**Introduction**

Self-driving cars or autonomous cars have been receiving tremendous attention due to Deep Neural Network (DNN) that automate a lot of manual tasks of driving the car. Autonomous cars are vehicles that are capable of sensing their environment and moving safely with little or no human input. The vehicles can interpret sensory information to identify appropriate navigation paths, as well as obstacles and relevant signage. The main motivation is to minimise road accident due to human error or fatigue while driving.

Diagram

Description automatically generatedThere a six different level of autonomous.

• Level 0: All functionality and systems of the car are controlled by humans. No automation.

• Level 1: Minor things like cruise control, automatic braking, or detecting something in the blind spot may be controlled by the computer, one at a time

• Level 2: A human is still required for safe operation and emergency procedures. The computer can perform some simultaneous automated functions, such as acceleration and steering.

• Level 3: The computer can control all critical operations of the car simultaneously including accelerating, steering, stopping, navigation and parking under most conditions. A human driver is still expected to be present in case they are alerted of an emergency.

• Level 4: The car is fully autonomous, without any need for a human driver, in some driving scenarios (i.e snowing).

• Level 5: The car is completely fully autonomous and capable of self-driving in every situation



In this project, the car will be simulated using Jetson Nano Bots and the road that cars are driven on are simulated with a black single lane as the marker that the car would track on. The car will be self-driven on the road based on the video stream captured by the front-facing camera on the car.

Formulation and Framing of ML Problem

**Problem Articulation**:

We will simulate a self-driving car and develop a deep learning model to detect the lane and map the position of the lane to direction to move (left or right) to keep the car on track of the road.

The objectives of this project are:

1. Develop a NN-based model that track the lane line.
2. Develop steering prediction algorithm to keep the car on the road.
3. Develop a NN-based acceleration/deceleration of the car. The car would accelerate to max road speed if the road is straight and de-acceleration when it is turning at the corner (left/right corners).
4. Develop a NN-based traffic light and road sign detection and change the speed of the car to obey the traffic rules.

The problem is best framed as a both a object detection task which predicts lane directions and traffic signs/lights classes as well as the location and dimensions of the objects indicated by bounding boxes.

**Starting with Simple Model:**

We will be starting with “Tensorflow Object Detection model (SSD-Resnet50)” which comprises of Single-Shot MultiBox Detector (SSD) for object detection and ResNet 50 for classification in the bounding box. This model serves as baseline when experimenting with more complex models.

Method

First, the image dataset for self-driving car is generated with the front camera on Jetson Nano Bot. These images can be classified by five classes of objects. They are: Left, Right, Straight, Traffic Light (Red, Yellow and Green) and Traffic sign (Stop etc). Below show images collected for different classes:



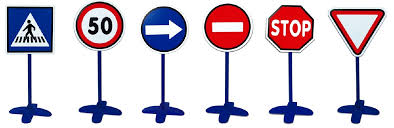
Left

Straight

Right

Self-driving image Dataset

Traffic sign



Traffic light

The image dataset is annotated and used to train on the Tensorflow Object Detection model (SSD-Resnet50) to detect the lane, traffic light and traffic sign. We will be using Single-Shot MultiBox Detector (SSD) for object detection and ResNet 50 for classification in the bounding box. The figure below shows the block diagram on how the self-driving car perform lane detection and steering of the wheels.

Vision

Object Detection

Predict direction & Speed

Steering command

Real-time cam

TFOD Model

Left, Right, straight, red, yellow, green light, traffic sign

Image

Turn left, Turn Right, Straight, Accelerate, Decelerate

Radian Accelerate Decelerate

Prediction

Steering

Self-driving Deep Learning Model

The trained model is deployed to the Jetson Nano Bot and the car is placed on a lane to begin driving around the track. The images captured by the front camera are resize to 224 x224 pixel before sending to the SSD-ResNet50 model for inferencing. It will detect objects like straight lane, left corner, right corner, traffic light and it’s state (red, yellow or green), and traffic sign such as stop sign. The model will give an output with 5 parameters for each object detected: Class, top, bottom, left and right pixel position.

A prediction module will analyse the objects that were being detected and return command for driving direction such as “Turn Left 30 degree” or “Drive Straight”. When the lane is straight, the module can determine if it is able to accelerate the car. If a corner is detected far away, it will maintain the speed of the car. When the corner is near, it will return command “Apply Brake” to slow down the car to turn safely around the bend/corner.

If traffic light is detected ahead, it will interpret the state of the traffic light such as red, yellow or green and return command such as “applied brake”. If traffic sign such as stop sign is detected, it will infer the distance from the car and return command such as “applied brake” when the sign is near so that the car will stop just before the stop sign.

The commands from the prediction module are sent to the steering module which will convert the command to low-level steering wheel signal. If command such as “Accelerate” or “Apply brake” is received, the steering module will sent accelerate or brake signal to the Jetson Nano Bot.

Intended Experiments

Icon

Description automatically generated

To test the deep learning module for the self-driving car, a test track on the left is created for testing purpose. The car will be tested by running on this track. We will monitor whether the car is able to navigate through the course.

We may also experiment with other models in the TensorFlow 1 Detection Model Zoo such as “SSD Inception” or “SSD mobilenet” as well as more accurate models in TensorFlow 2 Detection Model Zoo such as EfficientDet and CenterNet (which are not SSD based) and family of R-CNN nets, if time permits.

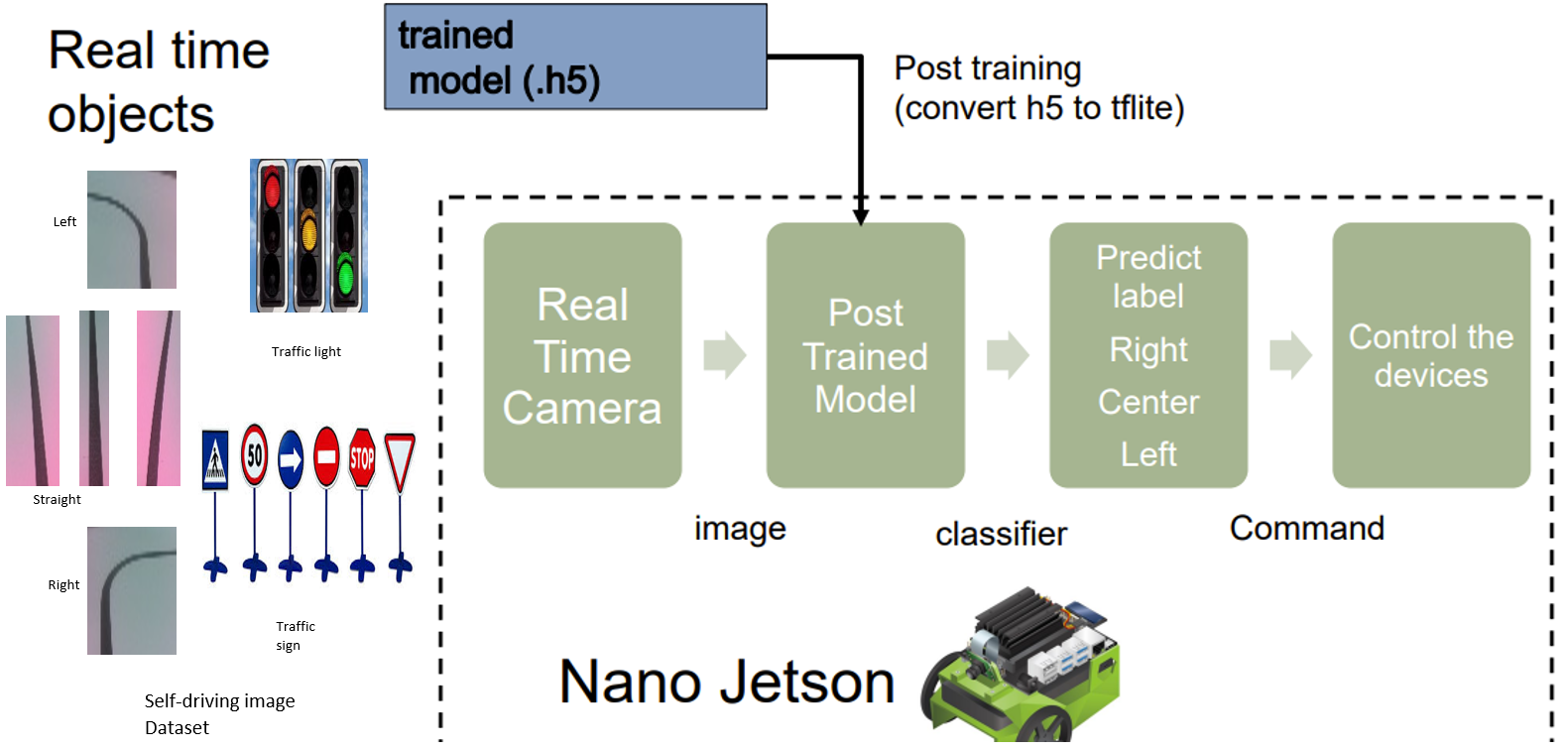
Performance Tuning and Optimization

Below are some of the things we can do to improve performance/accuracy of the model:

* Fine-tune the model by performing a transfer learning with pre-trained model from TFOD model zoo.
* Remove the last fully-connected layer and change the number of classes detected to the number of objects acquired in the self-driving dataset.
* Set all the classification layers in SSD to be trainable.
* Set some of top Convolution layers to be trainable since our objects are very different from the images acquired from pre-trained CoCo dataset.
* Set more top Convolution layers to be trainable to further improve on the accuracy if necessary.
* Add batch normalisation layer after each convolution layers.

Application-based Integration

During post-training quantization, TensorFlow Lite Converter will be used to convert the trained model in Tensorflow (saved model) or Keras (h5) formats to TensorFlow Lite format, thereby reducing the model size while also improving CPU and hardware accelerator latency, with little degradation in model accuracy.



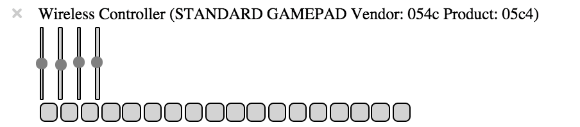
**Data Collection**

In this project, we will be doing data collecting using the Jetson Nano Bot to generate the dataset with game controller as shown in the figure on the right. The live camera feed from the onboard lens will capture images of what is in front of the jetbot car (see below left). This can be done by using jupyter notebook to control the car movements (forward, backward, stop, left and right) and then take pictures at various positions where the car is in. The use of ipywidgets serves as a way for us to interact with the car’s movement controls and facilitate the image capturing.

A picture containing person, person, large, table

Description automatically generatedIpywidge Controller is configured with the following joystick/button setting as shown below:

Left joystick Axis 2 Y slider Left joystick Axis 3 X slider Button 3 Straight Button 2 Go left Button 1 Go Right Button 13 Save Image



Note: The USB thumbdrive to communicate wirelessly with the gamepad controller must been inserted to the client machine on the remote side running Jetson Nano Bot’s Jupyterlab.

Data collected from the images are placed into the respective folders representing different classes of object. These images can be classified by five classes of objects. Other objects such as Traffic Light and Traffic sign will be added soon. Traffic Light can be further classified as Red Light, Yellow Light and Green Light. Traffic sign can have Stop sign, Right Turn, No Entry etc. Images are in jpeg format.

