Honours Project - MHW225671

# Final Report

2020-2021

Department of Computing

Submitted for the Degree of BSc Computing

**Project Title: Object Detection for Content Moderation**

**Name: Daniel Russell**

**Programme: BSc Computing**

**Matriculation Number: S1707149**

**Project Supervisor: Richard Holden**

**Second Marker: Robert Law**

**Word Count: 9407**
(*excluding contents pages, figures, tables, references and Appendices*)

**"Except where explicitly stated, all work in this report, including the appendices, is my own original work and has not been submitted elsewhere in fulfilment of the requirement of this or any other award"**

**Signed by Student: Daniel Russell Date: 24/01/2021**

# Contents

# Abstract

With the ever-increasing demand and userbase on online social media, online forums, online trading and content creation sites the need for big and small companies to automate the process of content moderation greatly increase as the workload becomes too great for just human moderators to handle. This project asks the question "which the most efficient object detecting model that can be used for the use case of content moderation?".

In order to answer this research question, a literature and technology review was conducted to research a possible dataset for use in training alongside research conducted on a selection of object detecting neural networks which were Faster R-CNN, EfficientDet and RetinaNet. Following this, a develop and test approach was adopted following the waterfall methodology to create a notebook application written in Python and executed on Google Collab to train, test and evaluate the three object detecting architectures using TensorFlow to determine which one of the three is the most efficient.

After encountering severe issues with Google Collab during testing and training, all test results and data gathered from the execution section was evaluated to conclude that the Faster R-CNN object detecting architecture was the most efficient out of the three chosen for this project by overall average precision, average recall and number of successful detections.

# 1.0 Introduction

This section of the report contains the introductory section outlining the subject matter of content moderation and object detection before delving deeper into the problem to be discussed and solved including an in-depth discussion on the approach considered to solve said problem to answer a defined research question.

This section will discuss content moderation and why it has become a necessary step in maintaining intellectual property rights, security and ensuring user satisfaction across any type of online application before further expanding on the topic of machine learning and deep learning regarding computer vision and object detecting architectures.

By the end of this section there will be a final discussion about the research question that this project is based on followed by the project aims and objectives considered as the approach to answer said research question before ending with a final overview of the remaining content of this report.

## *1.1 Project Background*

To compare the start of the digital age after the 90's era of early social media, content creation sites and online marketplaces there was far less users and online activity back then than compared to this present day. As different types of hardware become more affordable, more devices adapted for cross-platform interaction, more services and apps accepting increasing amounts of data from their users into company servers the need for big and small companies to police their platform greatly increases. With such an immense workload that exponentially increases for human moderators the only logical solution to this problem is to automate the process of content moderation to block and remove illegal or malicious activity.

### YouTube Content ID Copyright Detection Example

One such example of a company that uses an automated content moderation system is YouTube and their Content ID system developed by Google. As best described by Google's YouTube help centre "Content ID is YouTube's automated, scalable system that enables copyright owners to identify YouTube videos that include content that they own." (Youtube, 2020). In other words, it is an automated content moderation system that allows a copyright holder to sign-up and create a digital fingerprint for their copyrighted material that is then routinely used to compare with the millions of newly uploaded content to detect theft of intellectual property that would otherwise be impossible to moderate with humans alone.

There are many other ways to automate the process of content moderation like Youtube's Content ID system which uses digital fingerprinting but another way that content moderation can be automated is via the use of machine learning and AI.

### Machine Learning

Machine learning is an increasingly complex field in artificial intelligence and data science, as best summarised by IBM's definition of machine learning on their Cloud Learn Hub, "Machine learning is a branch of artificial intelligence (AI) focused on building applications that learn from data and improve their accuracy over time without being programmed to do so.” (IBM Cloud Education, 2020).

## Deep Learning

Deep learning is a subset of machine learning which makes full use of neural networks which are algorithms designed to behave like an artificial brain to allow a program to train and learn in a supervised (With labels), unsupervised (Without labels) or reinforced (via a reward-based system) manner to achieve an end goal or make an estimated prediction when making an inference.

One such an example of a neural network would be a CNN or Convolutional Network which is a neural network designed to work with image data that is commonly used in image classification and computer vision. A CNN is usually characterised by convolutional layers used to filter the data of an inserted image to detect features by generating a feature map before outputting the resulting feature map to a pooling layer to reduce the overall size of the resulting data (Stewart, 2019).

## Facebook AI Driven Hate Speech and Misinformation Detection Example

Facebook for several years has been using various types of and versions of machine learning technologies to monitor their platform to hide sensitive media, delete and block hate speech and policy violations as well as handling the responses to users who post such content to the public. In reference to a blog post made by the chief technology officer at Facebook. Facebook’s detection and decision making on hate speech went from 81% in 2019 to 95% in 2020 making use of a custom made cross-lingual library named XLM to train language models across multiple languages to detect hate speech alongside various other systems such as SimSearchNet which is a convolutional neural network image matching tool used to detect misinformation from similar looking media and Reinforced Integrity Optimizer(RIO) a reinforcement learning framework used optimize speech classification to detect hateful speech (Schroepfer, 2020).

## Computer Vision

Computer vision is also a growing field that has taken full advantage of deep learning to produce object detecting methodologies with an increasing degree of accuracy and efficiency, computer vision is best described in a blog post on Adobe’s XD Ideas, “Computer vision is the field of computer science that focuses on creating digital systems that can process, analyse, and make sense of visual data (images or videos) in the same way that humans do.” (Babich, 2020)

## Types of Computer Vision

When using a combination of computer vision and a deep learning algorithm it usually falls under one of several types of categories such as image classification which is when a whole Image is placed into a known category(label) based on the detection of one or multiple objects or a pattern in the said image.

Image segmentation on the other hand is different to image classification in that instead of categorising the whole image the neural network instead seeks to detect one or multiple objects or a pattern in an image to outline or mask the desired target from the image.

Like image segmentation however object detection is when a whole image is analysed by a neural network to detect one or multiple objects or a pattern to predict their location in the image and categorise each detected object usually in the form of a box border or key points rendered onto the image.

### Airbnb Object Detection and Image Classification Example

An example of object detection in use would be Airbnb's experiment in using machine learning to classify images into room categories to improve the experience of their customers. The team of data scientists at Airbnb used TensorFlow to compile and train a selection of models such as a retrained and a modified version of ResNet50 for image classification with some success before using Faster R-CNN to experiment with object detection of furniture to identify room types with both experiments using TensorFlow's Object Detection API. (Yao, 2018).

### TensorFlow

Explaining in short for the introduction to be expanded upon further in the literature review. The TensorFlow Framework is a free and open-sourced machine learning framework developed for C++ and Python by Google intended to be used to make the process of implementing, training and running inference of neural networks easier.

Moving on from the Airbnb example stated above, this project will also make use of TensorFlow and its associated object detection API to implement and train a set of objects detecting networks such as Faster-RCNN, EfficientDet and Retinanet.

### Faster-RCNN

Faster-RCNN is a popular object detecting architecture proposed in 2015 by Ross Girshick, Shaoqing Ren, Kaiming He and Jian Sun that makes use of three separate neural networks to detect objects. The first is a convolutional layer that extracts features from an image before passing the data through a detection proposal layer ending with a final layer to handle box predictions and final adjustments (KHAZRI, 2019).

### EfficientDet

EfficentDet is a relatively new object detecting architecture proposed in 2020 by Mingxing Tan, Ruoming Pang and Quoc V. Le at Google designed to be efficient and scalable via the use of a new bi-directional feature network and scaling rules. The EfficientDet architecture utilizes three layers to extract multiple levels of features from an image before feeding the data through a prediction and box layer (Yu & Tan, 2020).

### RetinaNet

RetinaNet is another popular object detecting architecture that is a one-stage detection model proposed in 2017 by Tsung-Yi Lin, Priya Goyal, Ross Girshick and Kaiming He at

Facebook AI research to create an accurate one stage detector. The RetinaNet architecture makes use of four layers total first of which are two layers to extract features from an image before passing it through two additional layers for predictions and box adjustments (ArcGis Developers, n.d.).

## *1.2 Project Methodology and Objectives*

### 1.2.1 Project Outline

This project aims to find an efficient object detecting neural architecture by first conducting research into a set of object detecting models such as EfficientDet, Faster-RCNN and RetinaNet to consider their structure and performance before moving to primary research to create a custom dataset of blacklisted objects to use before implementing and training the above stated object detecting models using TensorFlow and TensorFlow's provided object detection API to detect the said blacklisted objects from the dataset to gauge which model is the most efficient to use under the use case of content moderation.

All primary and secondary research conducted in this project contributes to the research question by researching and comparing a selection of object detecting architectures that can be implemented via the use of TensorFlow and the object detection API to implement, train and test the object detecting architectures considered for this project to find the most efficient model structurally in theory and in practical implementation for the purposes of object detection for content moderation.

### 1.2.3 Project Objectives

#### Project Objectives

Overall, the objectives of this capstone project can be listed as follows in this sub-section.

***Secondary Objectives***

***Investigate development technologies that can be used for machine learning.***

Preform research into technologies that be use alongside the Python programming language that could aid in the creation of a suited development environment for the implementation, training and testing of object detecting neural networks.

***Source potential datasets that could be used for primary research.***

Preform a search for any potential datasets that contain illicit content such as weaponry, hateful imagery or violence that could be used for primary research. If no suitable dataset is found, then a new dataset should be created as part of this project's primary objectives.

***Investigate object detecting neural networks.***

Preform research into the performance and structures of potential object detecting neural networks that should be considered for implementation in this project such as EfficientDet, Faster-RCNN and RetinaNet to compare during primary research.

***Primary Objectives***

***Collate a suitable dataset of images and process with labels.***
In order to begin the practical implementation of the project a significantly sized dataset of images must be sourced or created with properly labelled classes for use in the training and or testing of the chosen object detecting architectures EfficientDet, RetinaNet and Faster-RCNN. The dataset should be compatible with TensorFlow by being in a TFRecord format.

***Develop a test plan.***
With a compete dataset that has been partitioned into training, validation during training and testing post-mortem. A test plan should be written containing the number of objects in the image, expected positive/negative detection along with an actual detection field for each model implemented for later comparison alongside analytics gathered from training and testing.

***Acquire detection model weights.***
Acquire the pre-trained detection model weights of EfficientDet, Faster-RCNN and RetinaNet via TensorFlow's "Model Zoo" that is connected to the object detection API to process into a new set of objects detecting models trained from scratch using the collated dataset.

***Train models using TensorFlow.***
All setup now complete, the models should now be ready to be trained using TensorFlow and the scripts provided via TensorFlow's object detection API to systematically train each model using the same batch-size, number of classes, and number of steps before training ends. Data at this stage such as learning rate, elapsed time and loss will be recorded for future reference when making final comparison.

***Test trained models***
Upon all models completing training each model will be systematically made to run inference of each image present in the test dataset to gauge the number of true and false detections, confidence and elapsed times for final comparison.

***Evaluate results to reach conclusion.***
With all models trained and tested, all data collated from both training and testing such as elapsed times, true/false detections, number of detections, confidence, loss and learning rate from each object detecting model will be compared to conclude the most efficient model for object detection in this project.

### 1.2.4 Research Question

"What is the most efficient object detecting architecture that can be used to moderate content while requiring minimal training data and maintaining a relatively high degree of success?"

## *1.3 Outline of Contents*

The remaining chapters of this report will cover all secondary research conducted in the form of a literature and technology review, in which the literature and technology review shall cover technologies relevant to the project such as Google Colab, RoboFlow and TensorFlow before discussing a pistol dataset sourced from RoboFlow. Furthermore, the literature and technology review will expand upon the chosen object detection models such as Faster R-CNN, EfficientDet and RetinaNet in greater detail before making an initial assessment on which model is the most efficient in theory.

Next would be the execution section of the report which will cover all the primary research conducted on Google Colab implementing the chosen object detection models via TensorFlow's object detection API and running tests to produce a final set of statistics and test results for evaluation.

By the evaluation section a final evaluation of the results gathered from the detection models will be made alongside a comparison to any results gathered from the literature and technology review to estimate the possibility of error during implementation or testing. At the end of this report in the conclusion section, a conclusion will be reached to which model has performed the most efficiently to answer the above defined research question.

# 2.0 Literature and Technology Review

The literature review involves conducting research into technologies, papers, journals and articles that can demonstrate relevant concepts and ideas from similar projects on the subject matter of machine learning and object detection required to complete the project.

This literature review will build upon the abridged descriptions previously left off in the project introduction of the TensorFlow Framework and the three object-detecting neural networks considered for use in this project such as Faster-RCNN, EfficientDet and RetinaNet in greater depth and detail.

## *2.1 Investigate development technologies that can be used for machine learning.*

The purpose of this sub-section of the literature review is to conduct research into technologies that can be used alongside the Python programming language for the implementation, training and testing of object detecting neural networks such as the TensorFlow Framework and its associated object detection API.

### 2.1.1 RoboFlow

RoboFlow is a free to use start-up cloud service designed to make dataset management for computer vision easier by providing all the necessary tools to collate, bounding box label, split, pre-process and publish a created dataset with ease. RoboFlow provides a sizeable public library of models and published datasets specialised for computer vision that can be forked and exported or programmatically downloaded in several formats including JSON COCO, XML Pascal or a TensorFlow TFRecord (Bhattacharyya, 2020).

### 2.1.2 Google Colab

Google Collaboratory or Google Colab is a free to use hosted notebook service developed by Google that can be accessed via a browser to host and share notebooks that allow for the writing and execution of Python code alongside various other commands in the form of magic tags that can allow the notebook to execute command line commands, bash, extensions, etc. The machine resources available from Google Colab however such as memory, CPU, GPU and TPU are not guaranteed as they tend to fluctuate with use across many users and each session created is limited by time constraints to prevent unnecessary extended use of server resources. (Google, n.d.)

### 2.1.3 TensorFlow

Continuing from the introduction TensorFlow is a free to use and open-source machine learning framework that offers an impressive array of APIs for use in machine learning such as the previously mentioned object detection API while also having deep learning libraries such as Kera and Touch integrated into the framework itself. TensorFlow is also quick to compile alongside built-in support for multi-dimensional arrays such as Tensors and support for several CPUs, GPUs and TPU devices. (Messenger, 2018)

### 2.1.4 Object Detection API

The object detection API provided by TensorFlow provides compatibility to the two versions of TensorFlow (TF1 and TF2) alongside additional built-in scripts for a quick setup of the TensorFlow environment and scripts for commencing the training of models, debugging tools as well as additional scripts for detection box rendering, model evaluation and exporting (Rathod & Huang, 2020).

Another advantage to using the object detection API is its “Model Zoo” which is a library of pre-trained object detecting models trained on several public datasets alongside pre-made configuration files for each model to allow users to train from scratch using the pre-defined architecture of each configuration (Sethi, 2020).

## *2.2 Sourced Pistol Dataset from RoboFlow*

In the search for a dataset that would be suitable for this project, a pistol dataset was found published on RoboFlow by a user sourcing the public pistol dataset from the University of Granada (RoboFlow, 2020). The Dataset in question on RoboFlow is in the public domain and is titled “Pistols Dataset” containing 1 annotation class “Pistol” alongside a total of 2986 images with 3448 labels (RoboFlow, 2020).

The RoboFlow pistol dataset references a research group from the University of Granada discussing a selection of studies and public datasets around the topic of weapons detection for security and video surveillance. The datasets of this research group have been made available as open datasets available at DaSCI as known as The Audalusian Research Institute in Data Science and Computational Intelligence (Triguero, et al., n.d.).

From the DaSCI source it provides a description about the handgun detection dataset stating the dataset contains 3000 images of handguns from various angles and backgrounds including from images of video surveillance (DaSCI, 2020). It is also stated at the end of the dataset description that this dataset has been designed from a related journal using deep learning to create a handgun detection alarm able to detect handguns from a video feed. (DaSCI, 2020)

### 2.2.1 Associated Handgun Alarm Detector Journal

The Journal article referenced above from DaSCI is from a journal named Neurocomputing which in volume 275, on pages 66-72 is the referenced journal article “automatic handgun detection alarm in videos using deep learning” (DaSCI, 2020).

An overview of details from this journal article shows this handgun detection project made use of a combination of neural networks such as Faster-RCNN and another architecture named VGG16 which were evaluated on by using both the sliding window and region proposal detection approaches (Olmos, et al., 2017). Faster-RCNN showed the most promising results as it provided no false positive detection, 100% recall and shown it could detect a handgun and trigger an alarm in a time interval smaller than 0.2s in 27 out of 30 scenes total due to being more efficient when accessing data with an 85.21% precision (Olmos, et al., 2017).

The handgun detection project also went through many iterations of the same dataset to test the training of the two classification models Faster-RCNN and VGG16 effectively (Olmos, et al., 2017). The training started with 9100 images in one dataset which only contained 3990 images of pistols before being later narrowed down in their fifth database of 3000 high quality images in total of only images of pistols with 2 label classes to gauge the classification performance of each model. (Olmos, et al., 2017)

## *2.3 Investigate object detecting neural networks.*

Resuming from the introduction a furthermore in-depth explanation and examination of the three object-detecting architectures Faster-RCNN, EfficientDet and RetinaNet will be provided alongside a final in literature review comparison and initial hypothesis deciding which model in theory should be the most efficient in the use case of content moderation.

### 2.3.1 Performance Metrics

Before delving into object detecting architectures a few key measurement metrics commonly used to demonstrate a model's performance must be defined.

Intersection over Union (IoU) is a measurement commonly used in model evaluation to gauge the similarity of the generated bounding boxes compared to the labelled bounding boxes in the test dataset ranging between the values 0 and 1 (Aidouni, 2019).

Recall refers to the sensitivity of an object detecting model and the probability of the object detection model having a correct/false detection (Aidouni, 2019).

Precision refers to the probability of a generated bounding box from running inference matching the same coordinates of the bounding boxes defined in the dataset, usually ranging between the values of 0 and 1 (Aidouni, 2019).

Average Recall (AR) summarises the distribution of recall values across IoU thresholds (Aidouni, 2019).

Average Precision (AP) is a performance measurement used to summarize the precision-recall curve in a single numerical value by averaging precision across every recall value on the curve (Aidouni, 2019).

Mean Average Precision (mAP or interchangeably written as AP), the main evaluation metric used to gauge a detection model's overall accuracy by first calculating the AP for each class before averaging every AP over each class used by the detector  (Aidouni, 2019).

## 2.3.2 Region Proposal Network (RPN)

Before touching on Faster-RCNN a quick rundown of what a Region Proposal Network is should be given seeing as the Faster-RCNN architecture makes use of this network for region proposals (Ananth, 2019).

The Region Proposal Network (RPN) is a small convolution neural network with two output layers for box coefficients and classification scoring as laid out in *Image 1* (Ananth, 2019).

