frame_w = int(video_reader.get(cv2.CAP_PROP_FRAME_WIDTH))

video_writer = cv2.VideoWriter(video_out,
                               cv2.VideoWriter_fourcc(*'XVID'),
                               20.0,
                               (frame_w, frame_h))
# rotate the frame because some frame may rotated
def rotateImage(image, angle):
    image_center = tuple(np.array(image.shape[1::-1]) / 2)
    rot_mat = cv2.getRotationMatrix2D(image_center, angle, 1.0)
    result = cv2.warpAffine(image, rot_mat, image.shape[1::-1], flags=cv2.INTER_LINEAR)
    return result

tf.reset_default_graph()

with tf.Session() as sess:
    # import the convolution neral network graph
    saver = tf.train.import_meta_graph(r'C:\Users\teoh\Desktop\New folder (2)\tiny_yolo_544.meta')
    # initialziet the graph
    sess.run(tf.global_variables_initializer())
    # load the trained weight file to the graph
    saver.restore(sess,r'C:\Users\teoh\Desktop\New folder (2)\trail21\tiny-15')
    # dynamic dataset holder
    x = tf.get_default_graph().get_tensor_by_name("input/x:0")
    train = tf.get_default_graph().get_tensor_by_name("input/is_train:0")
    output = tf.get_default_graph().get_tensor_by_name("output:0")
    # repeat until the last frame of video
    for i in tqdm_notebook(range(nb_frames)):
        ret, image = video_reader.read()
        image= rotateImage(image, 0) # rotate the video frame
        input_image = cv2.resize(image, (544, 544))
        input_image = input_image / 255. # normalize the image
        input_image = input_image[:,:,::-1]
        input_image = np.expand_dims(input_image, 0)
        feed_dict = {x:input_image,train:False}
        prediction = output.eval(feed_dict) # start the predicting
        prediction = np.squeeze(prediction, axis=0)
        out = decode_netout(prediction,anchors,nb_class,obj_threshold,nms_threshold) # post-processing
        image = draw_boxes(image,out,labels=LABELS) # drawing the bounding box on each frame
        video_writer.write(np.uint8(image))

video_reader.release()
video_writer.release()
```