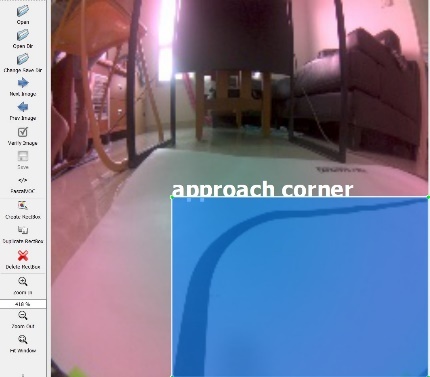
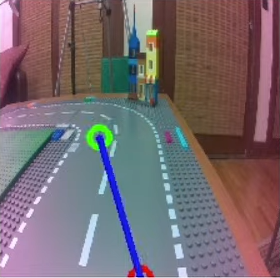
For inference using regression method, the images gathered for training dataset are 1000 images.

For inference using Object Detection method, the image gathered for training dataset are 816 images and validation dataset are 20 images.

The test dataset for both methods are 168 images.

Image pre-processing is performed on these image capture by the Jetson Nano Bot. These images are resized to 224 x 224 pixels and reshape to (1, 224,224, 3) to add a new dimension for batch size. Augmentation is used to improve the training accuracy by generating additional images with random horizontal flip.

**Annotation**



Pinpoint the location of x, y with a cursor Object labelling using LabelImg

For the regression method, the image is annotated using the gamepad controller to move the cursor to identify/locate the spot where the desired direction for the Jetson Nano Bot to moved towards (see above left figure). The annotated position given by the x-coordinate and y- coordinate are written to the filename of the image. For example, the filename is called “xy\_045\_097\_300e4e88-63cc-11eb-b891-72b5f773b75d” where x = 45 and y = 97. A unique identifier is appended to the back of the filename so that all filename would be unique. A total of 1000 images are annotated.

To label objects in the object detection method, the objects in the image are labelled by drawing a bounding box over the area of the desired object (see above right figure) using LabelImg. The targeted class of that object is selected and this information is stored as an XML file in Pascal VOC formal. A total of 816 images are annotated for the training dataset.

TFOD requires dataset to be saved in Tfrecord format. To create the required TFRecord for training set and testing set from the image files and xml files in training and testing directory respectively. After tfrecords are created, These tfrecords (“project\_train.record-00000-of-00001” [816 images] and “project\_test.record-00000-of-00001” [20 images]) they are stored in the data directory.

**Proposed Plan**

We intended to do 3 modules. For the road following, we decided to try on Deep Learning on Regression model and Object Detection. For the Object of Pedestrian, Vehicle, Traffic Sign and Traffic Lights detection, we intended to combine with existing road following so that the solution is as a whole.

|  |  |
| --- | --- |
| **SN** | **Plan Description** |
| 1 | Road Following using Regression |
| 2 | Road Following using Object Detection |
| 3 | Road Following with Pedestrian, Vehicle, Traffic Sign and Traffic Lights detection |

**Road Following using Regression**

Initially, we start off using classification of 3 classes (left, center, right). After experimenting with different combination of motor steering power and speed, the most it can have 2 round successful tracks but will soon it will run off the tracks unless we have more classes to define different turning angle depend on the car lane position. We research further and found that we can use either regression or objection detection to determine the left and right motor speed. Since our team have 4 members, we decided to try on these 2 methods and further divide into 2 sub-groups.

Since it is the regression, we will need to get the desired target location of x and y where the desired direction for the car to move toward to. Based on the official paper from the Nvidia and Jetbot video in the youtube, we notice it used gamepad/joystick and easier help to pin the location and capture the x and y co-ordinate and save as part of the file name of the image. We decide to invest to get a joystick to help us on data collection. The detail of data collection is discussed in the “Data Collection” section.

After collected the data about 1000+ image, we need to train our model preferable using GPU power as we have a lot of image to process it using NN model. Since the Jetbot has some limitation, we need to have a model which is suitable for mobile device to fix in and powerful enough to be train and predict the result. From the official web site, we notice there is pre-trained model (ResNet18 model trained on millions of images) for a new task that has possibly much less data available. We decide to use this pre-trained ResNet-18 model available on PyTorch TorchVision.

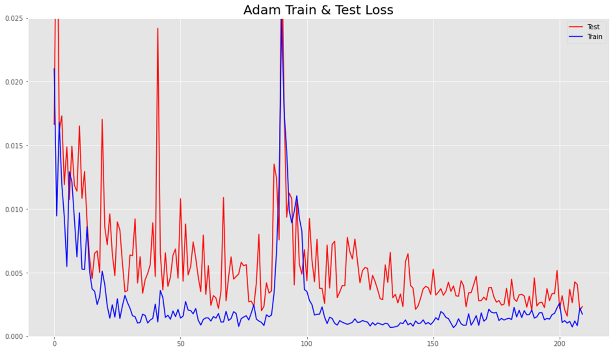
Makes use of transfer learning for the regression model for road following. The robot takes in a frame from the camera stream and the model’s goal is to get (x, y) coordinate for the green dot which the robot follows. Description of the ResNet18 model:

|  |  |
| --- | --- |
| Image for post |  |

ResNet18 Architecture

After extracting the (x, y) coordinates from the name of each picture path from the images that were taken in the data collection. After that, we do pre-processing. The pre-processing function returns the image with a tensor of the (x, y) coordinates of the green dot, the image and its label.

After preparing the data set, we proceed to do training, testing and evaluation to find out the accuracy of the model. Training the regression model, for this we load the image and the (x, y) label onto the ResNet18. For each epoch iteration, we calculate the loss with the mean squared error function, and the optimization is with Adam optimizer.

After making some fine tuning to the regression model, we were able to achieve good results in the training of the model. The accuracy on the test set is almost approaching that of the training set. The changes were increasing the test set slightly, reducing the jitter distortions and increasing the length of training.

After we have train the model, we need to do quantization of the model so that the performing computations and storing tensors at lower bit-widths than floating point precision which allow Jetson Nano to have a more compact model representation and the use of high performance vectorised operations.

After loaded the model into the Jetbot, we need to do fine turning of the Jetbot based on the speed gain, steering gain. Because of that, we need to use ipwidgets tools to help us interactive adjust it. Not only that, we need to calculate the Jetbot wheel motor speed values. Thanks to Nvida official web site, they provide formulae to calculate it as follow:

1. Get the angle using arc tangent of x and y co-ordinates
2. Based on the angle from the center of the Jetbot and sum the angle of steering gain plus the differences of current angle and previous angle and multiple with D gain to get now much to increase or decrease the left and right motor.

speed\_slider.value = speed\_gain\_slider.value

angle = np.arctan2(x, y)

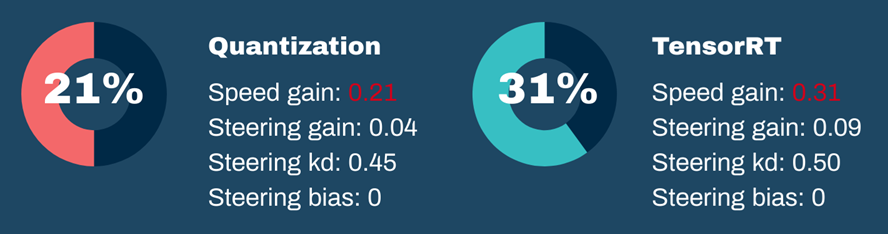
pid = angle \* steering\_gain\_slider.value + (angle - angle\_last) \* steering\_dgain\_slider.value

angle\_last = angle

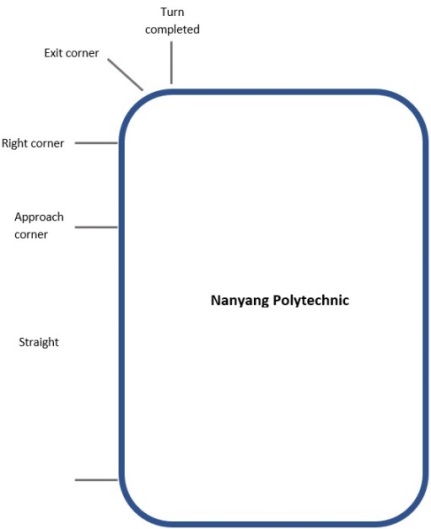
steering\_slider.value = pid + steering\_bias\_slider.value

robot.left\_motor.value = max(min(speed\_slider.value + steering\_slider.value, 1.0), 0.0)

robot.right\_motor.value = max(min(speed\_slider.value - steering\_slider.value, 1.0), 0.0)



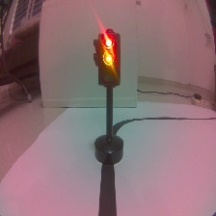
Based on the NVIDIA, TensorRT is a better high-performance deep learning inference than just use quantized file in the Jetbot. We decided to try both of them. To our surprise, the Jetbot with TensorRT engine can run up to max speed of 0.31 with no off the track running Jetbot for half an hour. As for Jetbot without TensorRT engine (only Quantization), the max speed is only of 0.21 with no off the track running Jetbot for half an hour. This proves that the TensorRT is a much better high-performance deep learning inference than quantized model.



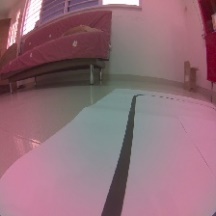
**Road Following with Pedestrian, Vehicle, Traffic Sign and Traffic Lights detection**

Since most of the code is given by Nvida with our slight modification to suit our requirement and we decided to do some advance enhancement to include object detection of Pedestrian, Vehicle, Traffic Sign and Traffic Lights detection. Since we have only a single lane, we could not manoeuvre the car to another lane. Therefore, some of the Traffic sign is not suitable as we cannot change lane. We can only set the car to go forward or to stop. Based on that, we have two classes namely stop or go and we decide to do 2 classes of classification NN model. We also notice that there is also official site from Nvida about collision avoidance which has similar concept. For the collision avoidance of the object detected, it will either move left or right based on your design. We decide we can make use of it and modified to according to our needs. Instead of manoeuvre the car to another direction, we stop the car by objection such as Traffic light (Red, Amber) and Traffic sign such as Stop sign etc. Here are some of the data collections:

Sample Image of **Blocked** (316 images):