The RPN is designed to make region proposals by being trained to generate every possible bounding box possible from a feature map which is generated by the RPN's backbone (MADALI, 2020). Upon making bounding box predictions each bounding box is then given an IoU (Intersection over union) value from the classifier to indicate the ground truth or the possibility of that bound box containing an object before making final calculations of regression coefficients to adjust the coordinates of each bounding box. (MADALI, 2020)

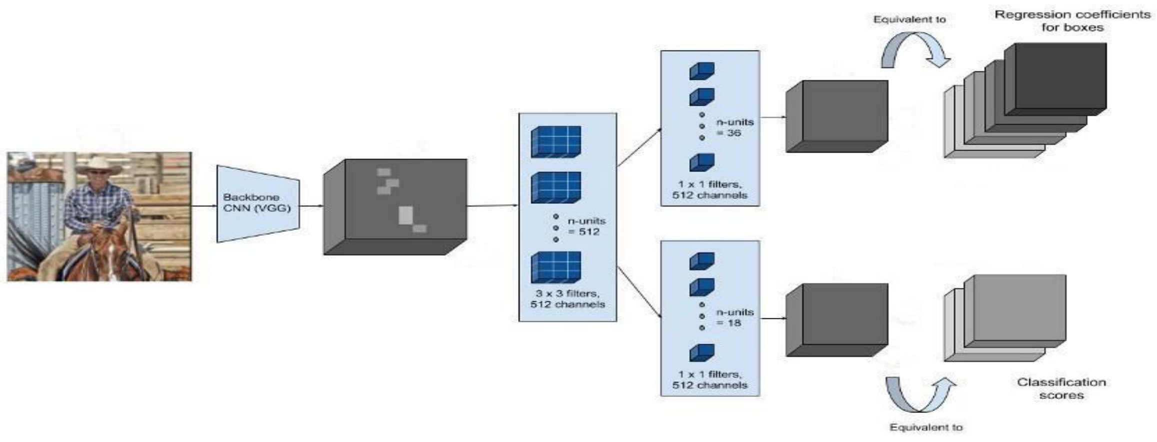

Backbone CNN (VGG)
3 x 3 filters, 512 channels
n-units = 512
n-units = 36
1 x 1 filters, 512 channels
n-units = 18
1 x 1 filters, 512 channels
Equivalent to
Regression coefficients for boxes
Classification scores
Equivalent to

***Image 1* – Region Proposal Network sourced from (Ananth, 2019)**

## 2.3.3 Faster RCNN

Continuing from when Faster-RCNN was first introduced in the introduction, Faster-RCNN is a popular object detecting architecture proposed in 2015 by Ross Girshick, Shaoqing Ren, Kaiming He and Jian Sun (Ren, et al., 2016).

Faster R-CNN is one of the latest additions to the R-CNN style of networks which are a collection of object detection algorithms characterised by being computationally efficient and quick at running inference during testing. The R-CNN style of networks usually make use of a region proposal algorithm such as a selective search algorithm but in the case of Faster R-CNN it instead makes use of the previously mentioned RPN for region proposals which greatly decreases proposal times from 2 seconds to 10 milliseconds (Ananth, 2019).

According to the paper that proposes the Faster R-CNN network, the paper describes the Faster R-CNN network as a network composed of two components. First being the newly added RPN network to make region proposals that is then fed into the modified Fast R-CNN detector to tell the detector were to look for possible objects (Ren, et al., 2016) which is best exemplified by *Image 2* (Ananth, 2019).

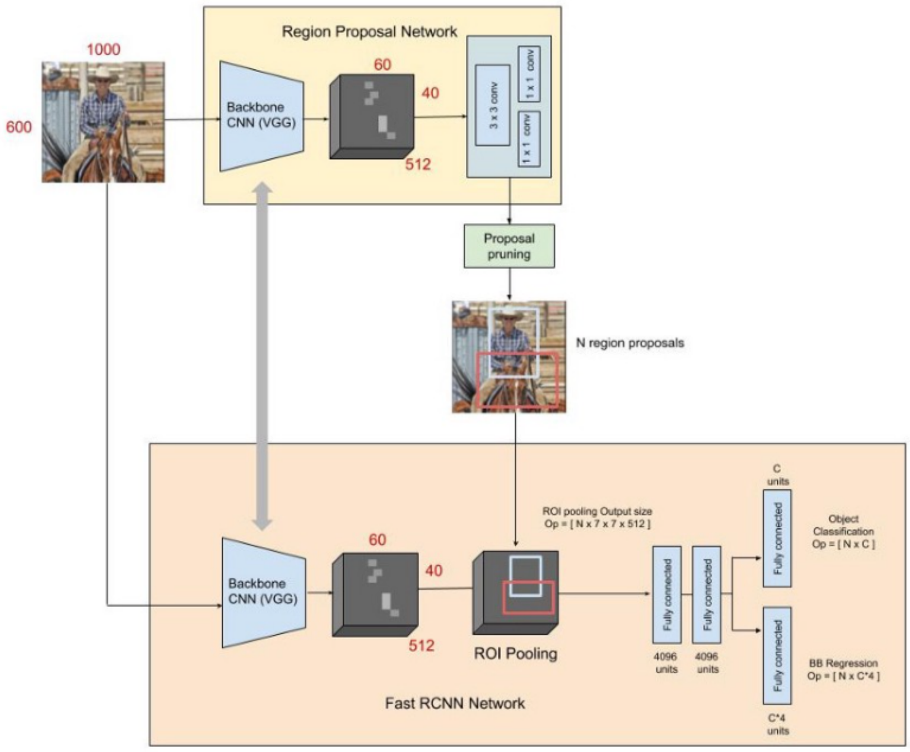
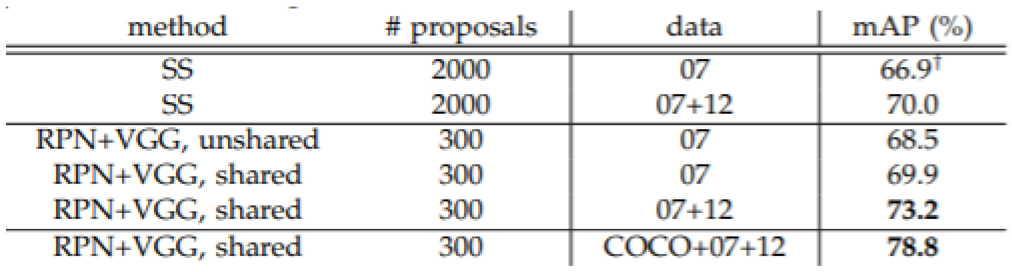


**Image 2 – Faster R-CNN Diagram sourced from (Ananth, 2019)**

Interestingly from testing shown from the paper proposing the Faster R-CNN detection network, the Faster R-CNN network was shown using the new combination of an RPN and VGG backbone for the region proposal algorithm allowing for features to be shared and unshared between the Fast R-CNN network (Ren, et al., 2016) The first performed unshared RPN and VGG perform similarly to selective search with an average precision of 68.5% but greatly improved in accuracy when sharing between the region proposal network was enabled as visible in *Table 1* (Ren, et al., 2016)**.**



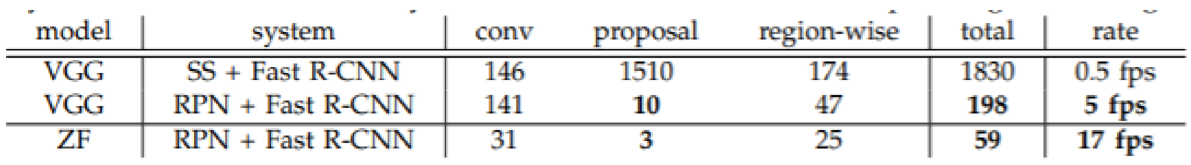
| method | # proposals | data | mAP (%) |
|---|---|---|---|
| SS | 2000 | 07 | 66.9† |
| SS | 2000 | 07+12 | 70.0 |
| RPN+VGG, unshared | 300 | 07 | 68.5 |
| RPN+VGG, shared | 300 | 07 | 69.9 |
| RPN+VGG, shared | 300 | 07+12 | **73.2** |
| RPN+VGG, shared | 300 | COCO+07+12 | **78.8** |

**Table 1 - Table 3 of Detection Results From** (Ren, et al., 2016)

This test demonstrates the newly proposed inclusion of an RPN network for Faster R-CNN greatly increases accuracy of the Faster R-CNN network over its previous implementations such as Fast R-CNN and R-CNN.

Not only does the inclusion of the RPN improve the overall accuracy of predictions from the Faster R-CNN, but it has also improved region proposal times and thus the overall timing of the Faster R-CNN detector which is exemplified with table 5 from the paper as seen in *Table 2* (Ren, et al., 2016).

From the total milliseconds in this table shows the use of an RPN by far decreases the timing of the Faster R-CNN model from 1830ms (1.83 seconds with Selective Search) to 59ms (0.059 seconds with a Region Proposal Network) when using ZF as the RPN's backbone (Ren, et al., 2016).



| model | system | conv | proposal | region-wise | total | rate |
|---|---|---|---|---|---|---|
| VGG | SS + Fast R-CNN | 146 | 1510 | 174 | 1830 | 0.5 fps |
| VGG | RPN + Fast R-CNN | 141 | **10** | 47 | **198** | **5 fps** |
| ZF | RPN + Fast R-CNN | 31 | 3 | 25 | 59 | 17 fps |

**Table 2- Table 5 of Timing Results from Table** (Ren, et al., 2016)

### 2.3.5 EfficientDet

Resuming the description from the introduction, EfficentDet is a relatively new object detecting architecture proposed in 2020 by Mingxing Tan, Ruoming Pang and Quoc V. Le at Google (Tan, et al., 2020).

According to the very same paper that published the EfficientDet architecture, the EfficientDet model is a one-stage detector that makes use of a combination of a pre-trained ImageNet and a convolutional network named EfficientNet as the backbone (Tan, et al., 2020). As evident from *Image 3* (Tan, et al., 2020) EfficientDet makes use of two other components which is a BiFPN used as the feature network that accepts inputs from levels p3 to p7 from the backbone to generate features that are then fused with each adjacent level across the feature network (Tan, et al., 2020). The features that have been fused are then passed to a classifier and box network to associate a class and a predicted bounding box around the objects. (Tan, et al., 2020).

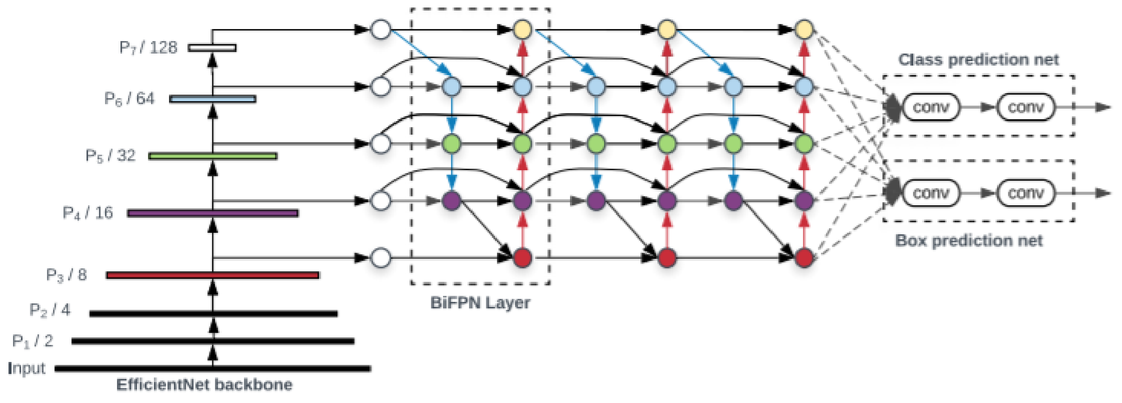
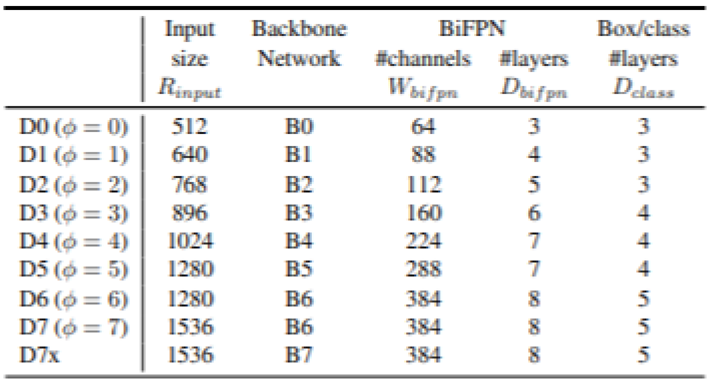


**Image 3 – Efficient Det Architecture Diagram sourced from (Tan, et al., 2020)**

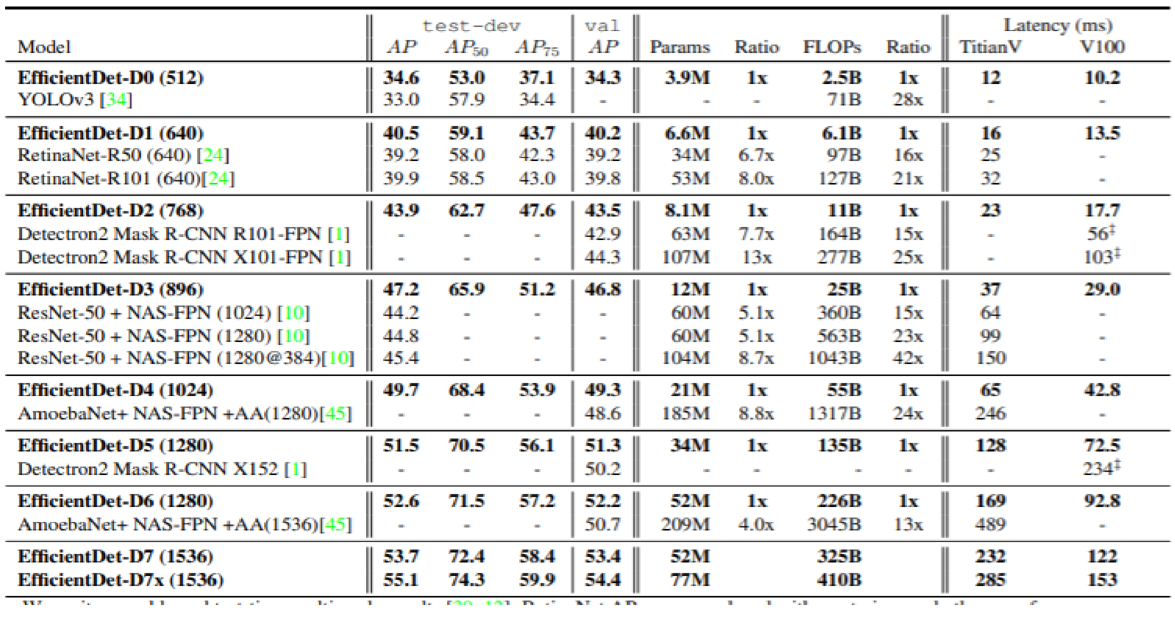
The EfficientDet paper further describes a set of equations defining the strict rules for the scalability of the EfficientDet network before numbering the upscaled versions of EfficientDet fromD0 to D7 with each upscaled version incrementing in input size, EfficientNet backbone size, number of channels and layers as seen in *Table 3* (Tan, et al., 2020).



| | Input size $R_{input}$ | Backbone Network | BiFPN #channels $W_{bifpn}$ | BiFPN #layers $D_{bifpn}$ | Box/class #layers $D_{class}$ |
|---|---|---|---|---|---|
| D0 ($\phi = 0$) | 512 | B0 | 64 | 3 | 3 |
| D1 ($\phi = 1$) | 640 | B1 | 88 | 4 | 3 |
| D2 ($\phi = 2$) | 768 | B2 | 112 | 5 | 3 |
| D3 ($\phi = 3$) | 896 | B3 | 160 | 6 | 4 |
| D4 ($\phi = 4$) | 1024 | B4 | 224 | 7 | 4 |
| D5 ($\phi = 5$) | 1280 | B5 | 288 | 7 | 4 |
| D6 ($\phi = 6$) | 1280 | B6 | 384 | 8 | 5 |
| D7 ($\phi = 7$) | 1536 | B6 | 384 | 8 | 5 |
| D7x | 1536 | B7 | 384 | 8 | 5 |

Table 3 – Table 1 of EffcientDet Scaling Configs sourced from (Tan, et al., 2020)

The EfficientDet paper then proceeds to conduct testing for each consecutive upscaled version to measure their accuracy and latency when running inference while making comparisons to similar object detecting models as best demonstrated by *Table 4* (Tan, et al., 2020).



| Model | test-dev $AP$ | test-dev $AP_{50}$ | test-dev $AP_{75}$ | val $AP$ | Params | Ratio | FLOPs | Ratio | Latency (ms) TitianV | Latency (ms) V100 |
|---|---|---|---|---|---|---|---|---|---|---|
| **EfficientDet-D0 (512)** | **34.6** | **53.0** | **37.1** | **34.3** | **3.9M** | **1x** | **2.5B** | **1x** | **12** | **10.2** |
| YOLOv3 [34] | 33.0 | 57.9 | 34.4 | - | - | - | 71B | 28x | - | - |
| **EfficientDet-D1 (640)** | **40.5** | **59.1** | **43.7** | **40.2** | **6.6M** | **1x** | **6.1B** | **1x** | **16** | **13.5** |
| RetinaNet-R50 (640) [24] | 39.2 | 58.0 | 42.3 | 39.2 | 34M | 6.7x | 97B | 16x | 25 | - |
| RetinaNet-R101 (640)[24] | 39.9 | 58.5 | 43.0 | 39.8 | 53M | 8.0x | 127B | 21x | 32 | - |
| **EfficientDet-D2 (768)** | **43.9** | **62.7** | **47.6** | **43.5** | **8.1M** | **1x** | **11B** | **1x** | **23** | **17.7** |
| Detectron2 Mask R-CNN R101-FPN [1] | - | - | - | 42.9 | 63M | 7.7x | 164B | 15x | - | $56^{‡}$ |
| Detectron2 Mask R-CNN X101-FPN [1] | - | - | - | 44.3 | 107M | 13x | 277B | 25x | - | $103^{‡}$ |
| **EfficientDet-D3 (896)** | **47.2** | **65.9** | **51.2** | **46.8** | **12M** | **1x** | **25B** | **1x** | **37** | **29.0** |
| ResNet-50 + NAS-FPN (1024) [10] | 44.2 | - | - | - | 60M | 5.1x | 360B | 15x | 64 | - |
| ResNet-50 + NAS-FPN (1280) [10] | 44.8 | - | - | - | 60M | 5.1x | 563B | 23x | 99 | - |
| ResNet-50 + NAS-FPN (1280@384)[10] | 45.4 | - | - | - | 104M | 8.7x | 1043B | 42x | 150 | - |
| **EfficientDet-D4 (1024)** | **49.7** | **68.4** | **53.9** | **49.3** | **21M** | **1x** | **55B** | **1x** | **65** | **42.8** |
| AmoebaNet+ NAS-FPN +AA(1280)[45] | - | - | - | 48.6 | 185M | 8.8x | 1317B | 24x | 246 | - |
| **EfficientDet-D5 (1280)** | **51.5** | **70.5** | **56.1** | **51.3** | **34M** | **1x** | **135B** | **1x** | **128** | **72.5** |
| Detectron2 Mask R-CNN X152 [1] | - | - | - | 50.2 | - | - | - | - | - | $234^{‡}$ |
| **EfficientDet-D6 (1280)** | **52.6** | **71.5** | **57.2** | **52.2** | **52M** | **1x** | **226B** | **1x** | **169** | **92.8** |
| AmoebaNet+ NAS-FPN +AA(1536)[45] | - | - | - | 50.7 | 209M | 4.0x | 3045B | 13x | 489 | - |
| **EfficientDet-D7 (1536)** | **53.7** | **72.4** | **58.4** | **53.4** | **52M** | | **325B** | | **232** | **122** |
| **EfficientDet-D7x (1536)** | **55.1** | **74.3** | **59.9** | **54.4** | **77M** | | **410B** | | **285** | **153** |

**Table 4 – Table 2 of EfficientDet Performance sourced from (Tan, et al., 2020)**

From the referenced table above from the paper proposing the EfficientDet architecture it is evident the most accurate upscaled version is D7x which is like D7 but has a larger EfficientNet backbone achieving an average precision of 54% at a latency of 285ms (0.285 seconds) on a TitanV GPU and 153ms (0.153 seconds) on a V100 GPU (Tan, et al., 2020).

However, the worst preforming version is D0 with an average precision of 34% but with the trade-off of 12ms on the TitanV GPU and 10ms on the V100 GPU (Tan, et al., 2020).

### 2.3.6 ResNet

Before going in depth into RetinaNet, a quick overview of what ResNet is should be given.

ResNet or otherwise known as a Residual Network is a single layer network commonly used as the backbone for a vast array of computer vision models (Feng, 2017). ResNet is mainly composed of a chain of convolutional layers that make use of what is called an “Identity Shortcut Connection” as shown in *Image 4* (Feng, 2017) that allows convolutional layers to be skipped avoiding the vanishing gradient problem by allowing data pass through an alternative route to flow (Feng, 2017).

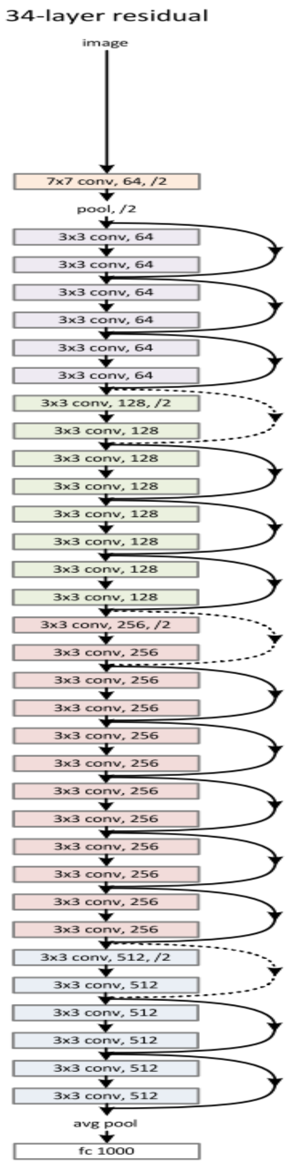


**Image 4 – ResNet Diagram sourced from** (Feng, 2017)

### 2.3.7 RetinaNet

Continuing from the introduction, RetinaNet is another popular object detecting architecture that is a one-stage detection model proposed in 2017 by Tsung-Yi Lin, Priya Goyal, Ross Girshick and Kaiming He at Facebook AI research (Lin, et al., 2018).

In addition to ResNet, RetinaNet makes use of a Feature Pyramid Network (FPN) in its architecture which is exemplified in *Image 6* (Lin, et al., 2016) which creates rich feature maps by combining lower resolution features with higher resolution features to increase the overall accuracy of the model (ArcGis Developers, n.d.).

The RetinaNet architecture makes use of four layers in total as visible in *image 5* (Tsang, 2019). Firstly, the RetinaNet architecture uses a ResNet as its backbone to produce feature maps at each ascending level of the ResNet backbone passing the data into the Feature Pyramid Network. The FPN then merges each generated feature map before passing data to the classification and box networks to provide a label to any detection and adjust the coordinates of any bounding boxes (ArcGis Developers, n.d.).

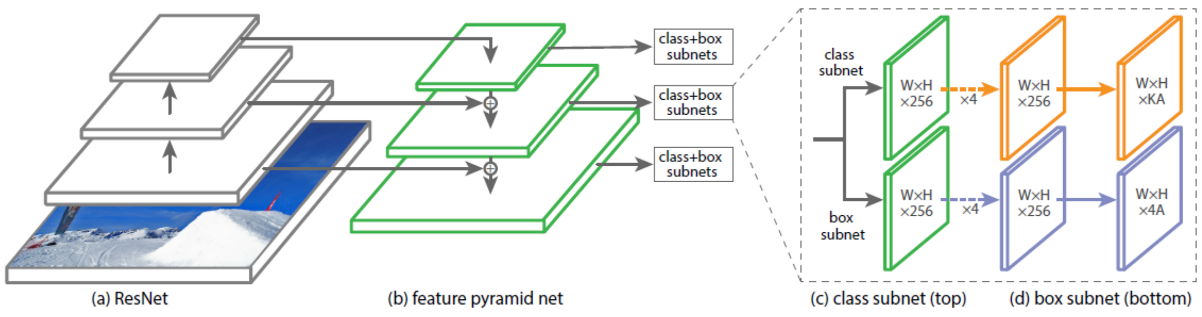


**Image 5 – RetinaNet Architecture Diagram sourced from (Tsang, 2019)**

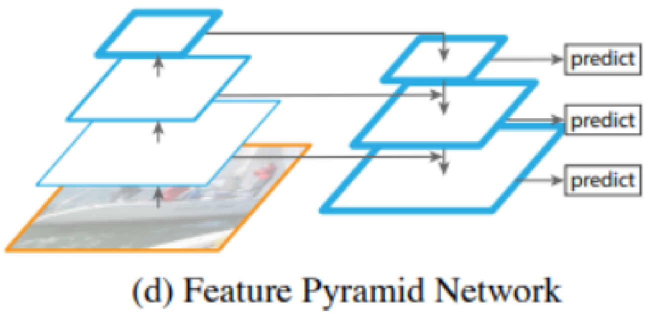
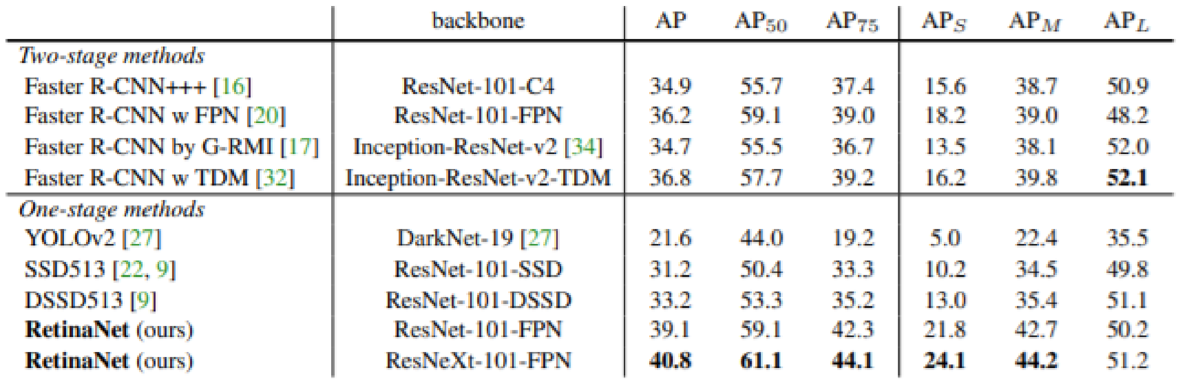


**Image 6 – Feature Pyramid Network sourced from (Lin, et al., 2016)**

From the paper proposing the RetinaNet architecture, several tests have been conducted comparing the average precision of the proposed RetinaNet against similar models shown in *Table 5* (Lin, et al., 2018). Both RetinaNet versions shown appear to outperform all two stage detectors and other one stage detectors with an average precision of 42.7% when using a

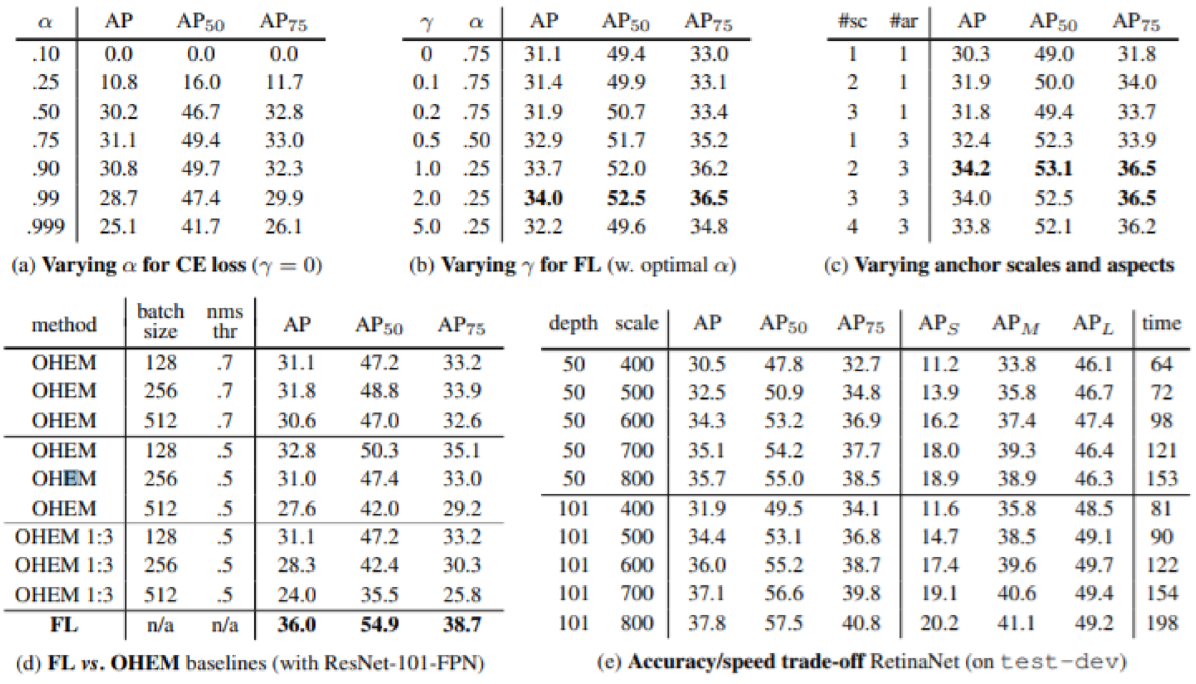
ResNet backbone and an average precision of 44.2% when using a ResNeXt backbone (Lin, et al., 2018).



| | backbone | AP | $AP_{50}$ | $AP_{75}$ | $AP_S$ | $AP_M$ | $AP_L$ |
|---|---|---|---|---|---|---|---|
| *Two-stage methods* | | | | | | | |
| Faster R-CNN+++ [16] | ResNet-101-C4 | 34.9 | 55.7 | 37.4 | 15.6 | 38.7 | 50.9 |
| Faster R-CNN w FPN [20] | ResNet-101-FPN | 36.2 | 59.1 | 39.0 | 18.2 | 39.0 | 48.2 |
| Faster R-CNN by G-RMI [17] | Inception-ResNet-v2 [34] | 34.7 | 55.5 | 36.7 | 13.5 | 38.1 | 52.0 |
| Faster R-CNN w TDM [32] | Inception-ResNet-v2-TDM | 36.8 | 57.7 | 39.2 | 16.2 | 39.8 | **52.1** |
| *One-stage methods* | | | | | | | |
| YOLOv2 [27] | DarkNet-19 [27] | 21.6 | 44.0 | 19.2 | 5.0 | 22.4 | 35.5 |
| SSD513 [22, 9] | ResNet-101-SSD | 31.2 | 50.4 | 33.3 | 10.2 | 34.5 | 49.8 |
| DSSD513 [9] | ResNet-101-DSSD | 33.2 | 53.3 | 35.2 | 13.0 | 35.4 | 51.1 |
| **RetinaNet** (ours) | ResNet-101-FPN | 39.1 | 59.1 | 42.3 | 21.8 | 42.7 | 50.2 |
| **RetinaNet** (ours) | ResNeXt-101-FPN | **40.8** | **61.1** | **44.1** | **24.1** | **44.2** | 51.2 |

**Table 5 – Table 2 of Object Detection Results sourced from** (Lin, et al., 2018)

Furthermore, from *Table 6 (e)* (Lin, et al., 2018) as seen in the paper proposing RetinaNet. The RetinaNet structure performs exorbitantly slowly if compared to the previously mentioned object detection models such as EfficientDet and Faster R-CNN. However, the results from *Table 6 (e)* (Lin, et al., 2018) does show the mean average precision remains high at an image scale of 400 however the mean average improves immensely at an image scale of 800 with a big inference speed drawback of 153ms (0.153s) at a backbone depth of 50 and 198ms (0.198s) at a backbone depth of 101 (Lin, et al., 2018).



| $\alpha$ | AP | $AP_{50}$ | $AP_{75}$ |
|---|---|---|---|
| .10 | 0.0 | 0.0 | 0.0 |
| .25 | 10.8 | 16.0 | 11.7 |
| .50 | 30.2 | 46.7 | 32.8 |
| .75 | 31.1 | 49.4 | 33.0 |
| .90 | 30.8 | 49.7 | 32.3 |
| .99 | 28.7 | 47.4 | 29.9 |
| .999 | 25.1 | 41.7 | 26.1 |

(a) **Varying $\alpha$ for CE loss** ($\gamma = 0$)

| $\gamma$ | $\alpha$ | AP | $AP_{50}$ | $AP_{75}$ |
|---|---|---|---|---|
| 0 | .75 | 31.1 | 49.4 | 33.0 |
| 0.1 | .75 | 31.4 | 49.9 | 33.1 |
| 0.2 | .75 | 31.9 | 50.7 | 33.4 |
| 0.5 | .50 | 32.9 | 51.7 | 35.2 |
| 1.0 | .25 | 33.7 | 52.0 | 36.2 |
| 2.0 | .25 | **34.0** | **52.5** | **36.5** |
| 5.0 | .25 | 32.2 | 49.6 | 34.8 |

(b) **Varying $\gamma$ for FL** (w. optimal $\alpha$)

| #sc | #ar | AP | $AP_{50}$ | $AP_{75}$ |
|---|---|---|---|---|
| 1 | 1 | 30.3 | 49.0 | 31.8 |
| 2 | 1 | 31.9 | 50.0 | 34.0 |
| 3 | 1 | 31.8 | 49.4 | 33.7 |
| 1 | 3 | 32.4 | 52.3 | 33.9 |
| 2 | 3 | **34.2** | **53.1** | **36.5** |
| 3 | 3 | 34.0 | 52.5 | **36.5** |
| 4 | 3 | 33.8 | 52.1 | 36.2 |

(c) **Varying anchor scales and aspects**

| method | batch size | nms thr | AP | $AP_{50}$ | $AP_{75}$ |
|---|---|---|---|---|---|
| OHEM | 128 | .7 | 31.1 | 47.2 | 33.2 |
| OHEM | 256 | .7 | 31.8 | 48.8 | 33.9 |
| OHEM | 512 | .7 | 30.6 | 47.0 | 32.6 |
| OHEM | 128 | .5 | 32.8 | 50.3 | 35.1 |
| OHEM | 256 | .5 | 31.0 | 47.4 | 33.0 |
| OHEM | 512 | .5 | 27.6 | 42.0 | 29.2 |
| OHEM 1:3 | 128 | .5 | 31.1 | 47.2 | 33.2 |
| OHEM 1:3 | 256 | .5 | 28.3 | 42.4 | 30.3 |
| OHEM 1:3 | 512 | .5 | 24.0 | 35.5 | 25.8 |
| **FL** | n/a | n/a | **36.0** | **54.9** | **38.7** |

(d) **FL *vs.* OHEM baselines** (with ResNet-101-FPN)

| depth | scale | AP | $AP_{50}$ | $AP_{75}$ | $AP_S$ | $AP_M$ | $AP_L$ | time |
|---|---|---|---|---|---|---|---|---|
| 50 | 400 | 30.5 | 47.8 | 32.7 | 11.2 | 33.8 | 46.1 | 64 |
| 50 | 500 | 32.5 | 50.9 | 34.8 | 13.9 | 35.8 | 46.7 | 72 |
| 50 | 600 | 34.3 | 53.2 | 36.9 | 16.2 | 37.4 | 47.4 | 98 |
| 50 | 700 | 35.1 | 54.2 | 37.7 | 18.0 | 39.3 | 46.4 | 121 |
| 50 | 800 | 35.7 | 55.0 | 38.5 | 18.9 | 38.9 | 46.3 | 153 |
| 101 | 400 | 31.9 | 49.5 | 34.1 | 11.6 | 35.8 | 48.5 | 81 |
| 101 | 500 | 34.4 | 53.1 | 36.8 | 14.7 | 38.5 | 49.1 | 90 |
| 101 | 600 | 36.0 | 55.2 | 38.7 | 17.4 | 39.6 | 49.7 | 122 |
| 101 | 700 | 37.1 | 56.6 | 39.8 | 19.1 | 40.6 | 49.4 | 154 |
| 101 | 800 | 37.8 | 57.5 | 40.8 | 20.2 | 41.1 | 49.2 | 198 |

(e) **Accuracy/speed trade-off** RetinaNet (on `test-dev`)

**Table 6 – Table 1of Abolition Experiments sourced from** (Lin, et al., 2018)

## 2.4 Conclusion

After completing all secondary research objectives from the literature and technology review a few conclusions can be made before moving on to primary research.

As the implementation of all primary research this project will be taking place on online cloud services, it can be concluded the use of RoboFlow and Google Colab will serve perfectly for this project.

Even though the handgun dataset found on RoboFlow was uploaded by a third party referencing the work done by the research group at University of Granada studying weapons detection for security and video surveillance it is still a good idea to consider this pistol dataset as it has a plentiful supply of 2973 training and or testing images (RoboFlow, 2020).

After conducting research on the three object detecting architectures considered for this project an initial comparison can be made to propose which one of the three models appears to be the most efficient from the results discussed from their respective proposal papers.

Overall, it can be estimated that the EfficientDet model is the fastest when comparing EfficientDet D0's shortest execution time in *Table 4* (Tan, et al., 2020) of 10ms against RetinaNet50's 64ms in *Table 6*(e) (Lin, et al., 2018) and Faster R-CNN using a ZF backbone 59ms in *Table 1* (Ren, et al., 2016).

However, Faster R-CNN appears to be the most accurate with the highest mean average precision of 78.8% in *Table 1* (Ren, et al., 2016) against EfficientDet's D7x 54.4% in *Table 2* (Tan, et al., 2020) and RetinaNet's 44.2% in *Table 5* (Lin, et al., 2018).

Faster R-CNN may be the most efficient architecture as it has the largest mean average precision with the trade-off being that it is the second-best architecture in the terms of execution times.

# 3.0 Execution

This section will cover and discuss all the primary research conducted during this project's lifecycle to satisfy the proposed research question. The research and development methods of this section will be defined and explained in full.

## *3.1 Research Type*

This project is a capstone style project meaning all the primary research is based around a "develop and test" methodology that involves developing a final prototype that can be evaluated and tested against strict criteria with the intent of answering a hypothesis.

## *3.2 Problem Analysis*

The main aim of this project is to source and create datasets for training and testing purposes before moving on to create a Python development environment that can facilitate the implementation of TensorFlow, TensorFlow's Object Detection API and the training, evaluation, exporting and testing of the three object detecting architectures (Faster R-CNN, EfficientDet and RetinaNet) to answer this project's research question.

Post development of this project's prototype, all the raw analytical data gathered from the training and testing processes will be processed into information in the form of tables so that conclusions can be drawn from primary research alongside the conclusions made at the end of the literature and technologies review to confirm which one of the three object detecting architectures is the most efficient.