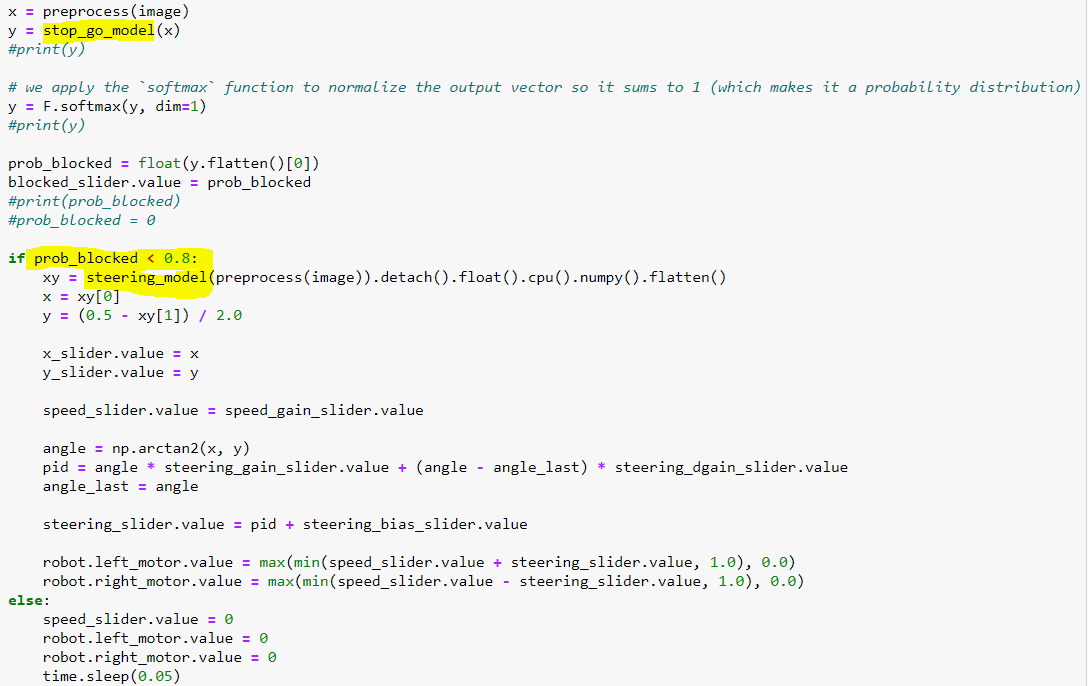


Sample Image of **Free** (1000 images):



We also use pre-trained ResNet-18 model with 1000 class labels. Since our classes label only have two class labels (Blocked and Free), we’ll replace the final layer with a new untrained layer that only has two outputs and train with “Blocked” and “Free” image. After we trained the model, we convert the model into TensorRT format.

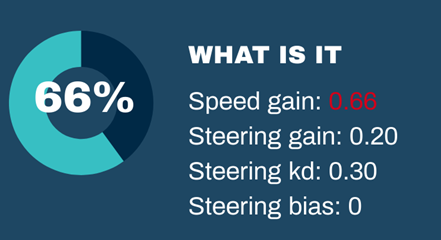
Now in the Jetbot, we have two model (Road Following and Road Obeying which follow traffic sign, traffic light and avoid collision). We create a check if there is blocked object detection detect based on the probability of more than 80%, we will stop the robot else it will continue with road following. Below is the source code to choose between road following or road object detection:



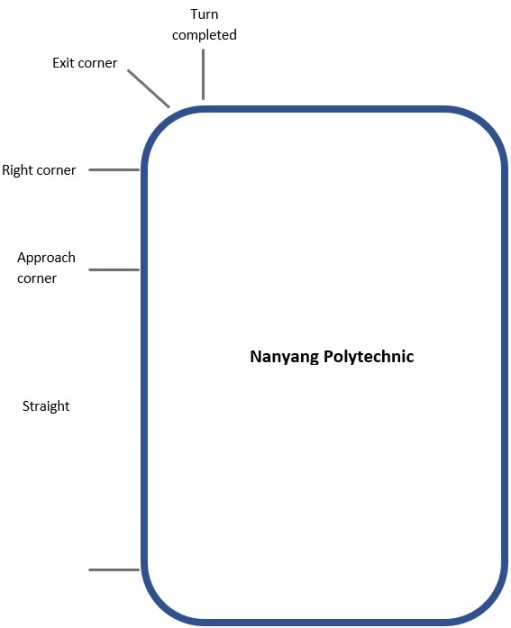
We notice there is a camera lag and after investigation there is too much image capture and causing a lot of backlog which causes camera lag and sometime Jetbot crashes due to error log full as it could not keep up the log recycle. To resolve this issue, we disable unnecessary log by set this in the Jetbot terminal to minimize Jetbot’s crashes:

Sudo service rsyslog stop

Sudo systemctl disable rsyslog

Another solution is to change to use Zmq camera instead of Jetbot camera library. We run a backend Zmq camera python script to reduce the screen speed to capture 21fps and clean up excess screen input so that there is no backlog and screen capture will be real time. This reduces the camera lag issues and greatly improves the speed of the Jetbot by increase the speed gain up to 0.7 

**Object Detection**

In the second part of self-driving car project, we will be using object detection for road tracking. The advantage is that it can do more tasks such as it can perceived the environment and act accordingly. The car can accelerate/decelerated based on the condition of the road by adjusting the power of the motor on the left and right wheels. It can detect objects like traffic lights, signs and pedestrians using the same DL model.

The cons being time taken to perform inferencing may exceed the limit due to the hardware platform used. In future, we can use better GPU on devices, a different quantisation method to optimise further and using Nvidia TensorRT for models.

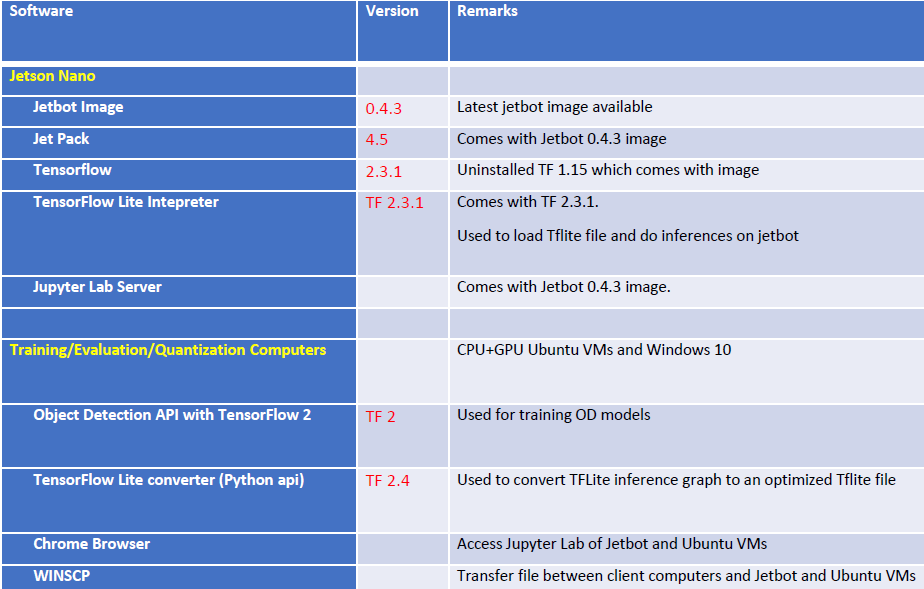
Applications suitable for deployment are: robots in the factory floor to transfer goods, cleaning and disinfection of shopping center and workplace, and patrolling of building complex for security and abnormalities in premise.

We have tried out three pre-trained models (SSD MobileNet V2, SSD MobileNet V1 FPN and SSD ResNet50). They are selected because SSD are the only detector support by TFLite in TensorFlow V2.3.1 for conversion to tflite files. MobileNet is selected due to the speed of the inference and ResNet50 for its accuracy in detection of the objects.

We performed transfer learning by initializing the model with the pre-trained weights and perform the training on all the classification layers and detection layers. The number of classes it can detect are 5 classes: *Straight*, *Approach corner*, *Right Corner*, *Exit Corner* and *Exit Corner Completed* as shown on the right.

In addition to enable the car to follow the road, we would also like the car to accelerate and decelerate depending the condition of the road. For example, when the car is on a straight road, the car can accelerate in speed by increasing the time it moves on the road before stopping for performing the next inference. When the car is approaching a corner or bend, it should start to maintain a slower speed. When the car is around the corner, it should reduce the speed, slow down and make more frequent check on the condition of the road when making the turn. It should continue to maintain the slow speed while keeping the car on track. When the car has completed making the turn, it adjusts the steering to straighten the direction and follow the road. When the car is on a straight road again, it could increase the speed and try to keep the car on the correct path on the road.

**Environment Setup**



Latest Jetbot image 0.4.3 is used to flash the micro sd card. Swap space is enabled and gui is disabled to conserve resource. Default disk size of 32 GB is resized to full 64 GB. TensorFlow Lite converter python api (requires TF >=2.4) is used instead of tflite\_convert tool as recommended by <https://www.tensorflow.org/lite/convert>.

1. Disable GUI to conserve memory

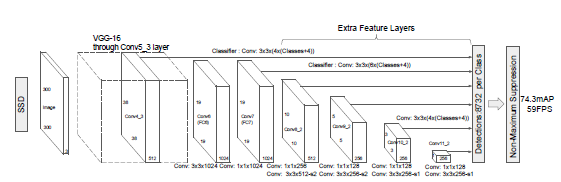
(./scripts/configure\_jetson.sh)

1. Enable swap memory

(./scripts/enable\_swap.sh)

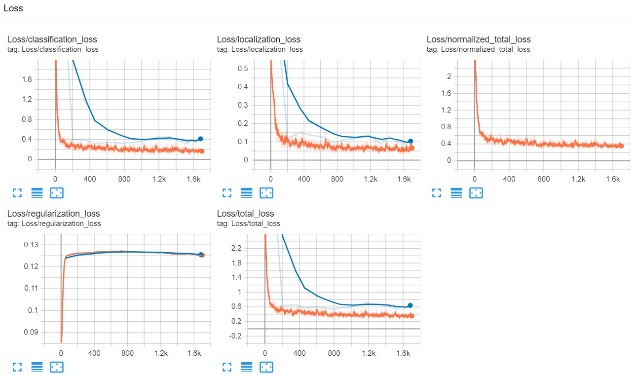
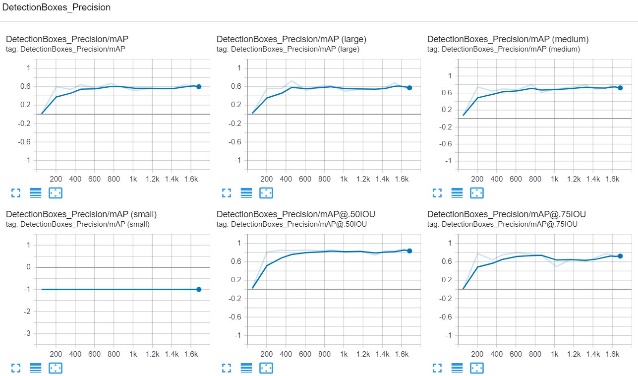
1. Default disk size of 32 GB is resized to full 64 GB
2. Connect WIFI without GUI

(sudo nmcli device wifi connect <SSID> password <PASSWORD>)

**SSD MobileNet V2**

The first model used was the SSD Mobilenet V2 as shown on the right. It is an efficient CNN architecture designed for mobile and vision application on embedded devices for real-time application. This architecture uses proven depth-wise separable convolutions to build lightweight deep neural networks. It splits all the convolutions on different subtasks. At the end it does approximately the same thing as traditional convolution but faster.

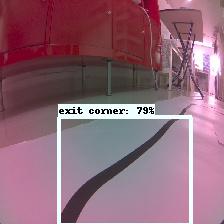
In addition, the author selected this model due to the road pattern being easily distinguishable from the 5 classes. For the SSD Mobilenet V2, the loss obtained was 0.301, overall precision and recall are 0.6 mAP and 0.7 AR. By using transfer learning, this accuracy was accomplished by training the object detection model for 1400 steps in around 50 mintues (see Appendix for more detail on performance metric graph).



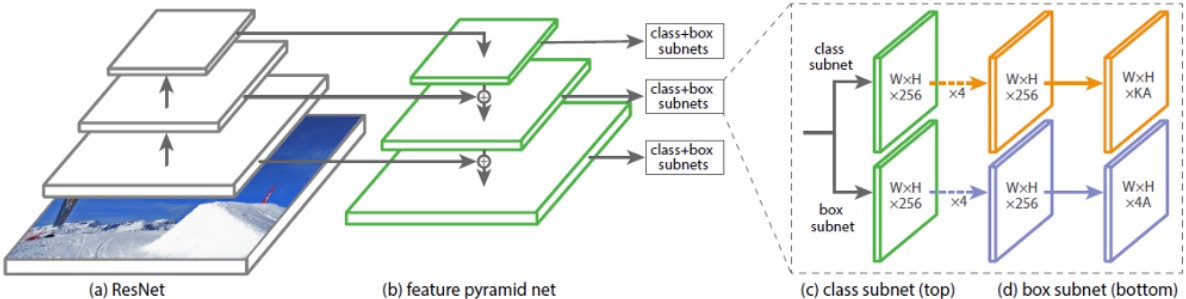
Detection of the 5 object classes on the Training data:



Evaluation on the Object Detection of the 5 classes on the Test data:

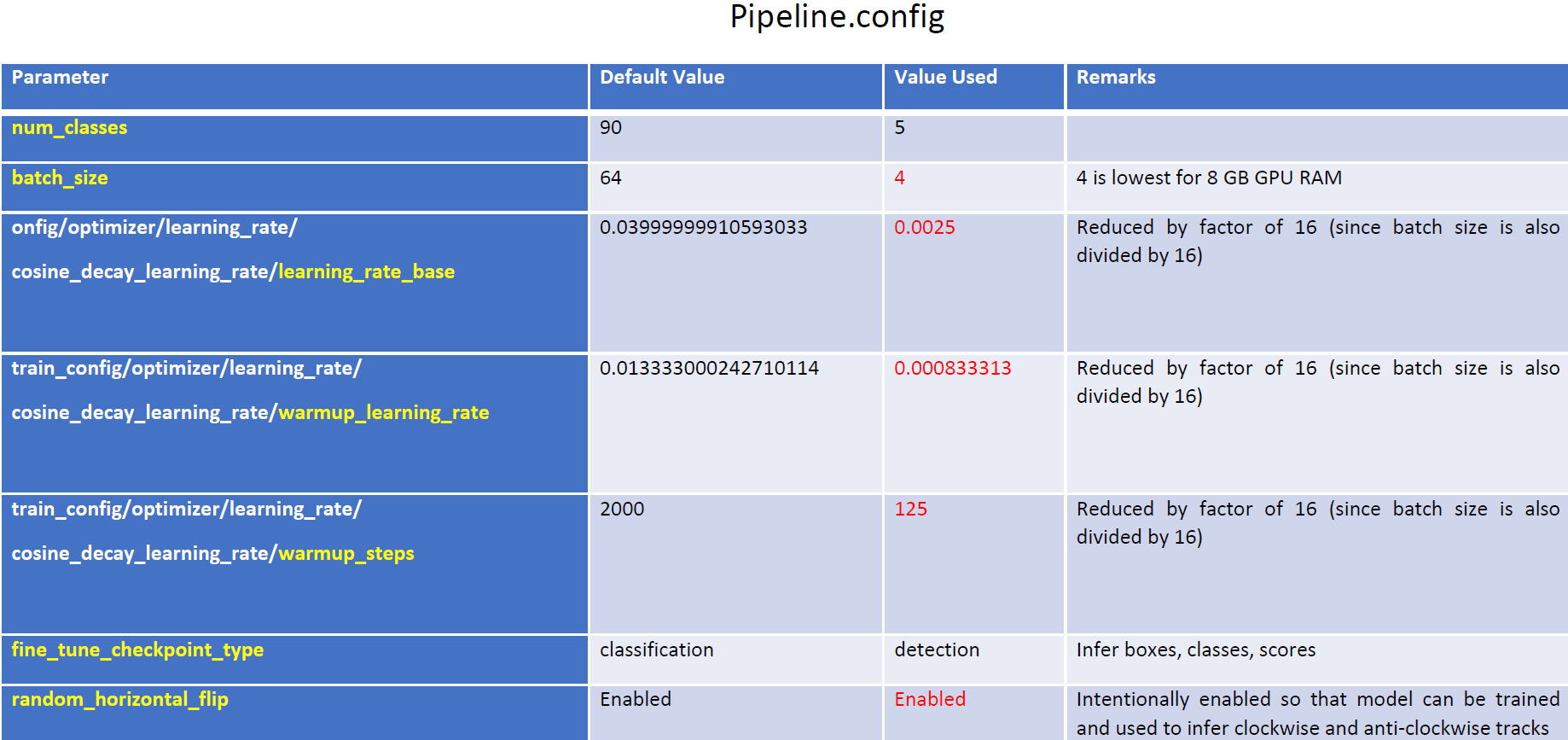


**SSD ResNet50 V1 FPN 640x640 (RetinaNet50)**



The RestinaNet50 is a SSD PFN object detection model based on the ResNet50 architecture trained with COCO dataset images. The published speed and COCO mAP of the pretrained model in tensorflow 2 object detection zoo is 46ms and 34.3 respectively. It trades greater latency for higher accuracy.

Below are the main changes made to pipeline configuration file for the customized SSD ResNet50 V1 FPN.

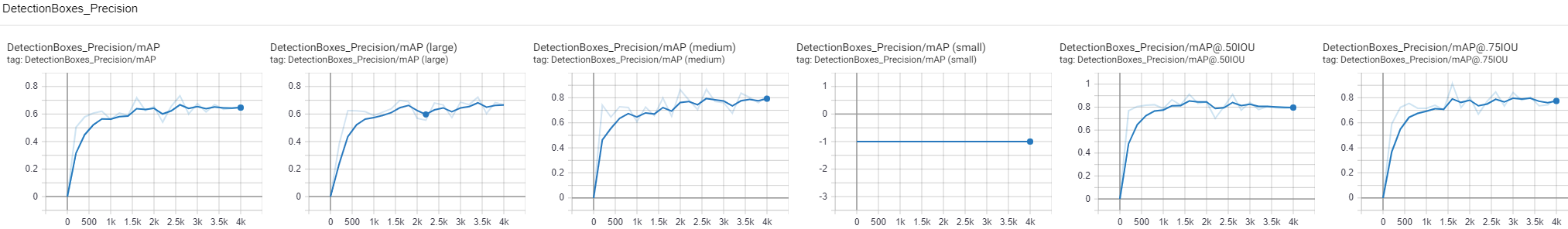


Batch size of 4 is used because other bigger values (i.e. 64,32, 16, 8) give out of memory error.

Because of the smaller batch size, we cannot use the high default learning rate of 0.04 (meant for default batch size 64) used by pretrained model. So we reduce default rate by a factor of 16 which is 0.0025. This gives a more stable and sustained training. We are not concerned about this rate being too high still because the learning rate will be gradually reduced as training proceeds by the Cosine Decay Schedule.

Warmup is usually needed to stabilise the initial training period when large batch size is used with high learning rate. The learning rate will be set smaller during the warmup period so that loss values will not have too drastic differences between steps. Because we are only using small batch size 4, and with a much reduced learning rate, there is no need for long warmup. Therefore, warmup steps is set to 125 steps.

Here are the TensorBoard charts generated:



DetectionBoxes\_Precision/mAP:

This plot is the mean average precision averaged over IOU thresholds ranging from .5 to .95 with .05 increments, and over all object sizes. Since it is mean AP (and not AP) therefore it is by nature averaged over the 5 classes. Optimal value at 3k steps is 0.6753.

DetectionBoxes\_Precision/mAP for large, medium and small objects:

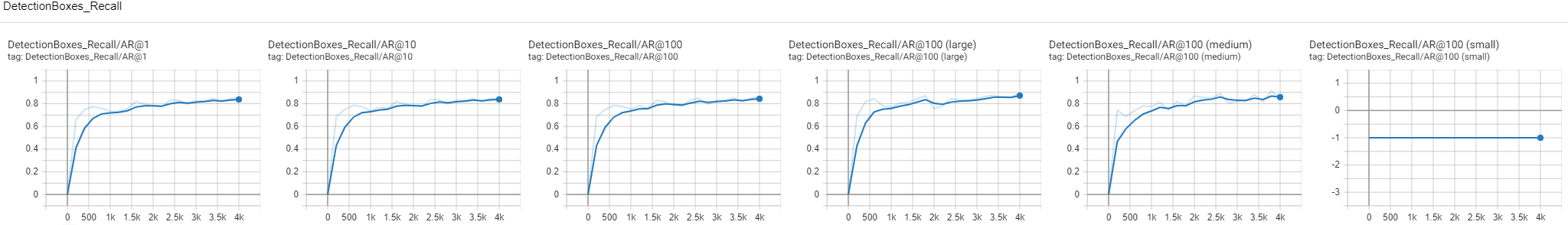
These plots are is breakdowns of “DetectionBoxes\_Precision/mAP” and is the mean average precision for small objects (area < 1024 pixels), medium objects (area 1024 to 9216 pixels) and large objects (area 9216 to 100m pixels). Our images are 224x224=50,176 pixels so there chance objects will fall under all 3 categories (small is unlikely due to annotated objects in training images is at least medium). Optimal values at 3k steps are 0.6859 (large) and 0.7611 (medium).

[DetectionBoxes\_Precision/mAP@.50IOU](mailto:DetectionBoxes_Precision/mAP@.50IOU):

This plot is the mean average precision at 50% IOU threshold (all object sizes). Achieving at least 50%IOU is a normal standard. We get optimal value of 0.8419 at 3k steps and is a good result.

[DetectionBoxes\_Precision/mAP@.75IOU](mailto:DetectionBoxes_Precision/mAP@.75IOU):

This plot is the mean average precision at 75% IOU threshold (all object sizes). Achieving at least 75% IOU is a very stringent standard. We get optimal value of 0.8419 at 3k steps and is a remarkable result. This means that at each recall value, we can get a higher precision level compared to baseline of 0.75 mAP (or 75% area under the PRC curve).

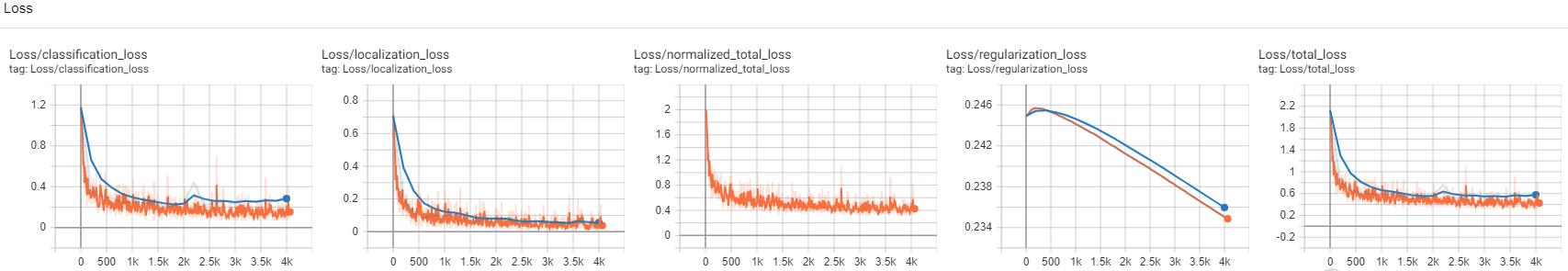


DetectionBoxes\_Recall/AR@1, 10 and 100:

Each of these plots are mean average recalls by the number of detections in the image. For example, AR@10 means that it will compute the mean average recall across all images with at most 10 detection, for all classes, and for all IOU thresholds from 0.5 to 1. We will see the plot with 100 detections since it is biggest set. We get optimal value of 0.829 at 3k steps and is a remarkable result.