## *3.3 Development Methodology*

This section will describe the style of development lifecycle followed by this project alongside an in-depth description of all the stages of development taken during the primary research of this project.

Due to the simple nature of this project only requiring a few deliverables and a linear set of primary objectives, all stages of this project's prototype implementation can be safely broken down into four steps to conform to a waterfall style of development.

The waterfall development methodology is a linear methodology that composes of several distinct phases of development (Reddy, 2019). Each phase of development is executed in a sequential manner requiring the work of the previous phase of development to complete before the next phase of development can begin (Reddy, 2019).

The waterfall development methodology is not a perfect development methodology and has a few undesirable drawbacks when compared to less linear and more adapting styles of development methodologies like Agile, DevOps and Scrum due to the fact that a waterfall style project cannot make snap changes to a project requirements without impacting the overall execution of the project as a whole all while also being more likely to encounter delays as each phase of development becomes too dependent on the previous development phases to carry out any results (Reddy, 2019).

### 3.3.1 Development Phases
With each primary objective broken down to conform to a waterfall style of development, each phase of development can be summed up as the following.

#### Phase 1. Requirements
The requirements stage of involves the translation of this project's primary objectives into system requirements for the design and implementation of this project's prototype to follow.

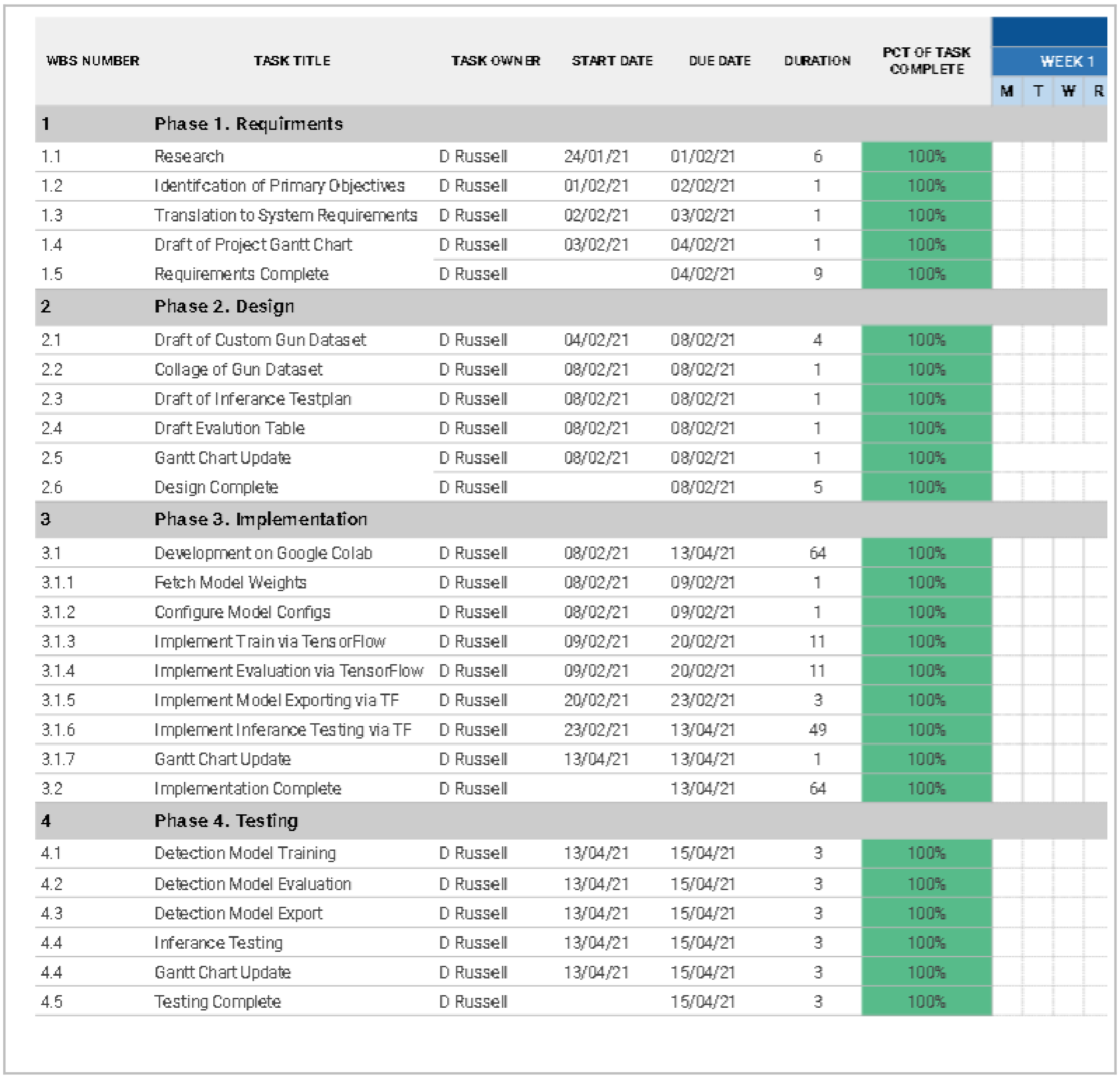
#### Phase 2. Design
The Design stage involves the overall design of this project by drafting a set of testing and evaluation tables for use during the testing stage of development alongside making and labelling a custom gun dataset for use in inference testing of the trained object detecting models.

#### Phase 3. Implementation
The implementation stage involves the overall development of the Python notebook on Google Collab by first implementing the TensorFlow environment, downloading the sourced handgun dataset for training and download the associated model weights from TensorFlow's Model Zoo before implementing how these models will be trained, evaluated and tested.

#### Phase 4. Testing
The testing stage of development involves gathering raw statistical data during the training process of the implemented object detecting models alongside recording data generated via TensorFlow's evaluation script after a successful train to derive more data that can be used during evaluation.

The testing stage consists of all the inference testing for each model made to run inference on the custom-made gun dataset created during the design stage to gauge how successful are the trained models in practice by recording all inference results onto the test plan derived from the design stage for later evaluation.

### 3.3.2 Gantt Chart
In order to track and schedule the overall work required for the project, the use of a Gantt Chart has been employed to keep track of each development phase while following the waterfall method of development as seen in *Figure 1*.



| WBS NUMBER | TASK TITLE | TASK OWNER | START DATE | DUE DATE | DURATION | PCT OF TASK COMPLETE | WEEK 1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | M | T | W | R |
| **1** | **Phase 1. Requirments** | | | | | | | | | |
| 1.1 | Research | D Russell | 24/01/21 | 01/02/21 | 6 | 100% | | | | |
| 1.2 | Identifcation of Primary Objectives | D Russell | 01/02/21 | 02/02/21 | 1 | 100% | | | | |
| 1.3 | Translation to System Requirements | D Russell | 02/02/21 | 03/02/21 | 1 | 100% | | | | |
| 1.4 | Draft of Project Gantt Chart | D Russell | 03/02/21 | 04/02/21 | 1 | 100% | | | | |
| 1.5 | Requirements Complete | D Russell | | 04/02/21 | 9 | 100% | | | | |
| **2** | **Phase 2. Design** | | | | | | | | | |
| 2.1 | Draft of Custom Gun Dataset | D Russell | 04/02/21 | 08/02/21 | 4 | 100% | | | | |
| 2.2 | Collage of Gun Dataset | D Russell | 08/02/21 | 08/02/21 | 1 | 100% | | | | |
| 2.3 | Draft of Inferance Testplan | D Russell | 08/02/21 | 08/02/21 | 1 | 100% | | | | |
| 2.4 | Draft Evalution Table | D Russell | 08/02/21 | 08/02/21 | 1 | 100% | | | | |
| 2.5 | Gantt Chart Update | D Russell | 08/02/21 | 08/02/21 | 1 | 100% | | | | |
| 2.6 | Design Complete | D Russell | | 08/02/21 | 5 | 100% | | | | |
| **3** | **Phase 3. Implementation** | | | | | | | | | |
| 3.1 | Development on Google Colab | D Russell | 08/02/21 | 13/04/21 | 64 | 100% | | | | |
| 3.1.1 | Fetch Model Weights | D Russell | 08/02/21 | 09/02/21 | 1 | 100% | | | | |
| 3.1.2 | Configure Model Configs | D Russell | 08/02/21 | 09/02/21 | 1 | 100% | | | | |
| 3.1.3 | Implement Train via TensorFlow | D Russell | 09/02/21 | 20/02/21 | 11 | 100% | | | | |
| 3.1.4 | Implement Evaluation via TensorFlow | D Russell | 09/02/21 | 20/02/21 | 11 | 100% | | | | |
| 3.1.5 | Implement Model Exporting via TF | D Russell | 20/02/21 | 23/02/21 | 3 | 100% | | | | |
| 3.1.6 | Implement Inferance Testing via TF | D Russell | 23/02/21 | 13/04/21 | 49 | 100% | | | | |
| 3.1.7 | Gantt Chart Update | D Russell | 13/04/21 | 13/04/21 | 1 | 100% | | | | |
| 3.2 | Implementation Complete | D Russell | | 13/04/21 | 64 | 100% | | | | |
| **4** | **Phase 4. Testing** | | | | | | | | | |
| 4.1 | Detection Model Training | D Russell | 13/04/21 | 15/04/21 | 3 | 100% | | | | |
| 4.2 | Detection Model Evaluation | D Russell | 13/04/21 | 15/04/21 | 3 | 100% | | | | |
| 4.3 | Detection Model Export | D Russell | 13/04/21 | 15/04/21 | 3 | 100% | | | | |
| 4.4 | Inferance Testing | D Russell | 13/04/21 | 15/04/21 | 3 | 100% | | | | |
| 4.4 | Gantt Chart Update | D Russell | 13/04/21 | 15/04/21 | 3 | 100% | | | | |
| 4.5 | Testing Complete | D Russell | | 15/04/21 | 3 | 100% | | | | |

Figure 1 - Project Gantt Chart

## *3.4 Project Requirements (Phase 1.)*

As described in the development methodology section, this phase of development will translate all primary objectives defined from the introduction section into functional requirements.

### 3.4.1 Functional Requirements

To summarise the primary objectives from the introduction section, they can be listed as the following.

- Collate a suitable dataset of images and process with labels.
- Develop a test plan.
- Acquire detection model weights.
- Train models using TensorFlow.
- Test trained models

After some reflection on the primary objectives, the functional requirements of this project's prototype and be translated by determining what mandatory features that must be implemented to meet these primary objectives to answer the research question.

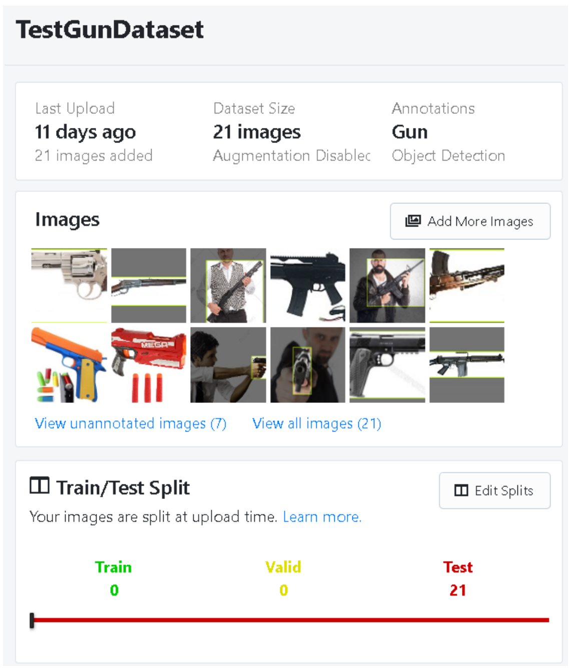
- Implementation must pull the Handgun dataset sourced from the University of Granada found on RoboFlow.
- Implementation must pull model weights of Faster R-CNN, EfficientDet and RetinaNet from TensorFlow's Model Zoo.
- Implementation must configure pulled model weights for training and evaluation.
- Implementation must be able to train a new set of objects detecting model via TensorFlow using the weights acquired from TensorFlow's Model Zoo.
- Implementation must be able to conduct inference testing with the newly trained set of objects detecting models via TensorFlow.
- Implementation must be able to evaluate the newly trained models for additional data.

## *3.5 Project Design (Phase 2)*

As described in the development methodology section, this phase of development will draft a custom gun dataset for inference testing of the trained object detection models followed by drafting a set of inference testing test plans, an evaluation table for post execution evaluation.

### 3.5.1 Custom Gun Dataset

Making use of RoboFlow and Google, twenty-one images have been acquired with a mixture of gun and toy gun examples, a full numbered collage of which can be seen in *Figure 3*. This small custom testing dataset as visible in *Figure 2* makes use of one annotation class and the train, test split is entirely dedicated to testing object detecting models once they have been successfully trained.



TestGunDataset

Last Upload: 11 days ago, 21 images added
Dataset Size: 21 images, Augmentation Disablec
Annotations: Gun, Object Detection

Images

View unannotated images (7) View all images (21)

Train/Test Split

Your images are split at upload time. Learn more.

Train 0
Valid 0
Test 21

Figure 2 - Custom Gun Dataset RoboFlow

The purpose of creating such a dataset composed of a mixture of toy and real guns of varying types and sizes is to test the ability of these object detecting models when handling unfamiliar data to see if the models can differentiate between different types of real guns and toy guns characterized by bright colours and orange tips.

### 3.5.2 Gun Dataset Collage

As can be seen in *Figure 3,* a selection of toy guns, handguns, historical firearms and other more modern firearms have been collated together in a dataset of 21 images.

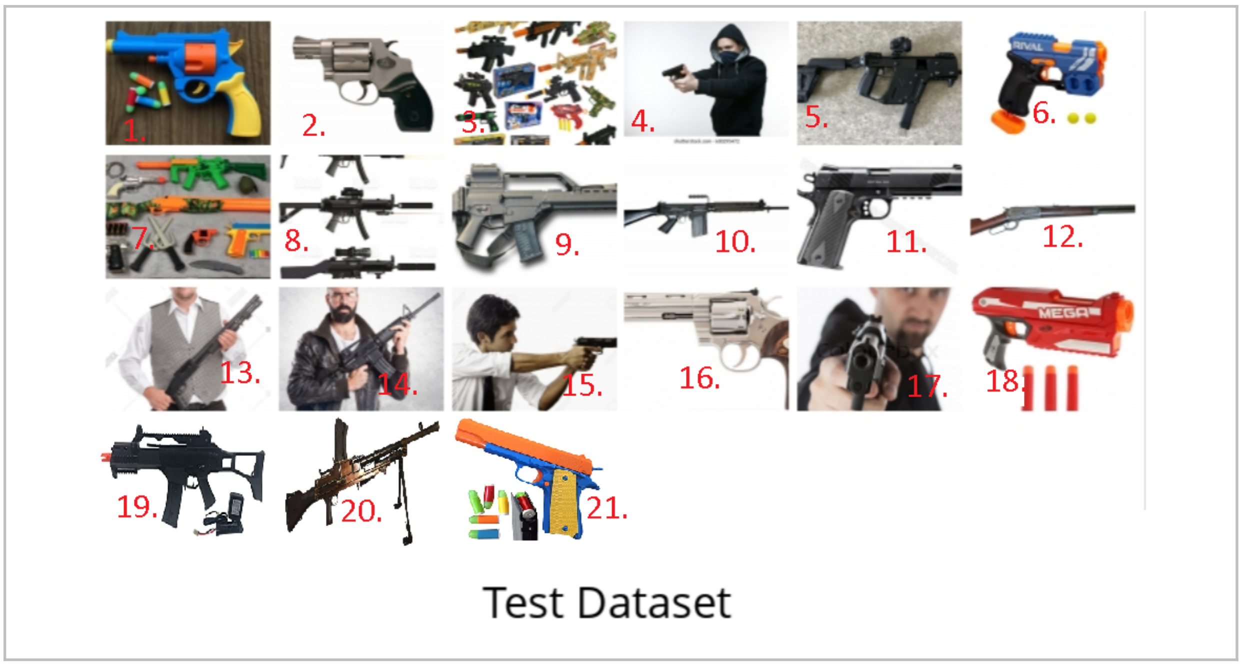
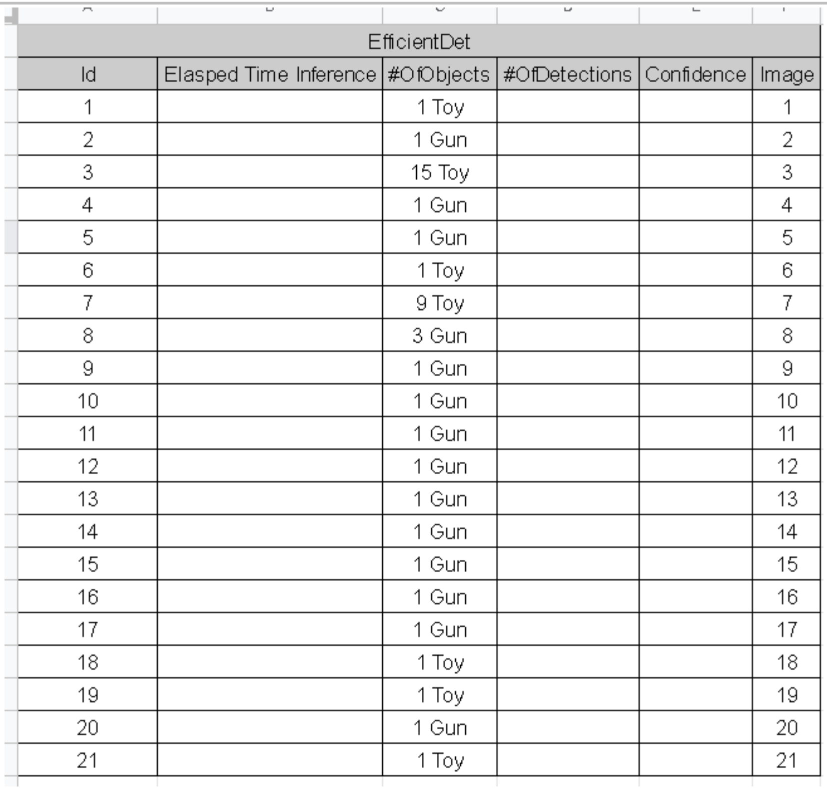


Figure 3 - Collage of Custom Gun Test Images

### 3.5.3 Inference Testing Test Plan

Making use of Google Sheets an inference test plan and evaluation table has been drafted for use with each trained object detection model as visible in *Figure 4,5*.

The inference test plan in *Figure 4* will be used during inference testing on the custom gun dataset as displayed in *Figure 3* to record the total elapsed time per inference made, number of detections made vs the actual number of object and confidence of each detection.