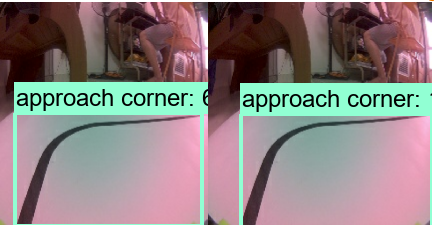
DetectionBoxes\_Recall/AR@100 large, medium and small objects

These plots are breakdowns of plot “DetectionBoxes\_Recall/AR@100” into different detected object size. Optimal values at 3k steps are 0.8188 (medium) and 0.8438 (large).



We can see that both classification and localization loss of training (blue) and evaluation (orange) curves are decreasing gradually, with the gap between them getting smaller which is a healthy sign. The regularization loss curves are decreasing which means that the regularization term in the loss function is adding bigger value to the overall loss function thereby causing any “useless” weights which do not contribute to good predictions to have small values. This is also a good sign showing overfitting is being managed properly. The lowest total loss is reached at step 3k (epoch 15) at checkpoint 16.

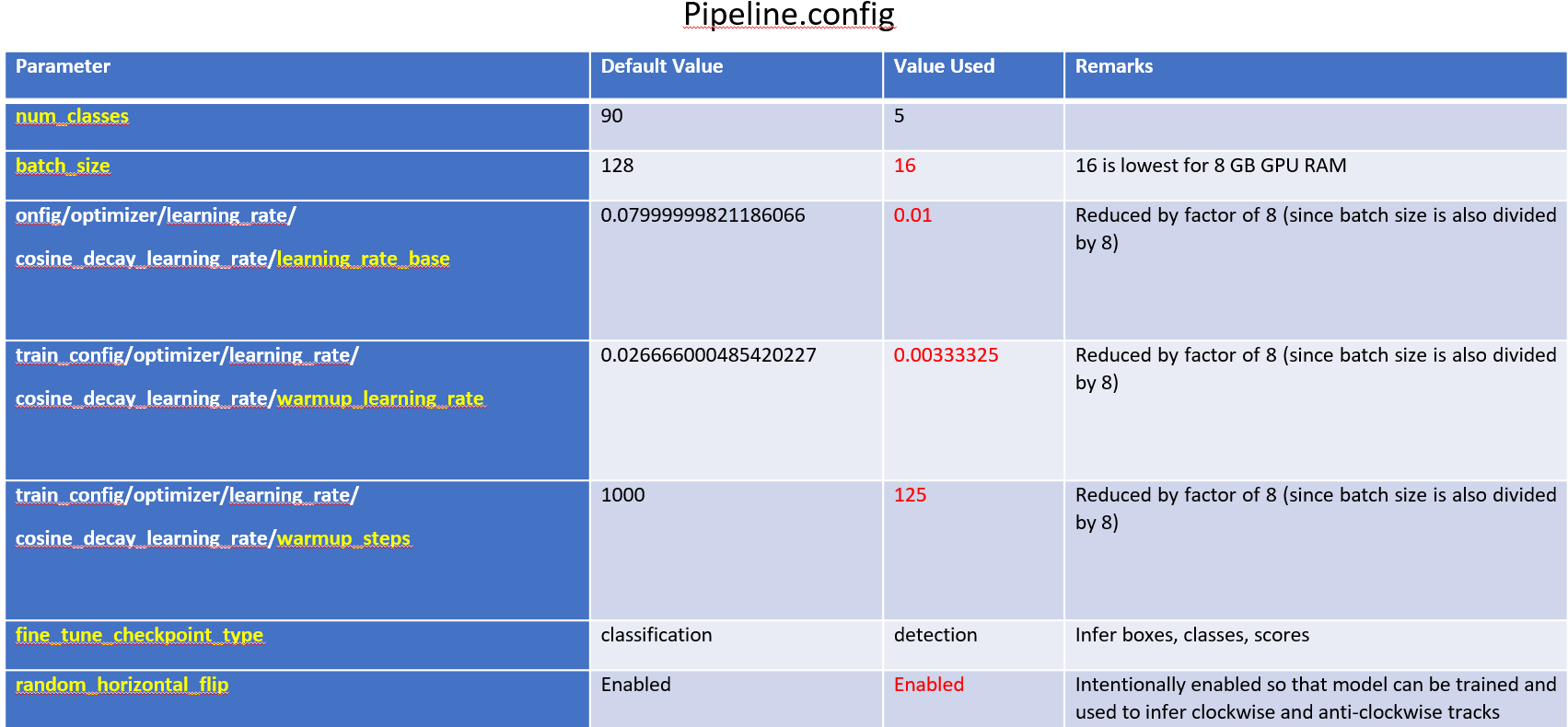
Tensorboard Evaluation Images (Left is predicted; right is truth):

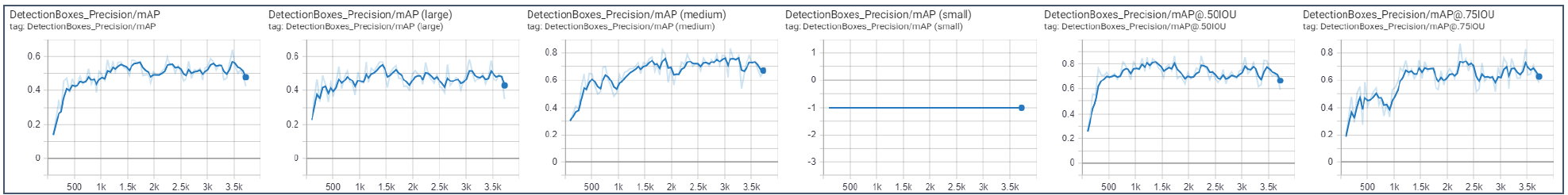
 

The evaluations results are quite accurate.

**SSD Mobilenet V2 FPN Lite 320x320**

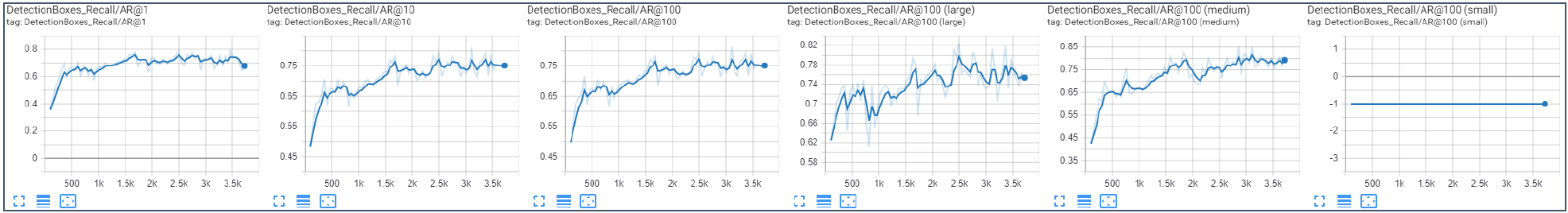


DetectionBoxes\_Precision/mAP; mAP for large, medium and small objects; [mAP@.50IOU](mailto:mAP@.50IOU); [mAP@.75IOU](mailto:mAP@.75IOU):



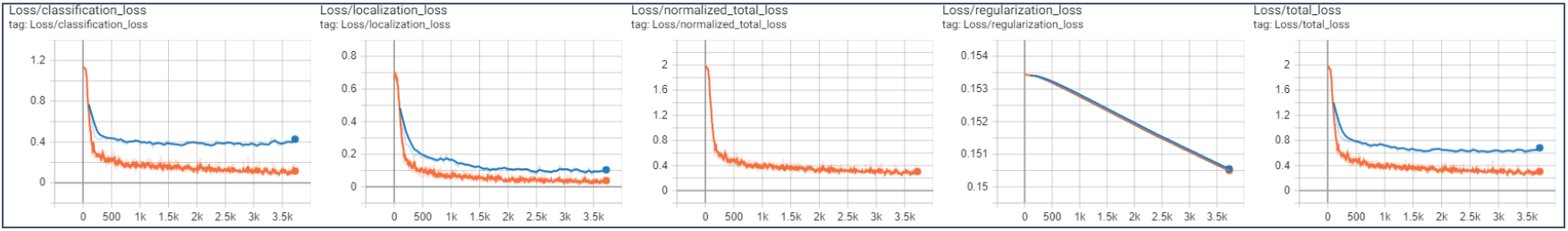
We get optimal value of 0.6387 for mAP at 3470 steps.

DetectionBoxes\_Recall/AR@1, 10 and 100; AR@100 large, medium and small objects:



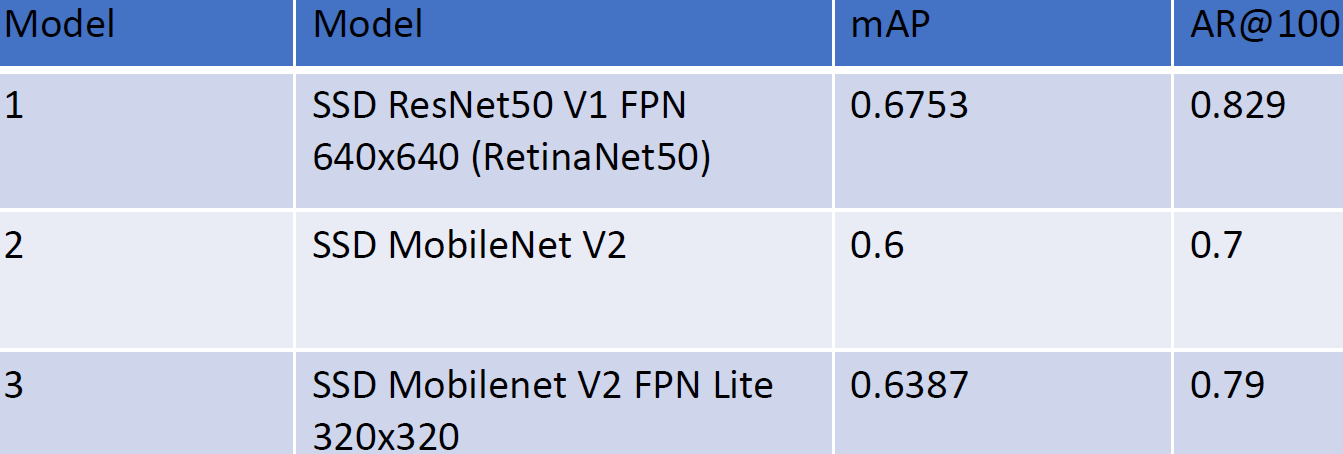
We get optimal value of 0.79 for AR@100 at 3470 steps.

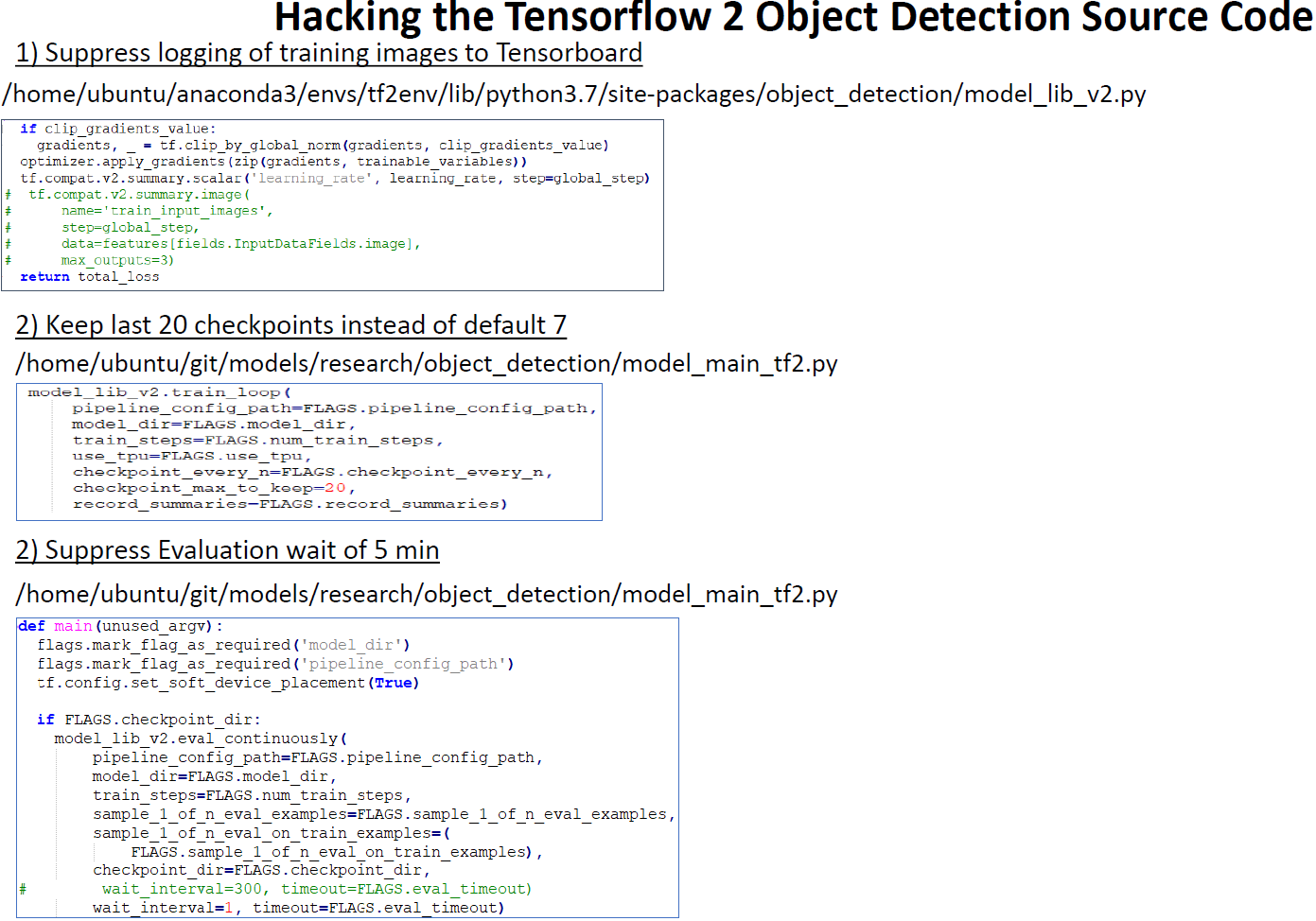
Loss/classification\_loss; localization\_loss; normalized\_total\_loss; regularization\_loss; total\_loss:



We get lowest total loss at step 3470.

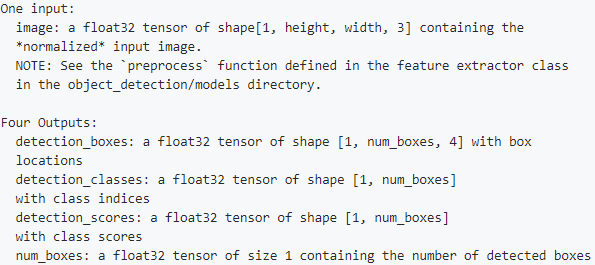
**Models Comparison**





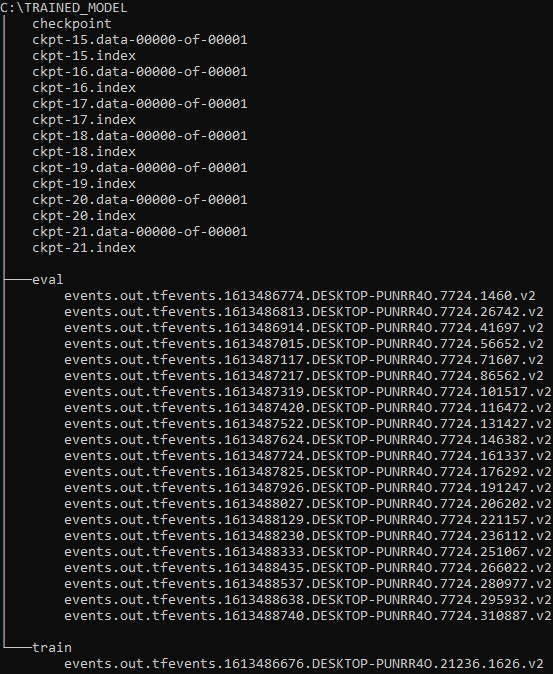
**TFLite Conversion**

TensorFlow Lite (TFLite) is TensorFlow’s lightweight solution for mobile and embedded devices (also om Jetson nano). It enables on-device machine learning inference with low latency and a small binary size. TensorFlow Lite uses many techniques for this such as quantized kernels that allow smaller and faster (fixed-point math) models. The following are the inputs and output formats of TFLiite interpreter:



Step 1: Export TFLite inference graph

This step generates an intermediate SavedModel that can be used with the TFLite Converter Python API which is the recommended way instead of the tflite\_convert tool.



set PIPELINE\_CONFIG\_PATH="pipeline.config"

set TRAINED\_CHECKPOPINT\_DIR="trained\_model\checkpoint"

set OUTPUT\_DIRECTORY="export\_tflite\_graph\_trained\_model"

python "models\research\object\_detection\export\_tflite\_graph\_tf2.py"

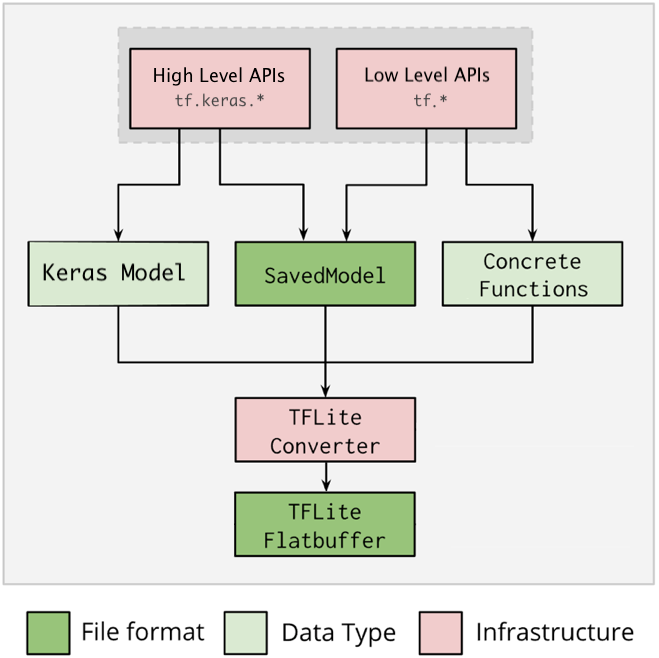
--pipeline\_config\_path=%PIPELINE\_CONFIG\_PATH%

--trained\_checkpoint\_dir=%TRAINED\_CHECKPOPINT\_DIR%

--output\_directory=%OUTPUT\_DIRECTORY%

Step 2: Convert to TFLite

This step uses the TensorFlow Lite Converter Python api to convert the SavedModel (from previous step) to TFLite format.



Based on diagram above, the conversion path taken for our project is:

High Level APIs -> SavedModel ->TFLite Converter -> TFLite Flatbuffer

converter= tf.lite.TFLiteConverter.from\_saved\_model("export\_tflite\_graph\_trained\_model\saved\_model")

tf.compat.v1.lite.TFLiteConverter

We also successfully tested with tf.compat.v1.lite.TFLiteConverter.from\_frozen\_graph with some pretrained models from TensorFlow 1 Detection Model Zoo. So Tensorflow 2 Lite Converter can still be used for TF1 models in the future if required.

converter = tf.compat.v1.lite.TFLiteConverter.from\_frozen\_graph(

graph\_def\_file=r"export\_tflite\_ssd\_graph\_ssdlite\_mobilenet\tflite\_graph.pb",

input\_arrays=['normalized\_input\_image\_tensor'],

output\_arrays=['TFLite\_Detection\_PostProcess',

'TFLite\_Detection\_PostProcess:1',

'TFLite\_Detection\_PostProcess:2',

'TFLite\_Detection\_PostProcess:3'],

input\_shapes={'normalized\_input\_image\_tensor': [1, 300, 300, 3]}

)

converter.allow\_custom\_ops = True

Note: In order to use tf.compat.v1.lite.TFLiteConverter, the SavedModel must be generated with export\_tflite\_ssd\_graph.py instead of export\_tflite\_graph\_tf2.py during step 1:

set PIPELINE\_CONFIG\_PATH="ssdlite\_mobilenet\_v2\_coco\_2018\_05\_09\pipeline.config"

set trained\_checkpoint\_prefix=”ssdlite\_mobilenet\_v2\_coco\_2018\_05\_09\model.ckpt"

set OUTPUT\_DIRECTORY=”export\_tflite\_ssd\_graph\_ssd\_mobilenet\_v2”

python "models\research\object\_detection\export\_tflite\_ssd\_graph.py"