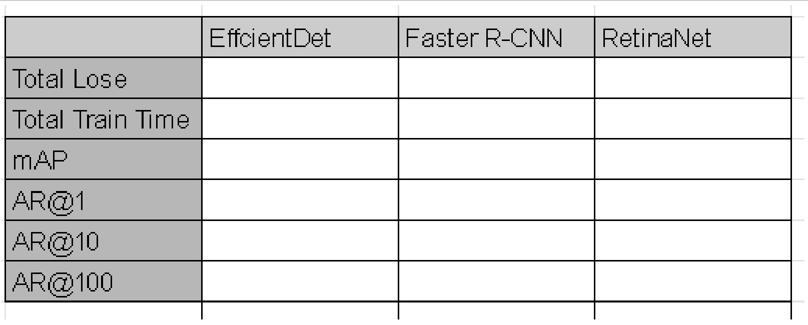
| EfficientDet | | | | | |
|---|---|---|---|---|---|
| Id | Elasped Time Inference | #OfObjects | #OfDetections | Confidence | Image |
| 1 | | 1 Toy | | | 1 |
| 2 | | 1 Gun | | | 2 |
| 3 | | 15 Toy | | | 3 |
| 4 | | 1 Gun | | | 4 |
| 5 | | 1 Gun | | | 5 |
| 6 | | 1 Toy | | | 6 |
| 7 | | 9 Toy | | | 7 |
| 8 | | 3 Gun | | | 8 |
| 9 | | 1 Gun | | | 9 |
| 10 | | 1 Gun | | | 10 |
| 11 | | 1 Gun | | | 11 |
| 12 | | 1 Gun | | | 12 |
| 13 | | 1 Gun | | | 13 |
| 14 | | 1 Gun | | | 14 |
| 15 | | 1 Gun | | | 15 |
| 16 | | 1 Gun | | | 16 |
| 17 | | 1 Gun | | | 17 |
| 18 | | 1 Toy | | | 18 |
| 19 | | 1 Toy | | | 19 |
| 20 | | 1 Gun | | | 20 |
| 21 | | 1 Toy | | | 21 |

Figure 4- Blank Inference Test Plan

### 3.5.4 Evaluation Table

The evaluation table as visible in *Figure 5* is used to record statistics displayed post running evaluation script from the object detection API after each model has been successfully trained.

The evaluation table will be used to record the total lose and elapsed time during training and the performance metrics mean average precision and average recall will be gathered from an evaluation script available in TensorFlow's Object Detection API.



| | EffcientDet | Faster R-CNN | RetinaNet |
|---|---|---|---|
| Total Lose | | | |
| Total Train Time | | | |
| mAP | | | |
| AR@1 | | | |
| AR@10 | | | |
| AR@100 | | | |

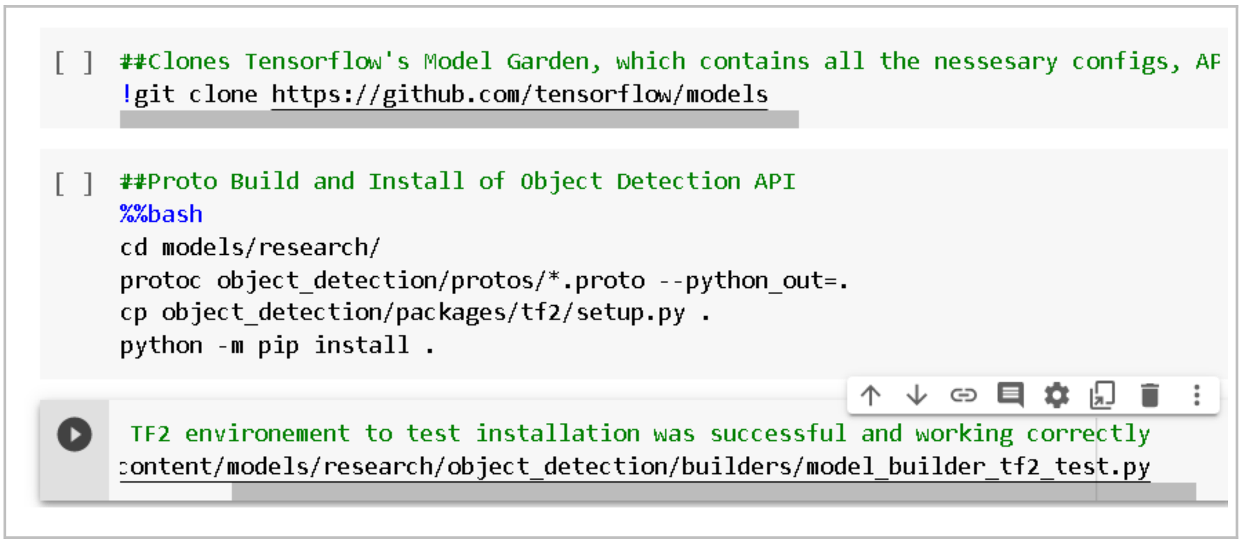
Figure 5- Blank Evaluation Table

## *3.6 Project Implementation (Phase 3)*

As described in the methodology section, the implementation stage will cover all the relevant implementation to this report in relation to the set-up of the TensorFlow environment, downloading the before mention handgun dataset from RoboFlow, downloading the pre-trained weights from TensorFlow's Model Zoo before configuring the weights into a new set of models to train from scratch, exporting the model for deployment before finally discussing the inference testing section of code.

Starting with *Code 1*, the TensorFlow environment can be quickly set-up via TensorFlow's object detection API. The git clone present in *Code 1* clones the TensorFlow's Model Garden repository before changing directory to where the object detection API is kept. By using Google's protoc or protocol buffer already installed on Google Colab, all scripts required for the object detection API are compiled from proto's to Python scripts that are ready for use.

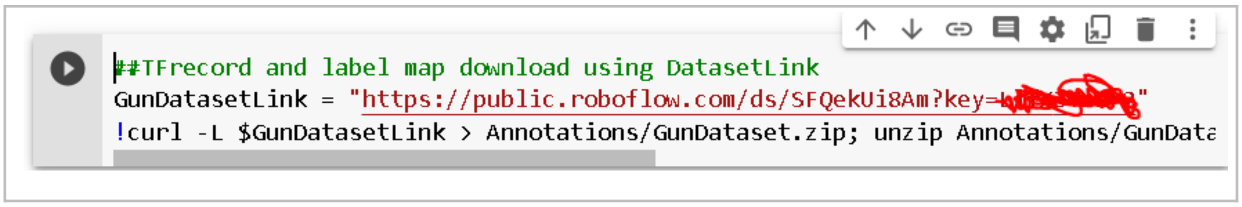
After compiling the proto's into Python scripts from *Code 1*, "setup.py" can then be invoked to quickly update TensorFlow and download all the necessary libraries required for object detection. Once the setup is complete "model_builder_tf2_test.py" can be invoked to verify that the installed TensorFlow environment can implement and test object detecting model.



```
[ ]  ##Clones Tensorflow's Model Garden, which contains all the nessesary configs, AP
     !git clone https://github.com/tensorflow/models

[ ]  ##Proto Build and Install of Object Detection API
     %%bash
     cd models/research/
     protoc object_detection/protos/*.proto --python_out=.
     cp object_detection/packages/tf2/setup.py .
     python -m pip install .

      TF2 environement to test installation was successful and working correctly
     ontent/models/research/object_detection/builders/model_builder_tf2_test.py
```

Code 1- Object Detection API Setup & Test Code Section

Moving on to the next relevant section of code during the implementation phase, *Code 2* makes use of the curl command to pull and export the handgun dataset from RoboFlow in the form of a zipped TFRecord which is the format needed for TensorFlow's object detection API scripts along with an associated label map that will also be stored in the Annotations folder.



```
##TFrecord and label map download using DatasetLink
GunDatasetLink = "https://public.roboflow.com/ds/SFQekUi8Am?key=[redacted]"
!curl -L $GunDatasetLink > Annotations/GunDataset.zip; unzip Annotations/GunData
```

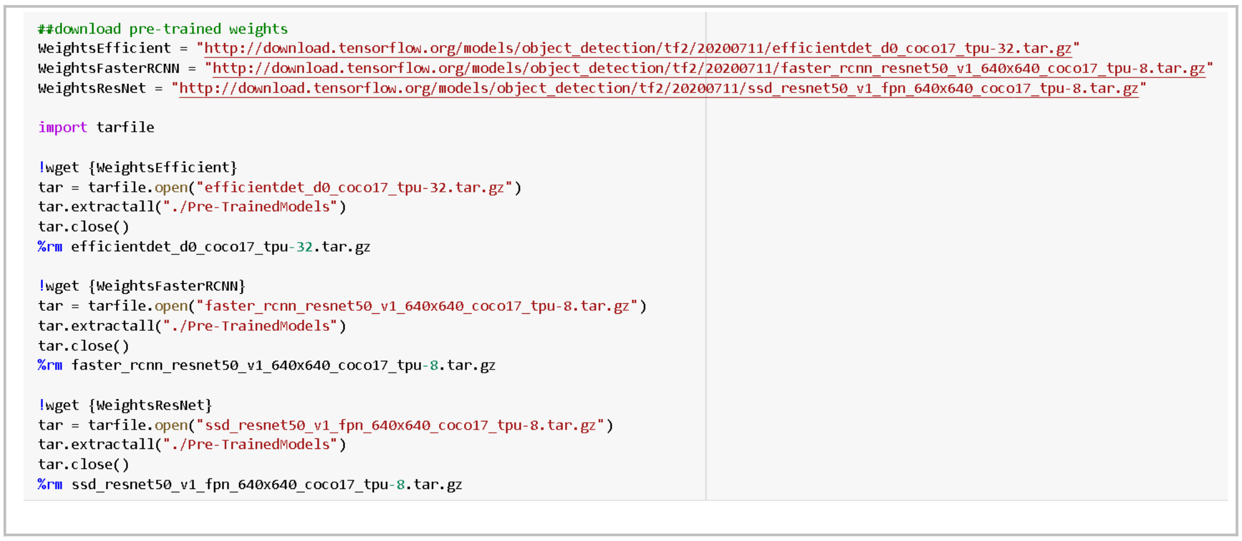
Code 2- RoboFlow Pistol Dataset Download Code Section

With the handgun dataset acquired and ready for training, the model weights for each object detection model (Faster R-CNN, EfficientDet and RetinaNet) are downloaded from TensorFlow's Model Zoo via the wget command as seen in *Code 3*. The contents of each model weights are then stored into a temporary folder named "Pre-trainedModels" for later extraction.

Due to machine resource limitations on Google Colab, the lightest versions of EfficentDet (D0), Faster R-CNN (Resnet 50) and RetinaNet (Resnet 50) have been chosen for training and testing.

EfficientDet D0 is the lightest variant of EfficientDet as it has the smallest image scale and backbone of all other upscaled variable of EfficentDet as seen in *Table 3* referenced in the literature review.

Faster R-CNN Resnet 50 and RetinaNet Resnet 50 are the smallest of the versions available on TensorFlow's Model Zoo due to these two architectures making use of a Resnet with a depth of 50 convolutional layer as their backbone.

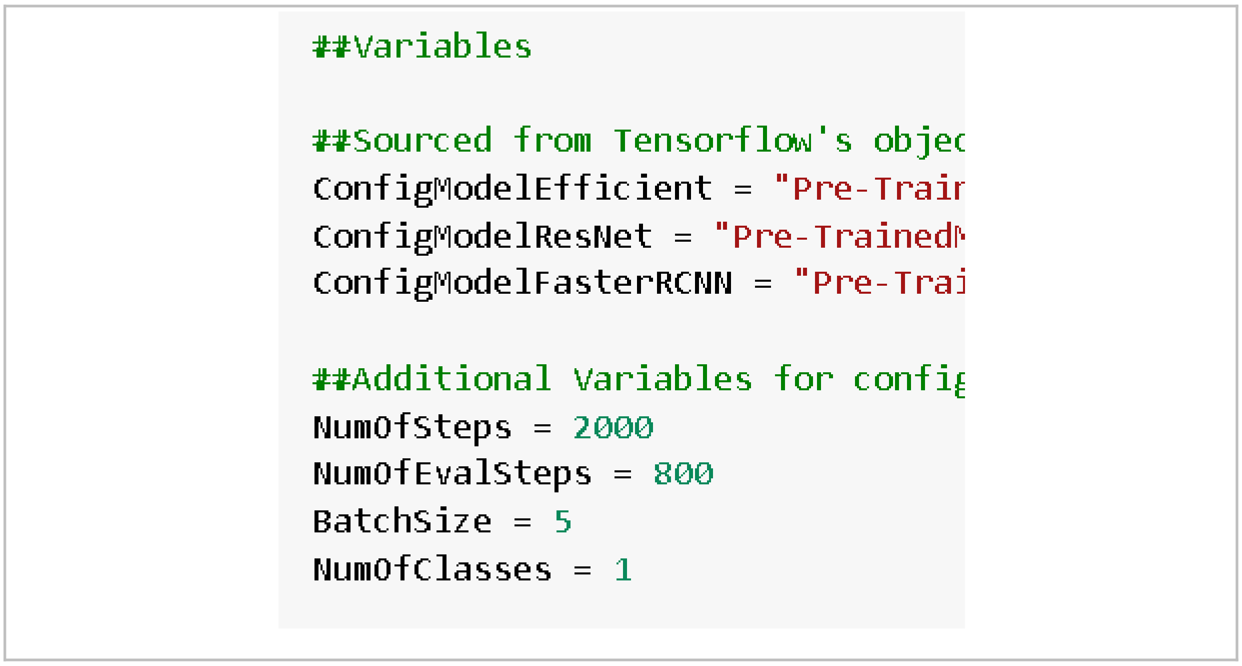


```
##download pre-trained weights
WeightsEfficient = "http://download.tensorflow.org/models/object_detection/tf2/20200711/efficientdet_d0_coco17_tpu-32.tar.gz"
WeightsFasterRCNN = "http://download.tensorflow.org/models/object_detection/tf2/20200711/faster_rcnn_resnet50_v1_640x640_coco17_tpu-8.tar.gz"
WeightsResNet = "http://download.tensorflow.org/models/object_detection/tf2/20200711/ssd_resnet50_v1_fpn_640x640_coco17_tpu-8.tar.gz"

import tarfile

!wget {WeightsEfficient}
tar = tarfile.open("efficientdet_d0_coco17_tpu-32.tar.gz")
tar.extractall("./Pre-TrainedModels")
tar.close()
%rm efficientdet_d0_coco17_tpu-32.tar.gz

!wget {WeightsFasterRCNN}
tar = tarfile.open("faster_rcnn_resnet50_v1_640x640_coco17_tpu-8.tar.gz")
tar.extractall("./Pre-TrainedModels")
tar.close()
%rm faster_rcnn_resnet50_v1_640x640_coco17_tpu-8.tar.gz

!wget {WeightsResNet}
tar = tarfile.open("ssd_resnet50_v1_fpn_640x640_coco17_tpu-8.tar.gz")
tar.extractall("./Pre-TrainedModels")
tar.close()
%rm ssd_resnet50_v1_fpn_640x640_coco17_tpu-8.tar.gz
```

Code 3 - Model Zoo Weights Download Code Section

*Code 4* is where the properties for training and evaluation are defined in the form of "NumOfSteps" which is the number of steps total for an executed training operation, "NumOfEvalStep" is the number of steps for a model evaluation, "BatchSize" is the number of images fed into the model during each training step and "NumofClasses" is the number of annotation classes for which the model is being trained from the previously downloaded handgun label map to detect which would be one class.

Due to restrictions of machine resources on Google Colab, the highest batch size achievable on Google Colab is 5 and the highest number of step achievable on Google Colab is 2000 due to Google Colab being strict on how long a program can execute on Google Colab's servers.

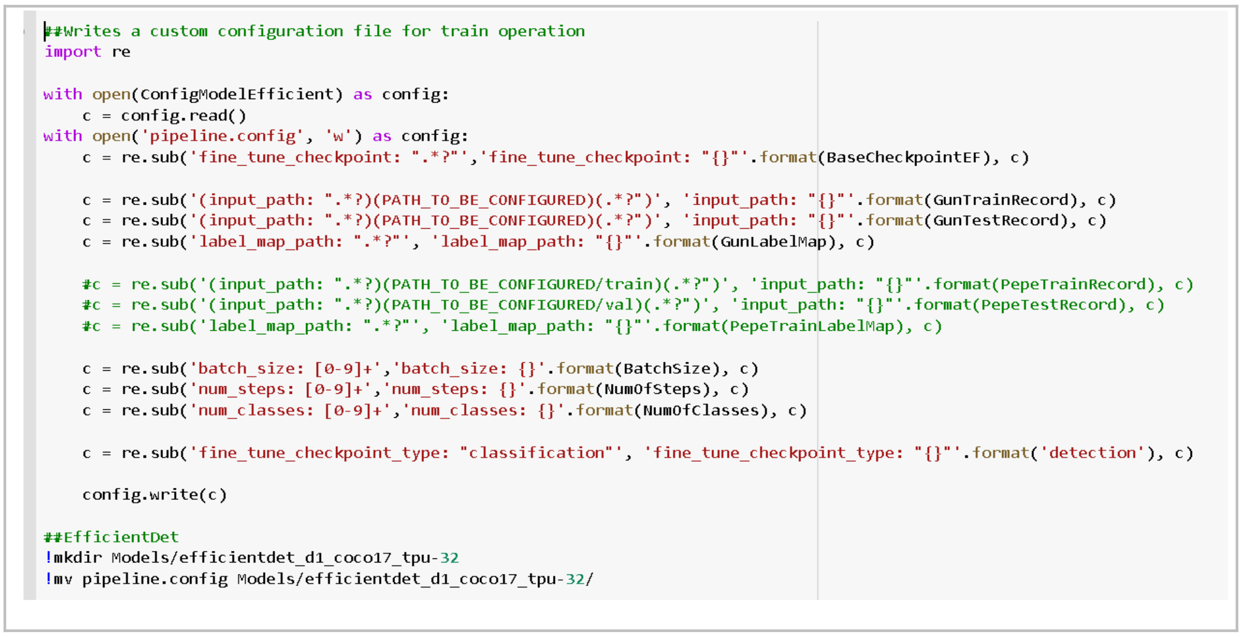


```
##Variables

##Sourced from Tensorflow's objec
ConfigModelEfficient = "Pre-Trair
ConfigModelResNet = "Pre-TrainedN
ConfigModelFasterRCNN = "Pre-Trai

##Additional Variables for config
NumOfSteps = 2000
NumOfEvalSteps = 800
BatchSize = 5
NumOfClasses = 1
```

Code 4 - Model Configuration Properties

In order to configure each pre-trained model weight to train from scratch using a new dataset (the handgun dataset acquired from RoboFlow) the detection model's configuration file must be modified to allow for these changes.

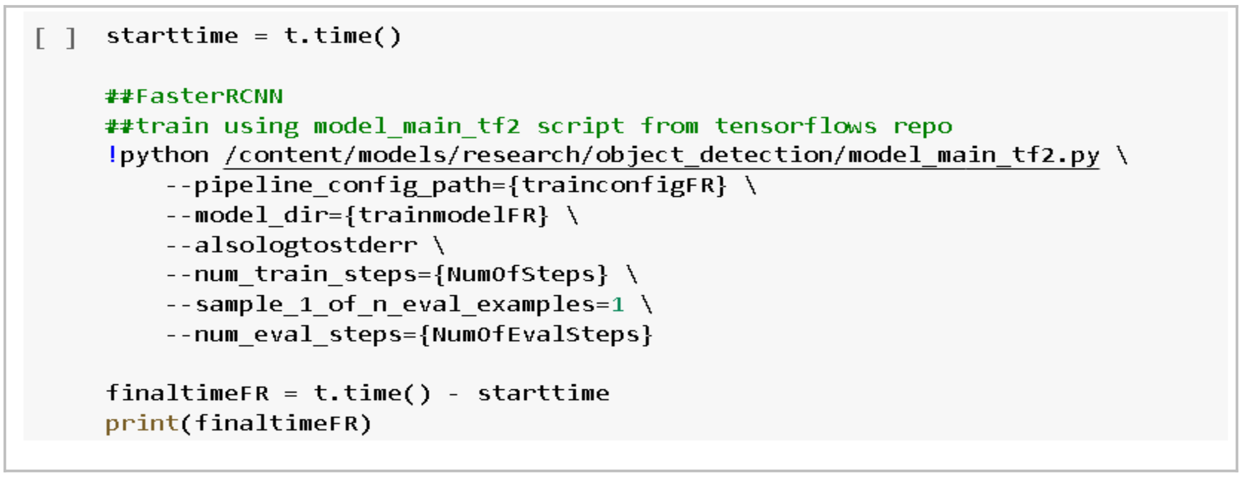
By using "config.read" as seen in *Code 5,* the configuration file content can be copied for modification via "re.sub" as seen in *Code 5* which substitutes the desired training properties, TFRecords and Label map into the new config file. Lastly the "Fine tune checkpoint" in *Code 5* is set to the last checkpoint present in each of the folder of each pre-trained model weight to act as a starting point for the new object detecting model.



```
##Writes a custom configuration file for train operation
import re

with open(ConfigModelEfficient) as config:
    c = config.read()
with open('pipeline.config', 'w') as config:
    c = re.sub('fine_tune_checkpoint: ".*?"','fine_tune_checkpoint: "{}"'.format(BaseCheckpointEF), c)

    c = re.sub('(input_path: ".*?)(PATH_TO_BE_CONFIGURED)(.*?")', 'input_path: "{}"'.format(GunTrainRecord), c)
    c = re.sub('(input_path: ".*?)(PATH_TO_BE_CONFIGURED)(.*?")', 'input_path: "{}"'.format(GunTestRecord), c)
    c = re.sub('label_map_path: ".*?"', 'label_map_path: "{}"'.format(GunLabelMap), c)

    #c = re.sub('(input_path: ".*?)(PATH_TO_BE_CONFIGURED/train)(.*?")', 'input_path: "{}"'.format(PepeTrainRecord), c)
    #c = re.sub('(input_path: ".*?)(PATH_TO_BE_CONFIGURED/val)(.*?")', 'input_path: "{}"'.format(PepeTestRecord), c)
    #c = re.sub('label_map_path: ".*?"', 'label_map_path: "{}"'.format(PepeTrainLabelMap), c)

    c = re.sub('batch_size: [0-9]+','batch_size: {}'.format(BatchSize), c)
    c = re.sub('num_steps: [0-9]+','num_steps: {}'.format(NumOfSteps), c)
    c = re.sub('num_classes: [0-9]+','num_classes: {}'.format(NumOfClasses), c)

    c = re.sub('fine_tune_checkpoint_type: "classification"', 'fine_tune_checkpoint_type: "{}"'.format('detection'), c)

    config.write(c)

##EfficientDet
!mkdir Models/efficientdet_d1_coco17_tpu-32
!mv pipeline.config Models/efficientdet_d1_coco17_tpu-32/
```

Code 5 - Configuration File Creation Code Section

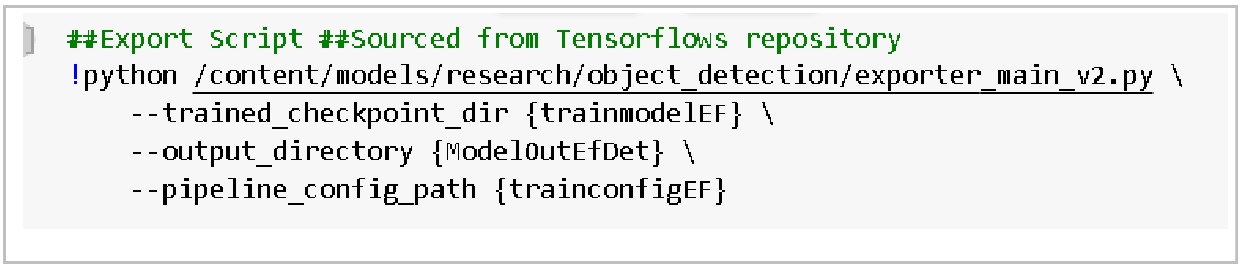
To begin training, TensorFlow's Object Detection API script "model_main_tf2.py" must be invoked with the following properties as seen in *Code 6.* The pipeline path refers to the configuration file associated to the model being trained alongside the new directory the model is to be trained to. The use of "t.time" is to calculate the total elapsed time during training the associated model.



```
[ ]  starttime = t.time()

     ##FasterRCNN
     ##train using model_main_tf2 script from tensorflows repo
     !python /content/models/research/object_detection/model_main_tf2.py \
         --pipeline_config_path={trainconfigFR} \
         --model_dir={trainmodelFR} \
         --alsologtostderr \
         --num_train_steps={NumOfSteps} \
         --sample_1_of_n_eval_examples=1 \
         --num_eval_steps={NumOfEvalSteps}

     finaltimeFR = t.time() - starttime
     print(finaltimeFR)
```

Code 6 - TensorFlow API Train Scrip

In *Code 7*, this section takes the latest checkpoint and config of the newly trained model to export the latest checkpoint as its own separate model to be saved in folder "Exported-Models".



```
##Export Script ##Sourced from Tensorflows repository
!python /content/models/research/object_detection/exporter_main_v2.py \
    --trained_checkpoint_dir {trainmodelEF} \
    --output_directory {ModelOutEfDet} \
    --pipeline_config_path {trainconfigEF}
```

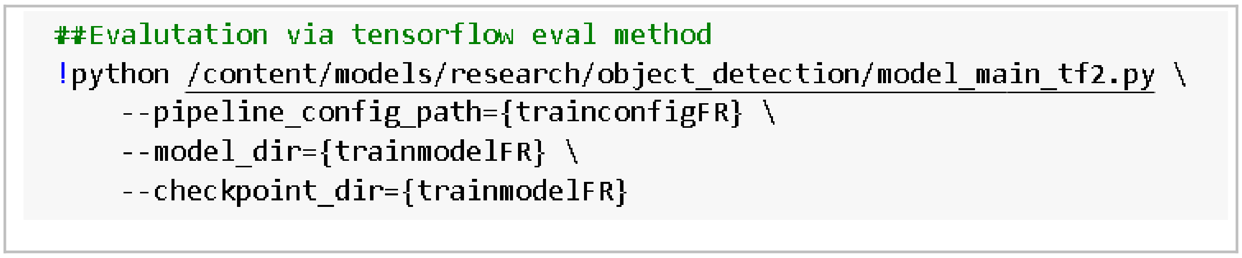
Code 7- TensorFlow API Exporter Script

## *3.7 Project Testing (Phase 4)*

As described in the methodology section, the testing stage composes of inference testing and the use of TensorFlow's evaluation script post training to derive additional data to gauge the overall performance of each trained model. Testing will make use of the tables defined in *Figure 4,5* to record key data required for evaluation.

During the training phase of testing, the evaluation table defined in *Figure 5* will be used to record the total elapsed time from when training starts to when training ends for each object detecting model. The evaluation table *Figure 5* will then be filled by data retrieved by running evaluation on each successfully trained model by using the code snippet in *Code 6*.

In order to run evaluation on a recently trained model the very same training script "model_main_tf2.py" must be invoked with only the model's configuration file, model directory and directory of the latest checkpoint from training.

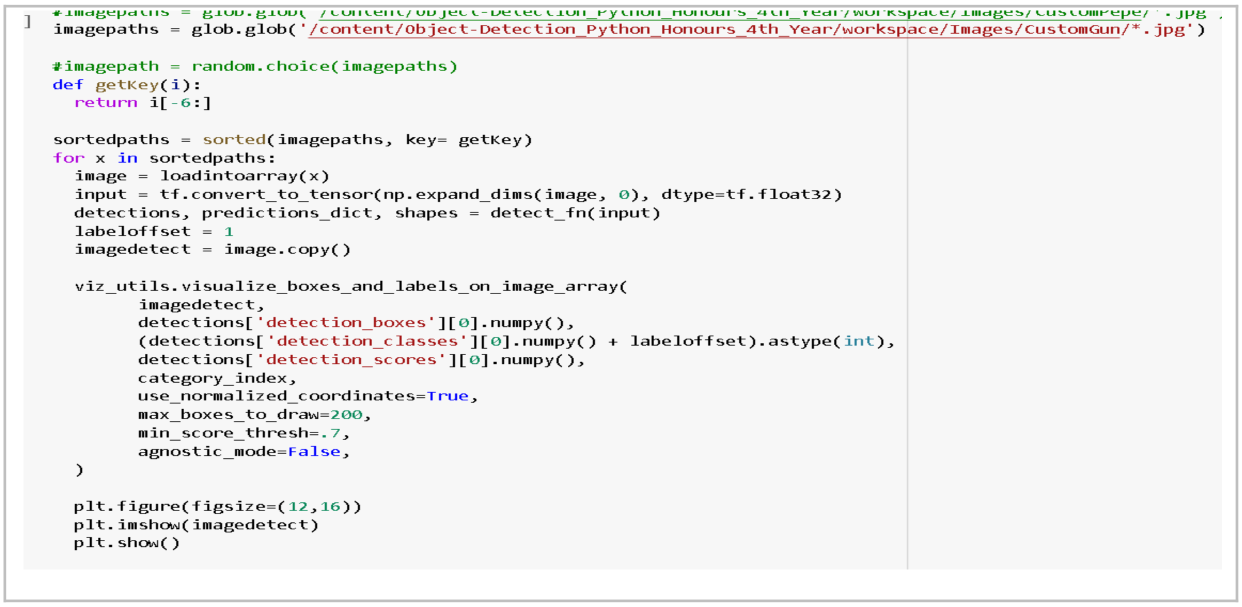


```
##Evalutation via tensorflow eval method
!python /content/models/research/object_detection/model_main_tf2.py \
    --pipeline_config_path={trainconfigFR} \
    --model_dir={trainmodelFR} \
    --checkpoint_dir={trainmodelFR}
```

Code 8 - TensorFlow API Evaluation Script

When conducting inference testing *Code 9* will be executed individually for each exported model. The code in *Code 9* will load images from the custom gun dataset into an array before individually converting each image into a tensor to feed into the object detecting model to run inference. As each inference is made the total time elapsed between each inference is recorded for each image into the test plan shown in Figure *4*.

Once predictions have been made, bounding boxes will be rendered onto a copy of the image with a class label and confidence threshold which will also be recorded in the test plan.



```
#imagepaths = glob.glob('/content/Object-Detection_Python_Honours_4th_Year/workspace/Images/CustomPepe/*.jpg')
imagepaths = glob.glob('/content/Object-Detection_Python_Honours_4th_Year/workspace/Images/CustomGun/*.jpg')

#imagepath = random.choice(imagepaths)
def getKey(i):
  return i[-6:]

sortedpaths = sorted(imagepaths, key= getKey)
for x in sortedpaths:
  image = loadintoarray(x)
  input = tf.convert_to_tensor(np.expand_dims(image, 0), dtype=tf.float32)
  detections, predictions_dict, shapes = detect_fn(input)
  labeloffset = 1
  imagedetect = image.copy()

  viz_utils.visualize_boxes_and_labels_on_image_array(
        imagedetect,
        detections['detection_boxes'][0].numpy(),
        (detections['detection_classes'][0].numpy() + labeloffset).astype(int),
        detections['detection_scores'][0].numpy(),
        category_index,
        use_normalized_coordinates=True,
        max_boxes_to_draw=200,
        min_score_thresh=.7,
        agnostic_mode=False,
  )

  plt.figure(figsize=(12,16))
  plt.imshow(imagedetect)
  plt.show()
```

Code 9 - Inference Code Section

### 3.7.2 Issues During Testing Phase

Issues identified during testing were identified during model evaluation and during inference testing. The errors identified during testing have a high potential of disrupting final testing and have lowered the overall quality of testing and evaluation results.

#### 3.7.2.1 Bad Training Due to a Lack of Machine Resources

Due to the nature of Google Colab machine resources are very limited and the model configuration properties set in *Code 4* are insufficient to train the three selected models (Faster R-CNN, EfficientDet, RetinaNet) effectively.

Attempts to resolve this were made by raising and lowering the batch size but that would only lead to training operations crashing from the lack of memory or still not enough batch size to train the models sufficiently. Further attempt to resolve this issue included increasing the number of steps during training but that only cause Google Colab to time out due to time constraints per session made.

#### 3.7.2.2 Unable to Evaluate RetinaNet

When attempting to execute TensorFlow's evaluation script on RetinaNet it would endlessly loop not completing the evaluation even though RetinaNet's configuration file is configured to stop the evaluation after 500 steps. Attempts were made to indirectly access the evaluation logs by downloading them from Google Colab and via TensorBoard but it was not possible to see the specific performance data needed for evaluation such as mAP, AR, etc.

# 4.0 Evaluation and Discussion

This section will discuss and evaluate the deliverables of this project from the literature review, implementation of the prototype and final testing to see if the research question of this project has been answered.

The research question defined in the introduction was: “What is the most efficient object detecting architecture that can be used to moderate content while requiring minimal training data and maintaining a relatively high degree of success?”

To answer such a research question, a set of metrics that must be considered for an object detection architecture to be concluded as the most efficient.

- Precision of the object detecting model.
- Speed of the object detecting model.
- Recall of the object detecting model.

As identified during the testing stage of execution, there were two severe errors during final testing that have greatly impacted the results of this project. Nevertheless, the impacted results still do show some promise as the search for the most efficient model can still be concluded from these results.

## *4.1 Model Analysis Evaluation*

Once the three object detecting models (Faster R-CNN, EfficentDet and RetinaNet) have been trained, model evaluation was conducted via the use of TensorFlow’s evaluation script which would output an array of performance metrics about that detection model as exemplified with the raw output in *Figure 6 and Figure 7.*

As visible from the filled evaluation table *Table 12* from final testing, RetinaNet has the largest total loss of 18.87 and a total train time of 1593s but as mentioned as an issue during the testing of RetinaNet it was not possible to evaluate RetinaNet via the evaluation script or via TensorFlow’s TensorBoard extension.

Faster R-CNN from *Table 12* has the second largest total loss at 0.64 and a total train time of 1605s but the object detection model has an average recall (AR@100) of 0.38 per one hundred detections and a mean average precision (mAP) of 0.35%.

Lastly from *Table 12* EfficientDet can be seen boasting the smallest total loss of 0.4 and a total train time of 967s but the EfficientDet model has an average recall (AR@100) of 0.33 per one hundred detections with a mean average precision (mAP) of 0.31%.

### 4.1.2 Conclusion

From the statistics gathered after each object detecting model being trained it still can be concluded that the highest performing model out of the three is the Faster R-CNN model due to the model having the greatest mean average precision of 0.35% and the greatest average recall values when compared to EfficientDet.

## *4.2 Inference Testing Evaluation*

During final inference testing, it has become apparent the results have been impacted by the errors identified during testing as there is a far smaller number of detections made by the detection model than initially desired.

As evident in *Table 7,* the Faster R-CNN model has a total inference time of 119s with 21 successful detections and 1 negative detection on image 9 in *Table 7*. The highest confidence percentage made by Faster R-CNN was 97% and the average percentage of each confidence reading was 80.45%.

As evident in *Table 8,* EfficientDet has a total inference time of 146s with 6 successful detections and 1 negative detection on image 11 in *Table 8.* The highest confidence percentage made by EfficientDet was 79% and the average percentage of each confidence reading was 75.57%.

As evident in *Table 9,* RetinaNet has a total inference time of 110s with only 2 successful detections the highest confidence percentage being 86% and the average percentage of each confidence reading was 80.5%.

### 4.2.2 Conclusion

From analysis of the three object detecting models during inference testing it can be concluded that Faster R-CNN performed the best as the Faster R-CNN model has the greatest number of detections of 21 with the highest percentage of confidence at 97% even though the average confidence is similar to RetinaNet with 80.45%.

# 5.0 Conclusion and Further Work

This section will review the conclusions made during evaluation section and the end conclusion of the literature review section to come to a definitive answer to this project's research question.

This section will first provide a project resume summarizing the problem to be solved by this project before moving to discuss this project's conclusion along with any limitations that have impacted the overall conclusion and future work that can be conducted for this project.

## *5.1 Project Resume*

With the ever-increasing demand and userbase on online social media, online forums, online trading and content creation sites the need for big and small companies to automate the process of content moderation greatly increase as the workload becomes too great for just human moderators to handle.

The research question for this project is as follow: "What is the most efficient object detecting architecture that can be used to moderate content while requiring minimal training data and maintaining a relatively high degree of success?"

In order to answer this research question, a literature review was conducted to investigate technologies that can be used to develop, train and test a selection of object detecting architectures from the literature review which were Faster R-CNN, EfficientDet and RetinaNet. Before moving on to develop a custom dataset and a notebook application making use of TensorFlow and Google Colab to train and test the previously mentioned object detection models.

After encountering severe issues during testing, all test result and data gathered was evaluated to conclude that the Faster R-CNN object detecting architecture was the most efficient out of the three chosen for this project.

## *5.2 Conclusion*

After conducting research on the three object-detecting architectures (Faster R-CNN, EfficientDet and RetinaNet) via the literature review followed by the results yielded from the final prototype implemented during the execution stage, the research question can finally be definitively answered.

Overall, the project's research question has successfully been answered by producing a prototype notebook that can be executed on cloud services such as Google Colab albeit not in the manner as initially desired from the start of this project's execution.

The conclusion made in the literature review on section 2.8 of this report shows the Faster R-CNN architecture outperforming EfficientDet and RetinaNet by having the greatest mean average precision of 78% from *Table 1* and the second faster inference time of 59ms also from *Table 1.*

In the inference testing evaluation on section 4.2, it was concluded the Faster R-CNN model performed the best during inference testing due to the object detecting model having the greatest number of successful detections and the highest percentage of confidence at 97% at an average confidence of 80.45%.

In the model evaluations made on section 4.1 it was concluded the Faster R-CNN model had the best metrics by having the greatest mean average precision of 0.35% and the greatest average recall (AR@100) of 0.38 per one hundred detections.

With all factors from the project's execution considered it can be concluded that the Faster R-CNN architecture is the most efficient object detecting architecture for use in content moderation.

### 5.2.2 Limitations

Even though the research question has been satisfied by the conclusions drawn from the literature review and evaluation sections, there are still a few limitations that have made the execution stage of this project a great deal more difficult than it should have been.

Due to Google Colab's restrictions on machine resources made available on Google Colab (memory, CPU, GPU and TPU's) served as a first come, first serve basis with a limited time for use it spontaneously reduces the overall quality of code execution and thus making the overall quality of training and testing of the object detection models less reliable on Google Colab. Additional machine resource would also mean being able to use scaled up versions of the same object detecting architectures which have a greater mean average precision as demonstrated in the architecture investigation throughout section 2.3 of the literature review.

### 5.2.3 Future work

Even though the errors encountered during final inference testing and model evaluation in the execution stage have greatly hampered the development of a high-quality set of test results, the literature review and prototype still prove that the Faster R-CNN model is the most efficient object detection model out of the three chosen to answer this project's research question.

The prototype will benefit greatly from being run on a more reliable cloud service and or a locally owned device with the necessary GPU and memory to execute the Python notebook with a better set of training properties than what is displayed in *Code 4.* If the batch size and number of steps were increased without Google Colab's memory and time restrictions the models will most certainly have trained and inference tested more effectively.