--pipeline\_config\_path=%PIPELINE\_CONFIG\_PATH%

--trained\_checkpoint\_prefix=%trained\_checkpoint\_prefix%

--output\_directory=%OUTPUT\_DIRECTORY%

--add\_postprocessing\_op=true

Post-training quantization

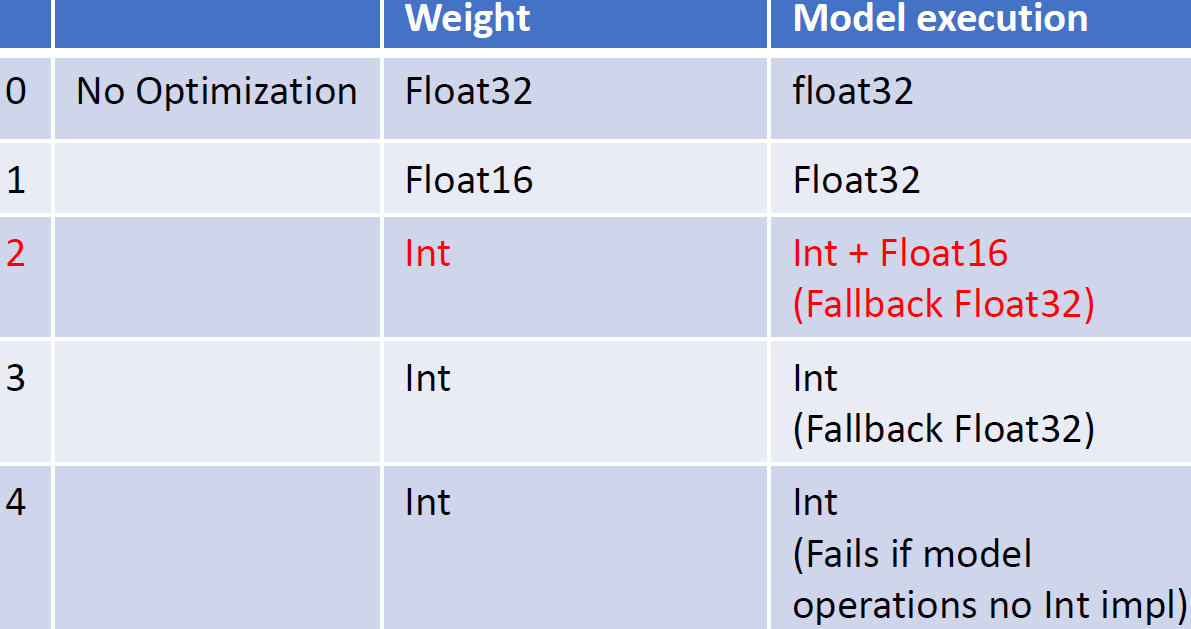
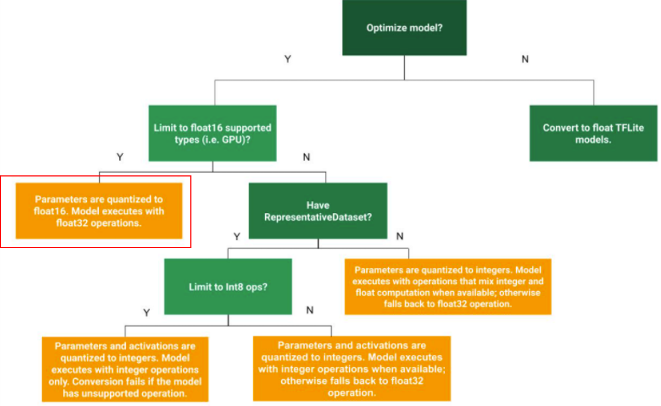
We choose to use Float16 quantization (see red box below) among the various post-training quantization options. This is because it will not degrade accuracy that much while the models we used such as SSD mobilenet v2, is quite light weight and fast and thus does not need the full benefits of performance that comes with more sophisticated quantization methods such as Full integer quantization.

Benefits of float16 quantization:

1. Model sized is halved since float32 becomes float16
2. Loss of accuracy kept at minimal
3. Supports GPU delegate which can operate directly on float16 data, resulting in faster execution than float32 computations.

Possible disadvantages of float16 quantization:

1. limited latency reduction as compared to integer quantization (our selected models are light enough to forego this)
2. limited to GPU since CPU will dequantize a already quantized model (Jetson nano runs with GPU during inference so this is not an issue)



converter.optimizations = [tf.lite.Optimize.DEFAULT]

converter.target\_spec.supported\_types = [tf.float16]

tflite\_quant\_model = converter.convert()

open("trained\_model\_float16\_q.tflite", "wb").write(tflite\_quant\_model)

The following are the performance when tested with inferring an image:

|  |  |  |
| --- | --- | --- |
| **Model** | **No Optimization** | **Float 16 Optimization** |
| Custom SSD Mobilenet V2 | < 1 sec | << 1 sec |
| Custom SSD ResNet50 V1 FPN | > 5 sec | >> 5 sec |

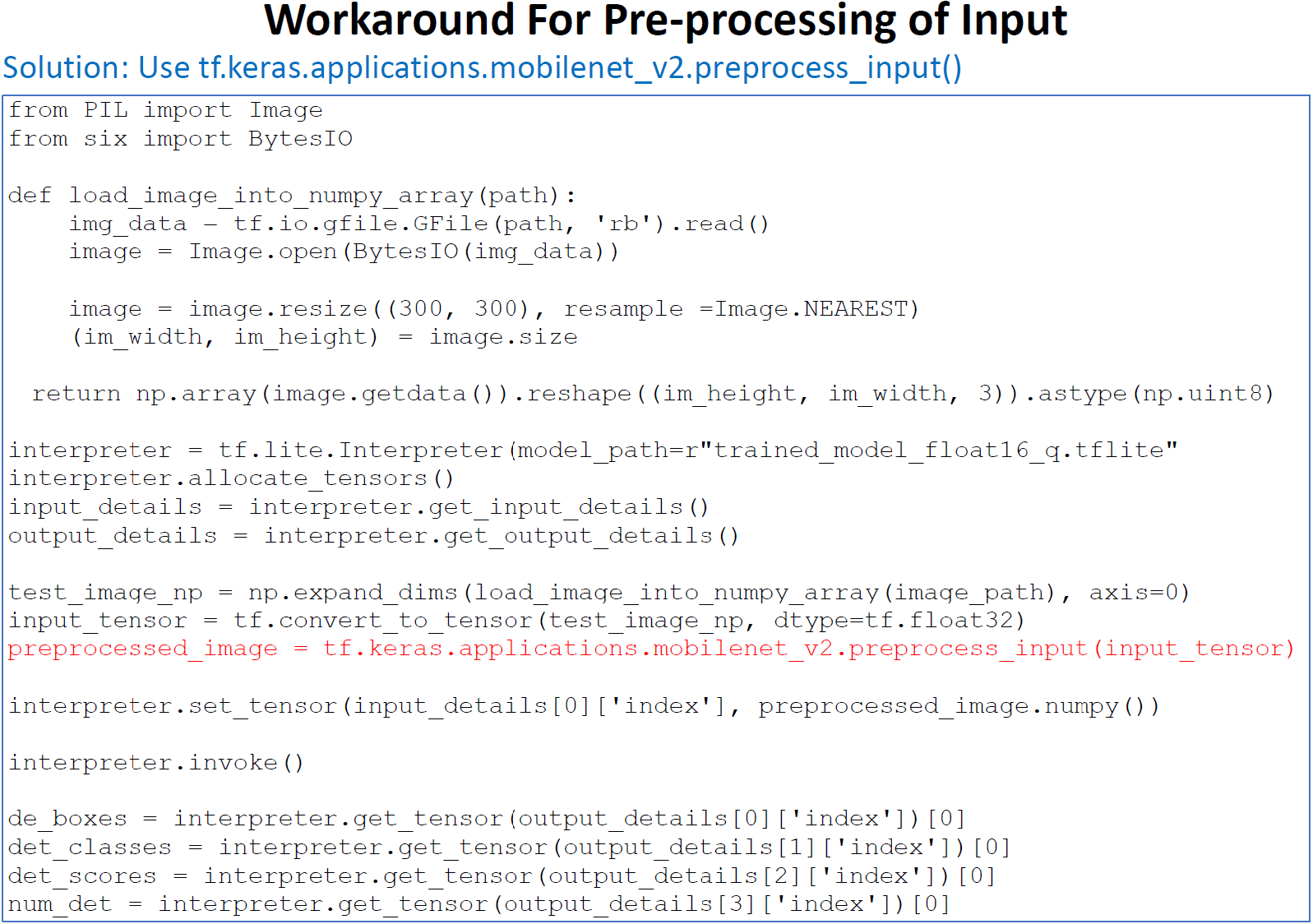
In view of the faster response time for inference, we chose the custom trained SSD mobilenet v2 as our final model for deployment into the jetbot nano.

**TFLite Inference**

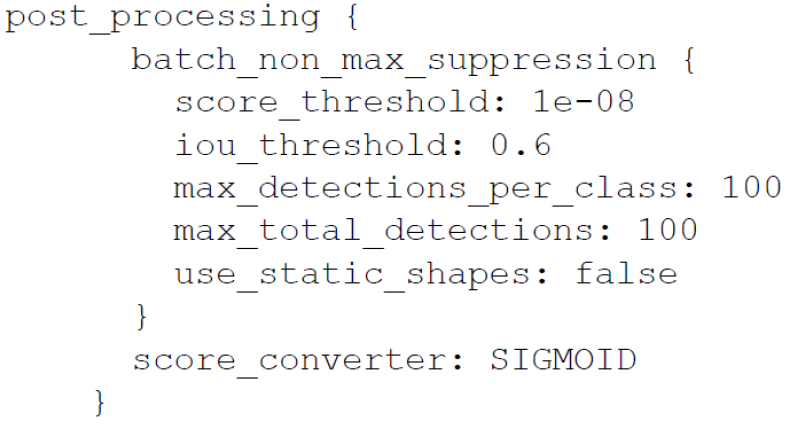
The inference process involves pre-processing steps (and optionally a post processing steps). Pre-processing is done on the input image by using the Keras function tf.keras.applications.mobilenet\_v2.preprocess\_input. We had initially planned to use the object\_detection.utils.get\_configs\_from\_pipeline\_file to get the pipeline configuration info from the pipeline.config file and then use it to build the detection mode by use of object\_detection.builders.build. Then with the model, we can invoke its preprocess function to pre-process the input image.



However, this will require us to install “Object Detection API with TensorFlow 2” on Jetson Nano which we failed to do so due to incompatible modules that comes preinstalled on jetpack 4.5. Therefore, we use tf.keras.applications.mobilenet\_v2.preprocess\_input as a workaround. The following code fragment shows the pre-processing and inference process:



However, as there are currently no support for post processing functions available in Keras, our application code needs to handle it. This is the post-processing config extracted from customized SSD mobilnet v2 pipeline.config:



This means non max suppression is not done and so there will be many nearby detections for a single object. So, the solution is that application code simply takes the predicted object with highest confidence score and ignore the rest. This is possible because we only expect 1 object to be detected for detecting lane class.

Predict Module

After passing through the Object Detection module to find out the condition of the road based on the bounding boxes, the classification result of the object is used by Predict module to decide on the steering direction of Jetson Nano Bot and the speed of the car. This is determined by the following the rules based on Decision Tree.

1. Divide the 224x224 image into 5 regions: *Far-Left*, *Left*, *Middle*, *Right*, *Far-Right*
2. Convert image to grayscale.
3. Extract two horizontal strips from the bounding box (5 x length of bounding box).
4. First strip is 50 pixels from bottom of BB.
5. Second strip is 10 pixels from bottom of BB.
6. Processing of the strip:
7. Convert to black and white with threshold of intensity value= 90.
8. Perform Erosion with filter = 5x5.
9. Perform Dilation with filter = 5x5.
10. Find the first pixel with intensity value = 0 (black colour pixel)
11. Calculate position and gradient from point p1 and p2.