## 6.0 References

Aidouni, M. E., 2019. *Evaluating Object Detection Models: Guide to Performance Metrics.* [Online]
Available at:
manalelaidouni.github.io/Evaluating-Object-Detection-Models-Guide-to-Performance-Metrics.html
[Accessed 14 04 2021].

Ananth, S., 2019. *Faster R-CNN for object detection.* [Online]
Available at:
https://towardsdatascience.com/faster-r-cnn-for-object-detection-a-technical-summary-474c5b857b46
[Accessed 18 11 2020].

ArcGis Developers, n.d. *How RetinaNet works?.* [Online]
Available at:
developers.arcgis.com/python/guide/how-retinanet-works/#:~:text=RetinaNet%20is%20one%20of%20the,dense%20and%20small%20scale%20objects.&text=RetinaNet%20has%20been%20formed%20by,and%20Focal%20Loss%20%5B2%5D.
[Accessed 5 04 2021].

Babich, N., 2020. *What is Computer Vision & How Does it Work? An Introduction.* [Online]
Available at:
https://xd.adobe.com/ideas/principles/emerging-technology/what-is-computer-vision-how-does-it-work/
[Accessed 17 11 2020].

Bhattacharyya, J., 2020. *Step by Step Guide To Object Detection Using Roboflow.* [Online]
Available at: analyticsindiamag.com/step-by-step-guide-to-object-detection-using-roboflow/
[Accessed 5 4 2020].

DaSCI, 2020. *Weapon Detection.* [Online]
Available at: dasci.es/transferencia/open-data/24705/
[Accessed 8 4 2021].

Feng, V., 2017. *An Overview of ResNet and its Variants.* [Online]
Available at: towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035
[Accessed 5 04 2021].

Google, n.d. *Colaboratory Frequently Asked Questions.* [Online]
Available at:
research.google.com/colaboratory/faq.html#:~:text=Colaboratory%2C%20or%20%E2%80%9CColab%E2%80%9D%20for,learning%2C%20data%20analysis%20and%20education.
[Accessed 5 04 2021].

IBM Cloud Education, 2020. *Machine-Learning.* [Online]
Available at: https://www.ibm.com/cloud/learn/machine-learning
[Accessed 17 11 2020].

KHAZRI, A., 2019. *Faster RCNN Object detection.* [Online]
Available at:
towardsdatascience.com/faster-rcnn-object-detection-f865e5ed7fc4#:~:text=Faster%20RCNN%20is%

20an%20object,SSD%20(%20Single%20Shot%20Detector).
[Accessed 5 04 2021].

Lin, T.-Y.et al., 2016. *Feature Pyramid Networks for Object Detection.* [Online]
Available at: arxiv.org/abs/1612.03144
[Accessed 05 04 2021].

Lin, T.-Y.et al., 2018. *Focal Loss for Dense Object Detection.* [Online]
Available at: arxiv.org/pdf/1708.02002.pdf
[Accessed 4 04 2021].

MADALI, N., 2020. *Demystifying Region Proposal Network (RPN).* [Online]
Available at: medium.com/@nabil.madali/demystifying-region-proposal-network-rpn-faa5a8fb8fce
[Accessed 2 4 2021].

Messenger, M., 2018. *What exactly is TensorFlow.* [Online]
Available at: medium.com/datadriveninvestor/what-exactly-is-tensorflow-80a90162d5f1
[Accessed 21 11 2020].

Olmos, R., Tabik, S. & Herrera, F., 2017. Automatic handgun detection alarm in videos using deep learning. *Neurocomputing,* Volume 275, pp. 66-72.

Rathod, V. & Huang, J., 2020. *TensorFlow 2 meets the Object Detection API.* [Online]
Available at: blog.tensorflow.org/2020/07/tensorflow-2-meets-object-detection-api.html
[Accessed 2 4 2021].

Reddy, S., 2019. *Waterfall Methodology in Project Management — Phases, Benefits.* [Online]
Available at:
medium.com/@sudarhtc/waterfall-methodology-in-project-management-phases-benefits-85393be2f1d
[Accessed 12 04 2021].

Ren, S., He, K., Girshick, R. & Sun, J., 2016. *Faster R-CNN: Towards Real-Time Object Detection.* [Online]
Available at: arxiv.org/pdf/1506.01497.pdf
[Accessed 18 11 2020].

RoboFlow, 2020. *Pistols Dataset.* s.l.:University of Grenada.

Schroepfer, M., 2020. *How AI is getting better at detecting hate speech.* [Online]
Available at: ai.facebook.com/blog/how-ai-is-getting-better-at-detecting-hate-speech
[Accessed 02 04 2021].

Sethi, A., 2020. *Build your own object detection model using TensorFlow API.* [Online]
Available at:
analyticsvidhya.com/blog/2020/04/build-your-own-object-detection-model-using-tensorflow-api/
[Accessed 21 11 2020].

Solawetz, J., 2020. *A Thorough Breakdown of EfficientDet for Object Detection.* [Online]
Available at:
https://towardsdatascience.com/a-thorough-breakdown-of-efficientdet-for-object-detection-dc6a15788b73
[Accessed 18 11 2020].

Stewart, M., 2019. *Simple Introduction to Convolutional Neural Networks.* [Online]
Available at:
towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac
[Accessed 10 4 2021].

Tan, M., Pang, R. & Le, Q. V., 2020. *EfficientDet: Scalable and Efficient Object Detection.* [Online]
Available at: https://arxiv.org/pdf/1911.09070v7.pdf
[Accessed 18 11 2020].

Triguero, F. H. et al., n.d. *Weapons detection for security and video surveillance.* [Online]
Available at: https://sci2s.ugr.es/weapons-detection#RP
[Accessed 2 4 2021].

Tsang, S.-H., 2019. *Review: RetinaNet — Focal Loss (Object Detection).* [Online]
Available at: towardsdatascience.com/review-retinanet-focal-loss-object-detection-38fba6afabe4
[Accessed 22 03 2021].

Yao, S., 2018. *Categorizing Listing Photos at Airbnb.* [Online]
Available at:
https://medium.com/airbnb-engineering/categorizing-listing-photos-at-airbnb-f9483f3ab7e3
[Accessed 20 11 2020].

Youtube, 2020. *Using Content ID.* [Online]
Available at: https://support.google.com/youtube/answer/3244015?hl=en-GB&ref_topic=4515467
[Accessed 17 11 2020].

Yu, A. & Tan, M., 2020. *EfficientDet: Towards Scalable and Efficient Object Detection.* [Online]
Available at: ai.googleblog.com/2020/04/efficientdet-towards-scalable-and.html
[Accessed 5 04 2021].

# 7.0 Appendices

### Full GitHub Code Listing Source:

github.com/DeRuss404/Object-Detection_Python_Honours_4th_Year

### Pistol Dataset Source from RoboFlow:

public.roboflow.com/object-detection/pistols

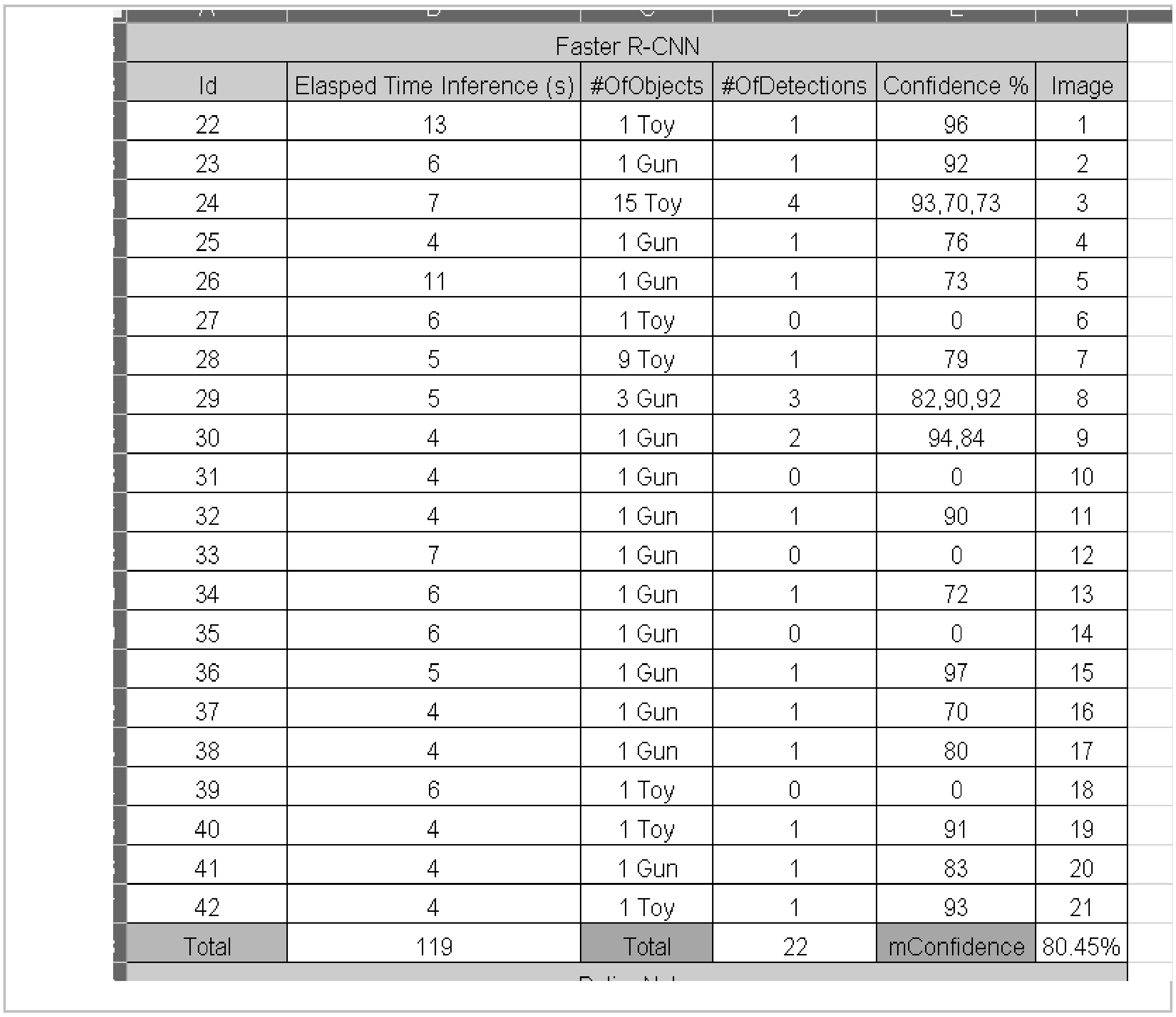
### Raw Handgun Dataset Source from DaSCI:

github.com/ari-dasci/OD-WeaponDetection/tree/master/Pistol%20detection

### Embedded Testplan, Gantt Chart:

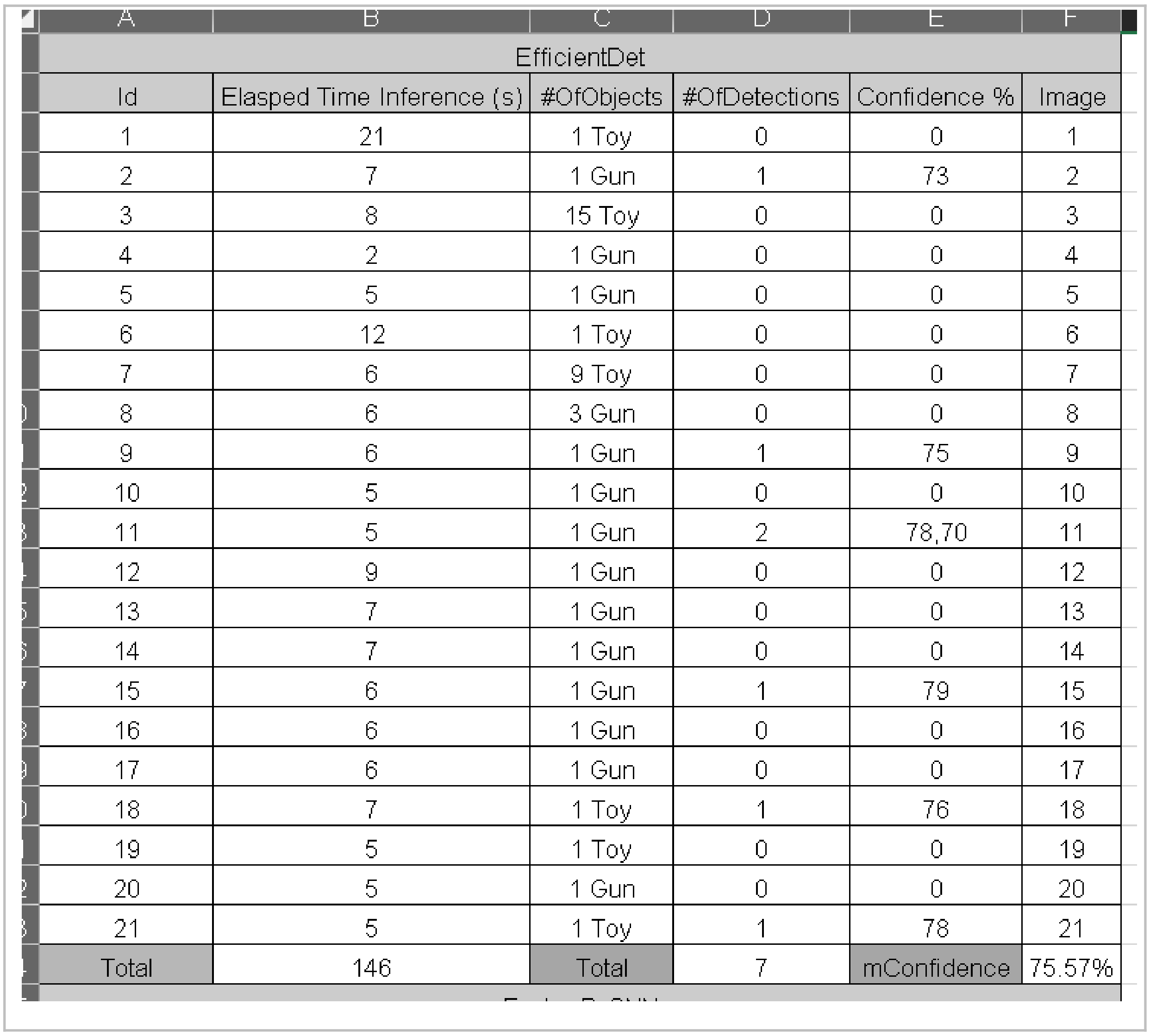
HonoursTestPlanFilled.xlsx

Honours Gantt Chart.pdf



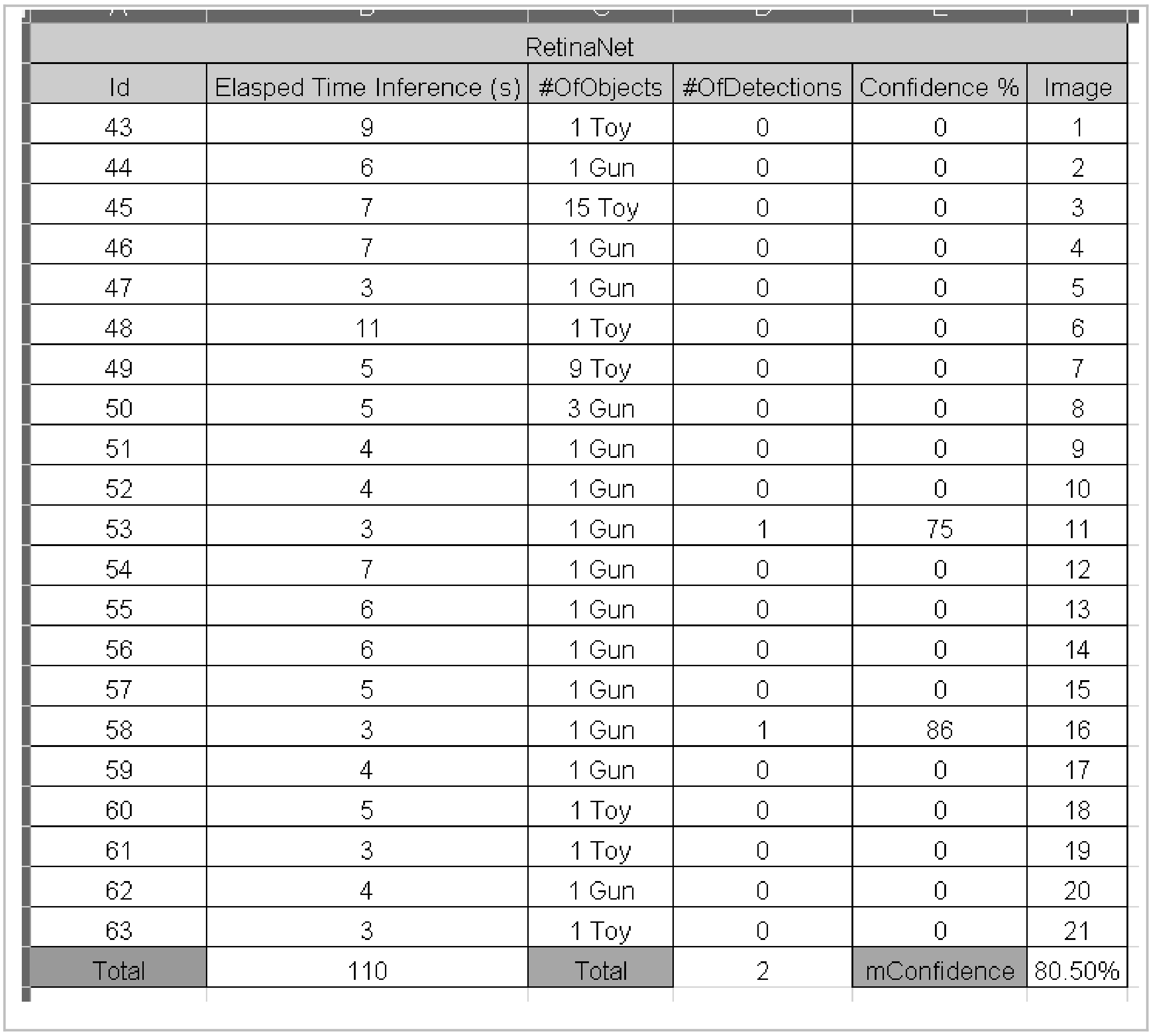
| Faster R-CNN | | | | | |
|---|---|---|---|---|---|
| Id | Elasped Time Inference (s) | #OfObjects | #OfDetections | Confidence % | Image |
| 22 | 13 | 1 Toy | 1 | 96 | 1 |
| 23 | 6 | 1 Gun | 1 | 92 | 2 |
| 24 | 7 | 15 Toy | 4 | 93,70,73 | 3 |
| 25 | 4 | 1 Gun | 1 | 76 | 4 |
| 26 | 11 | 1 Gun | 1 | 73 | 5 |
| 27 | 6 | 1 Toy | 0 | 0 | 6 |
| 28 | 5 | 9 Toy | 1 | 79 | 7 |
| 29 | 5 | 3 Gun | 3 | 82,90,92 | 8 |
| 30 | 4 | 1 Gun | 2 | 94,84 | 9 |
| 31 | 4 | 1 Gun | 0 | 0 | 10 |
| 32 | 4 | 1 Gun | 1 | 90 | 11 |
| 33 | 7 | 1 Gun | 0 | 0 | 12 |
| 34 | 6 | 1 Gun | 1 | 72 | 13 |
| 35 | 6 | 1 Gun | 0 | 0 | 14 |
| 36 | 5 | 1 Gun | 1 | 97 | 15 |
| 37 | 4 | 1 Gun | 1 | 70 | 16 |
| 38 | 4 | 1 Gun | 1 | 80 | 17 |
| 39 | 6 | 1 Toy | 0 | 0 | 18 |
| 40 | 4 | 1 Toy | 1 | 91 | 19 |
| 41 | 4 | 1 Gun | 1 | 83 | 20 |
| 42 | 4 | 1 Toy | 1 | 93 | 21 |
| Total | 119 | Total | 22 | mConfidence | 80.45% |

Table 7- Inference Test Table Faster R-CNN



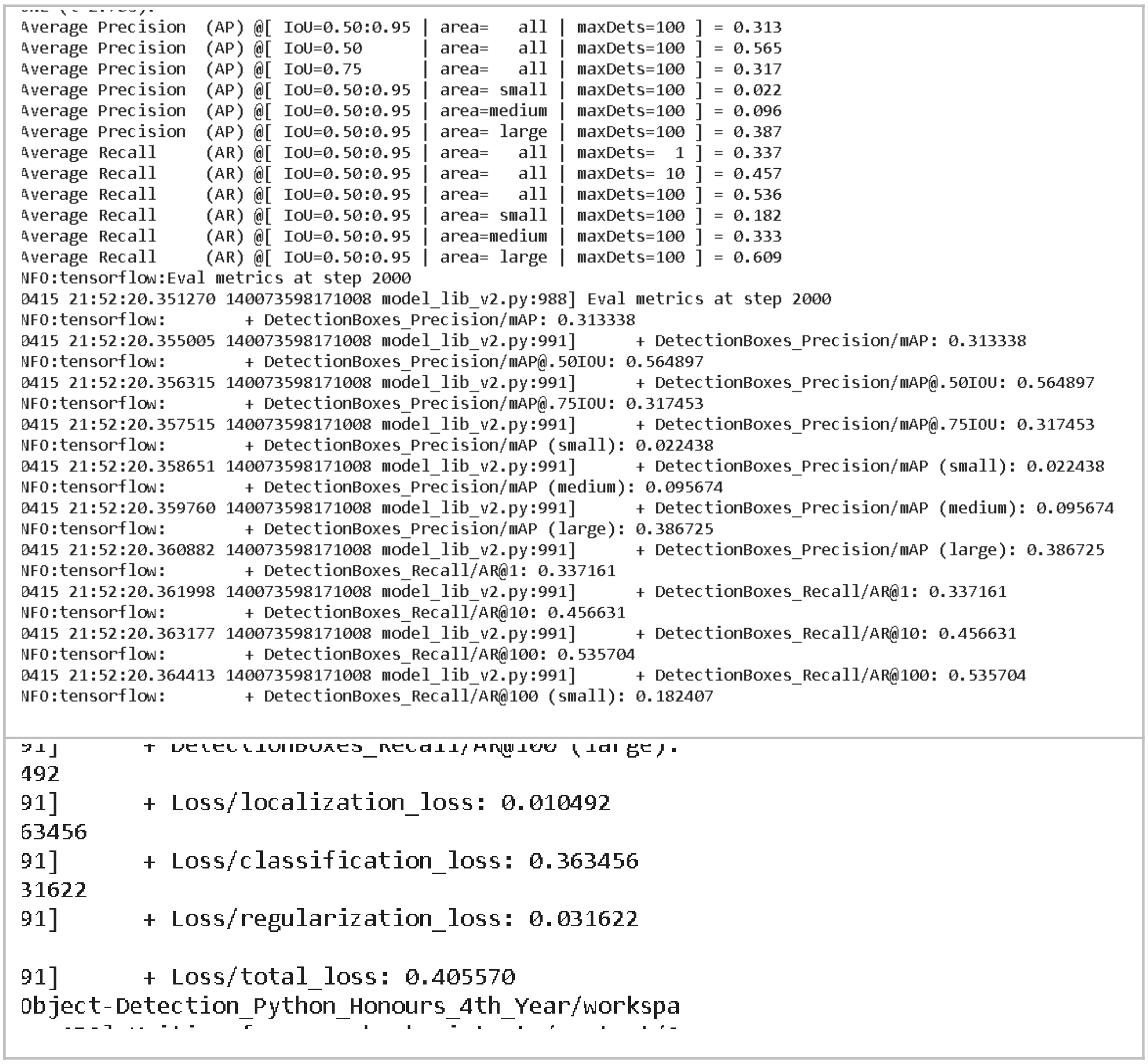
| A | B | C | D | E | F |
|---|---|---|---|---|---|
| EfficientDet | | | | | |
| Id | Elasped Time Inference (s) | #OfObjects | #OfDetections | Confidence % | Image |
| 1 | 21 | 1 Toy | 0 | 0 | 1 |
| 2 | 7 | 1 Gun | 1 | 73 | 2 |
| 3 | 8 | 15 Toy | 0 | 0 | 3 |
| 4 | 2 | 1 Gun | 0 | 0 | 4 |
| 5 | 5 | 1 Gun | 0 | 0 | 5 |
| 6 | 12 | 1 Toy | 0 | 0 | 6 |
| 7 | 6 | 9 Toy | 0 | 0 | 7 |
| 8 | 6 | 3 Gun | 0 | 0 | 8 |
| 9 | 6 | 1 Gun | 1 | 75 | 9 |
| 10 | 5 | 1 Gun | 0 | 0 | 10 |
| 11 | 5 | 1 Gun | 2 | 78,70 | 11 |
| 12 | 9 | 1 Gun | 0 | 0 | 12 |
| 13 | 7 | 1 Gun | 0 | 0 | 13 |
| 14 | 7 | 1 Gun | 0 | 0 | 14 |
| 15 | 6 | 1 Gun | 1 | 79 | 15 |
| 16 | 6 | 1 Gun | 0 | 0 | 16 |
| 17 | 6 | 1 Gun | 0 | 0 | 17 |
| 18 | 7 | 1 Toy | 1 | 76 | 18 |
| 19 | 5 | 1 Toy | 0 | 0 | 19 |
| 20 | 5 | 1 Gun | 0 | 0 | 20 |
| 21 | 5 | 1 Toy | 1 | 78 | 21 |
| Total | 146 | Total | 7 | mConfidence | 75.57% |

Table 8 – Inference Test Table EfficientDet



| RetinaNet | | | | | |
|---|---|---|---|---|---|
| Id | Elasped Time Inference (s) | #OfObjects | #OfDetections | Confidence % | Image |
| 43 | 9 | 1 Toy | 0 | 0 | 1 |
| 44 | 6 | 1 Gun | 0 | 0 | 2 |
| 45 | 7 | 15 Toy | 0 | 0 | 3 |
| 46 | 7 | 1 Gun | 0 | 0 | 4 |
| 47 | 3 | 1 Gun | 0 | 0 | 5 |
| 48 | 11 | 1 Toy | 0 | 0 | 6 |
| 49 | 5 | 9 Toy | 0 | 0 | 7 |
| 50 | 5 | 3 Gun | 0 | 0 | 8 |
| 51 | 4 | 1 Gun | 0 | 0 | 9 |
| 52 | 4 | 1 Gun | 0 | 0 | 10 |
| 53 | 3 | 1 Gun | 1 | 75 | 11 |
| 54 | 7 | 1 Gun | 0 | 0 | 12 |
| 55 | 6 | 1 Gun | 0 | 0 | 13 |
| 56 | 6 | 1 Gun | 0 | 0 | 14 |
| 57 | 5 | 1 Gun | 0 | 0 | 15 |
| 58 | 3 | 1 Gun | 1 | 86 | 16 |
| 59 | 4 | 1 Gun | 0 | 0 | 17 |
| 60 | 5 | 1 Toy | 0 | 0 | 18 |
| 61 | 3 | 1 Toy | 0 | 0 | 19 |
| 62 | 4 | 1 Gun | 0 | 0 | 20 |
| 63 | 3 | 1 Toy | 0 | 0 | 21 |
| Total | 110 | Total | 2 | mConfidence | 80.50% |

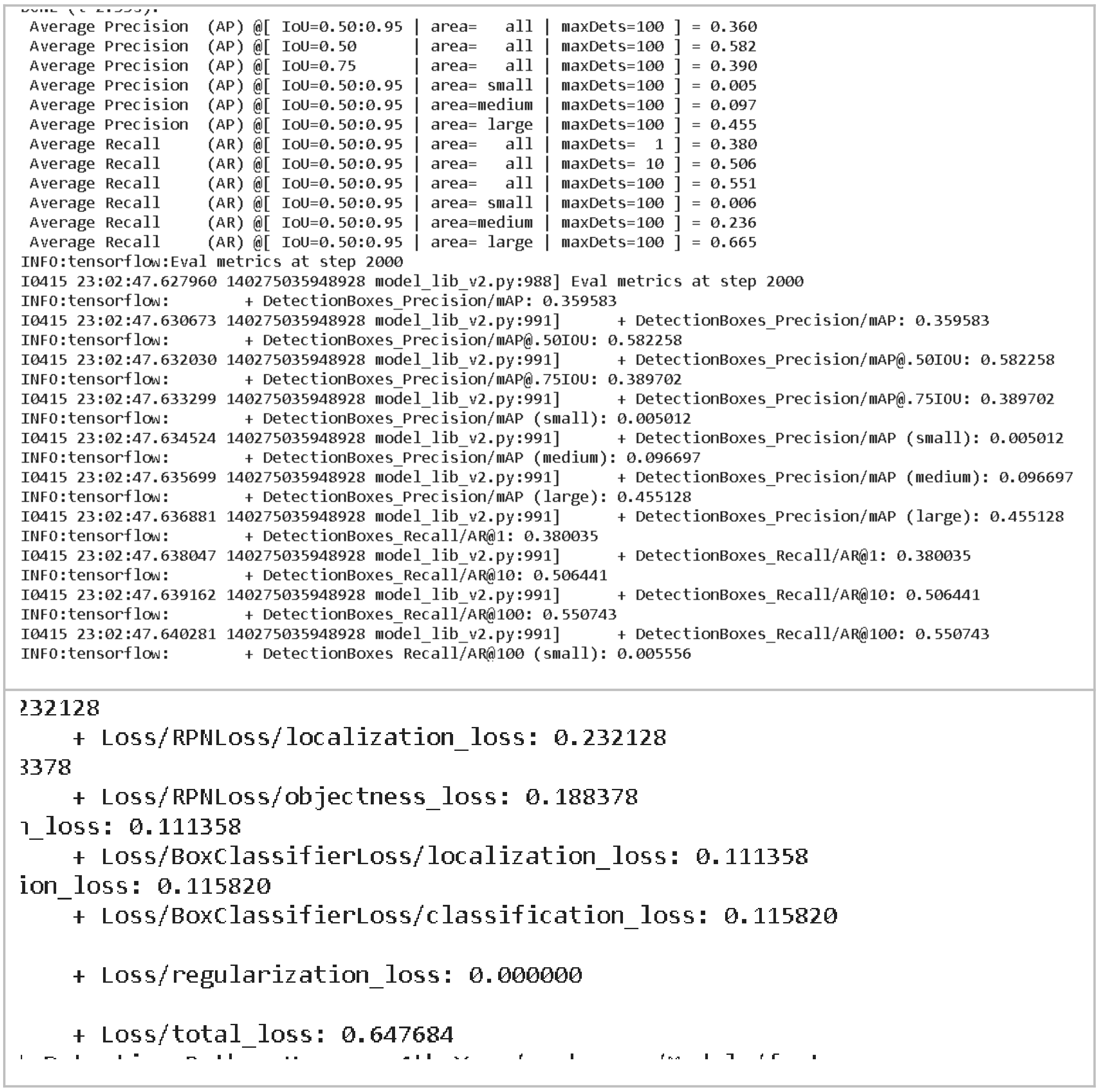
Table 9 – Inference Test Table RetinaNet



```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.313
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.565
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.317
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.022
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.096
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.387
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.337
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.457
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.536
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.182
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.333
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.609
NFO:tensorflow:Eval metrics at step 2000
0415 21:52:20.351270 140073598171008 model_lib_v2.py:988] Eval metrics at step 2000
NFO:tensorflow:        + DetectionBoxes_Precision/mAP: 0.313338
0415 21:52:20.355005 140073598171008 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP: 0.313338
NFO:tensorflow:        + DetectionBoxes_Precision/mAP@.50IOU: 0.564897
0415 21:52:20.356315 140073598171008 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP@.50IOU: 0.564897
NFO:tensorflow:        + DetectionBoxes_Precision/mAP@.75IOU: 0.317453
0415 21:52:20.357515 140073598171008 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP@.75IOU: 0.317453
NFO:tensorflow:        + DetectionBoxes_Precision/mAP (small): 0.022438
0415 21:52:20.358651 140073598171008 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP (small): 0.022438
NFO:tensorflow:        + DetectionBoxes_Precision/mAP (medium): 0.095674
0415 21:52:20.359760 140073598171008 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP (medium): 0.095674
NFO:tensorflow:        + DetectionBoxes_Precision/mAP (large): 0.386725
0415 21:52:20.360882 140073598171008 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP (large): 0.386725
NFO:tensorflow:        + DetectionBoxes_Recall/AR@1: 0.337161
0415 21:52:20.361998 140073598171008 model_lib_v2.py:991]      + DetectionBoxes_Recall/AR@1: 0.337161
NFO:tensorflow:        + DetectionBoxes_Recall/AR@10: 0.456631
0415 21:52:20.363177 140073598171008 model_lib_v2.py:991]      + DetectionBoxes_Recall/AR@10: 0.456631
NFO:tensorflow:        + DetectionBoxes_Recall/AR@100: 0.535704
0415 21:52:20.364413 140073598171008 model_lib_v2.py:991]      + DetectionBoxes_Recall/AR@100: 0.535704
NFO:tensorflow:        + DetectionBoxes_Recall/AR@100 (small): 0.182407
```

```
91]       + DetectionBoxes_Recall/AR@100 (large):
492
91]       + Loss/localization_loss: 0.010492
63456
91]       + Loss/classification_loss: 0.363456
31622
91]       + Loss/regularization_loss: 0.031622

91]       + Loss/total_loss: 0.405570
Object-Detection_Python_Honours_4th_Year/workspa
```

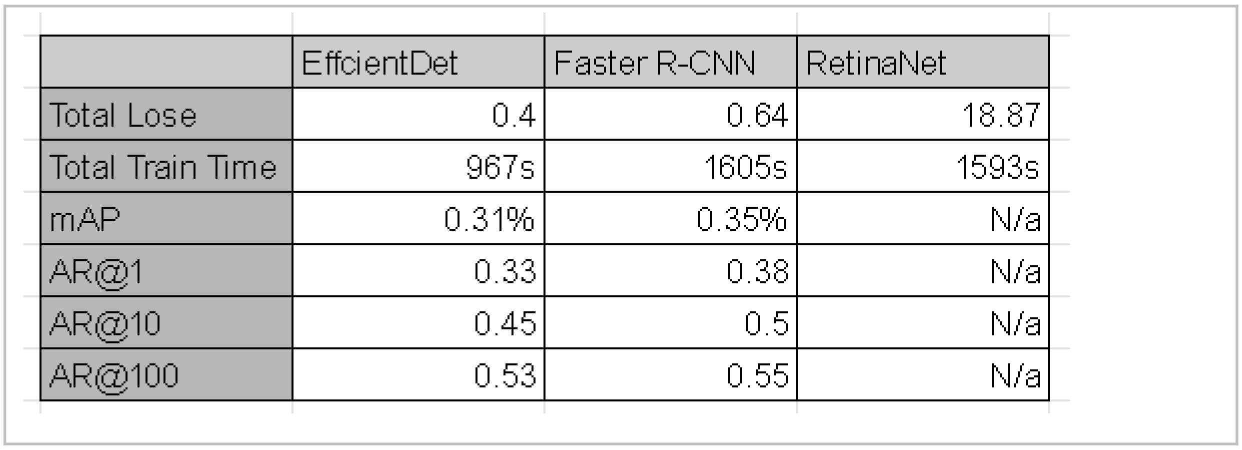
Figure 6 – Raw EfficientDet Evaluation Log



```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.360
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.582
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.390
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.005
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.097
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.455
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.380
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.506
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.551
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.006
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.236
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.665
INFO:tensorflow:Eval metrics at step 2000
I0415 23:02:47.627960 140275035948928 model_lib_v2.py:988] Eval metrics at step 2000
INFO:tensorflow:        + DetectionBoxes_Precision/mAP: 0.359583
I0415 23:02:47.630673 140275035948928 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP: 0.359583
INFO:tensorflow:        + DetectionBoxes_Precision/mAP@.50IOU: 0.582258
I0415 23:02:47.632030 140275035948928 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP@.50IOU: 0.582258
INFO:tensorflow:        + DetectionBoxes_Precision/mAP@.75IOU: 0.389702
I0415 23:02:47.633299 140275035948928 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP@.75IOU: 0.389702
INFO:tensorflow:        + DetectionBoxes_Precision/mAP (small): 0.005012
I0415 23:02:47.634524 140275035948928 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP (small): 0.005012
INFO:tensorflow:        + DetectionBoxes_Precision/mAP (medium): 0.096697
I0415 23:02:47.635699 140275035948928 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP (medium): 0.096697
INFO:tensorflow:        + DetectionBoxes_Precision/mAP (large): 0.455128
I0415 23:02:47.636881 140275035948928 model_lib_v2.py:991]      + DetectionBoxes_Precision/mAP (large): 0.455128
INFO:tensorflow:        + DetectionBoxes_Recall/AR@1: 0.380035
I0415 23:02:47.638047 140275035948928 model_lib_v2.py:991]      + DetectionBoxes_Recall/AR@1: 0.380035
INFO:tensorflow:        + DetectionBoxes_Recall/AR@10: 0.506441
I0415 23:02:47.639162 140275035948928 model_lib_v2.py:991]      + DetectionBoxes_Recall/AR@10: 0.506441
INFO:tensorflow:        + DetectionBoxes_Recall/AR@100: 0.550743
I0415 23:02:47.640281 140275035948928 model_lib_v2.py:991]      + DetectionBoxes_Recall/AR@100: 0.550743
INFO:tensorflow:        + DetectionBoxes_Recall/AR@100 (small): 0.005556
```

```
232128
    + Loss/RPNLoss/localization_loss: 0.232128
3378
    + Loss/RPNLoss/objectness_loss: 0.188378
n_loss: 0.111358
    + Loss/BoxClassifierLoss/localization_loss: 0.111358
ion_loss: 0.115820
    + Loss/BoxClassifierLoss/classification_loss: 0.115820

    + Loss/regularization_loss: 0.000000

    + Loss/total_loss: 0.647684
```

Figure 7- Raw Faster R-CNN Evaluation Log



| | EffcientDet | Faster R-CNN | RetinaNet |
|---|---|---|---|
| Total Lose | 0.4 | 0.64 | 18.87 |
| Total Train Time | 967s | 1605s | 1593s |
| mAP | 0.31% | 0.35% | N/a |
| AR@1 | 0.33 | 0.38 | N/a |
| AR@10 | 0.45 | 0.5 | N/a |
| AR@100 | 0.53 | 0.55 | N/a |

Table 12- Evaluation Table