The result of the Object Detection is one of the following:

i) Straight

iii Sub-region

3

5

4

1

2



Left

Middle

Right

i) Position of bounding box



Middle



Middle

ii) Gradient of road

First, determine whether the position of the bounding box in the image (see (i) above).

The area of the image is further sub-divided into 5 sub-portions:

*Far -Left* *[1]*, *Left* *[2]*, *Middle* *[3]*, *Right* *[4]* and *Far -Right* *[5]* as shown in (iii) above.

Direction of motion is determined by each of the 5 sub-portions (for right corner):

*Middle* *[3]*: No change in direction, speed of the car can be accelerated:

forward(0.4) # 0.4 second

*Left* *[2]*: left(0.2) # 0.2 second *Right* *[4]*: right(0.5) # 0.5 second

*Far-Left* *[1]*: left(0.5) # 0.5 second *Far -Right* *[5]*: right(0.5) # 0.5 second

ii) Approximate Corner



Left

Middle

Right

i) Position of bounding box (for right corner)

If distance from corner > 140 pixels then

Goto Function *Straight*

else

*Middle* *[3]*: No change in direction, speed of the car can be decelerated:

forward(0.2) # 0.2 second

*Left* *[2]*: forward(0.2) # 0.2 second *Right* *[4]*: right(0.2) # 0.2 second

*Far -Left* *[1]*: left(0.2) # 0.2 second *Far -Right* *[5]*: right(0.5) # 0.5 second

iii) Right Turn



Left

Middle

Right

i) Position of bounding box (for right corner)

If distance from corner > 100 pixels then

*Far -Right* *[5]*: right(0.2) # 0.2 second

*Far -Left* *[1], Left* *[2], Middle* *[3], Right* *[4]*: forward(0.2) # 0.2 second

Else

*Far -Right* *[5]*: right(0.2) # 0.2 second

*Far -Left* *[1], Left* *[2], Middle* *[3], Right* *[4]*: right(0.2) # 0.2 second

iv) Exit Corner



Left

Middle

Right

i) Position of bounding box (for right corner) )

*Far -Left* *[1], Left* *[2], Middle* *[3], Right* *[4], Far -Right* *[5]*: right(0.2) # 0.2 second

v) Exit Corner Completed



Middle (left corner) )



Middle (right corner)

ii) Gradient of road



Left

Middle

Right

i) Position of bounding box (for right corner) )

If (0 < gradient < 1) then # gradient is between 0 – 45 degree angle

right(0.4) # 0.4 second

else if (1 <= gradient < 1.732) then # gradient is between 45 – 60 degree angle

right(0.2) # 0.2 second

else if (gradient >= 1.732) or (gradient <= -1.732) # gradient is between 60 – 120 degree angle

forward(0.1) # 0.1 second

else if (gradient > -1.732) # gradient is between 120 – 180 degree angle

left(0.5) # 0.5 second

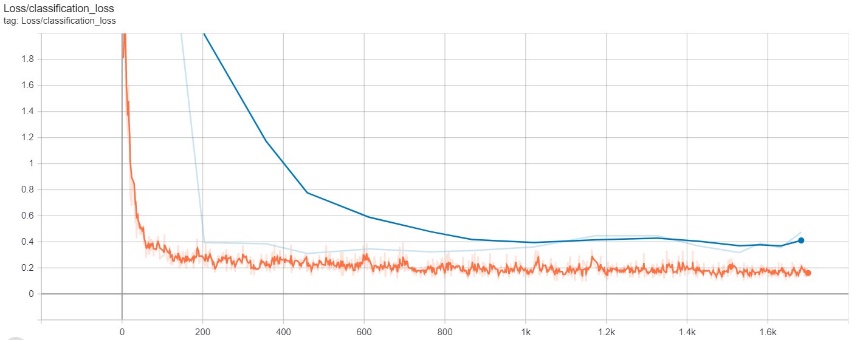
Prior research on the topic

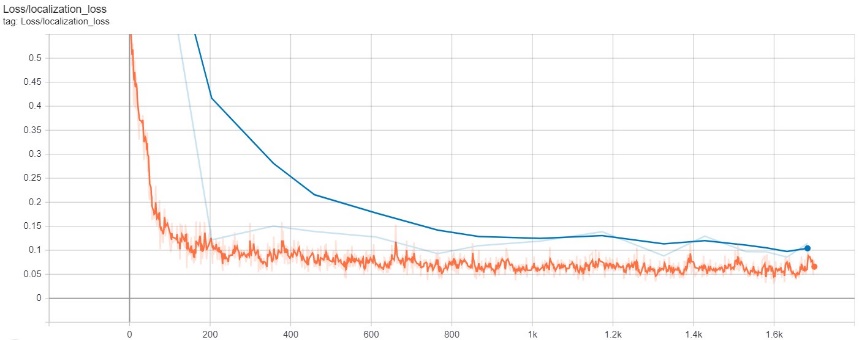
An example of prior research on a similar topic is:

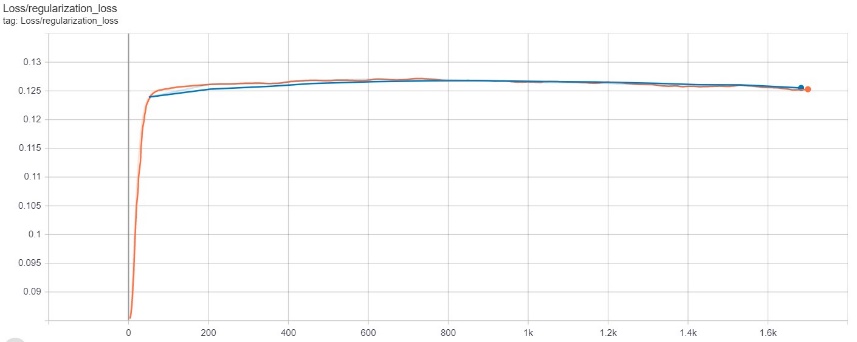
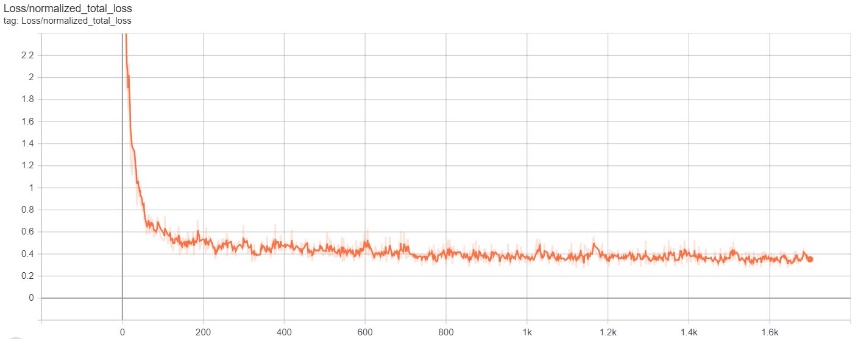
Islam, Chowdhury, Li, Hu (2019), VISION-BASED NAVIGATION OF AUTONOMOUS VEHICLE IN ROADWAY ENVIRONMENTS WITH

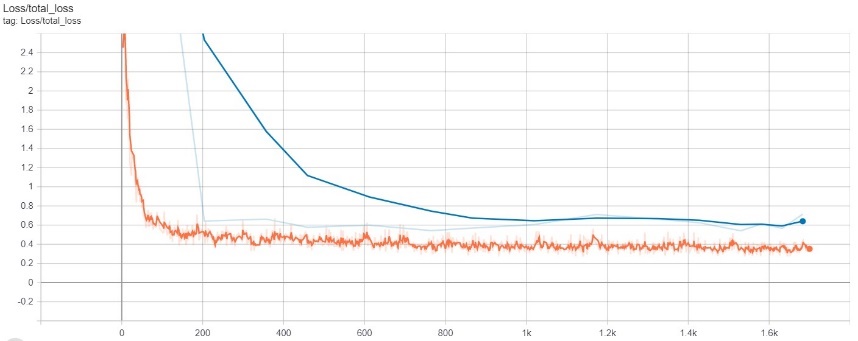
UNEXPECTED HAZARDS (<https://arxiv.org/ftp/arxiv/papers/1810/1810.03967.pdf>)

**Appendix**

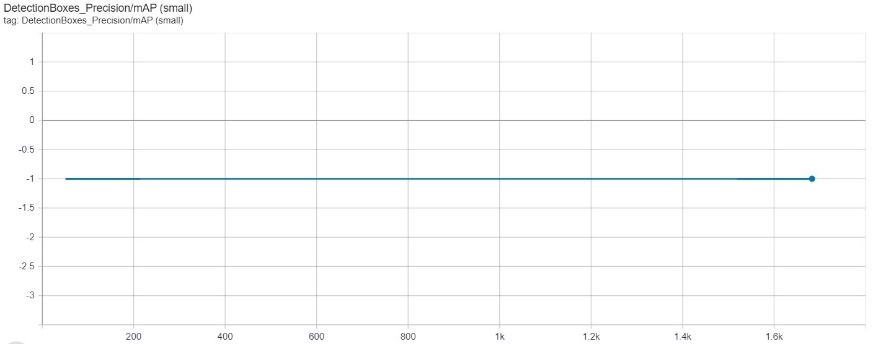
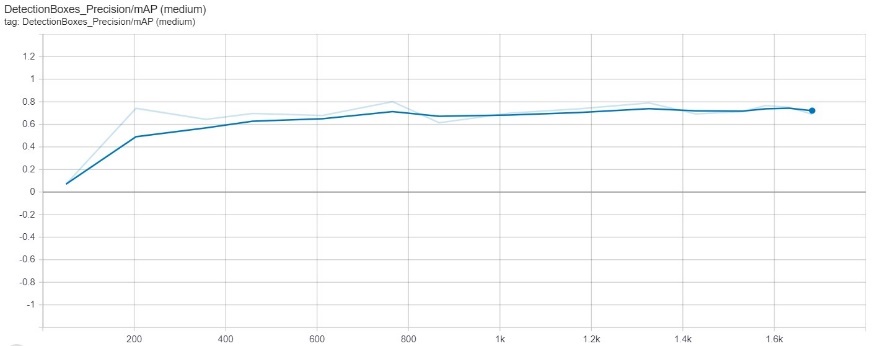
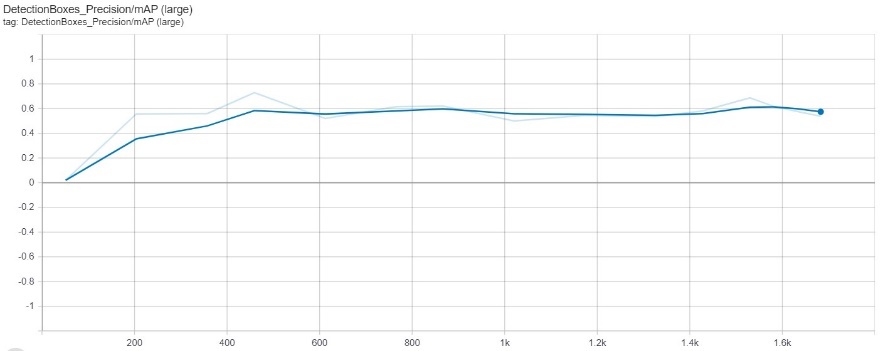
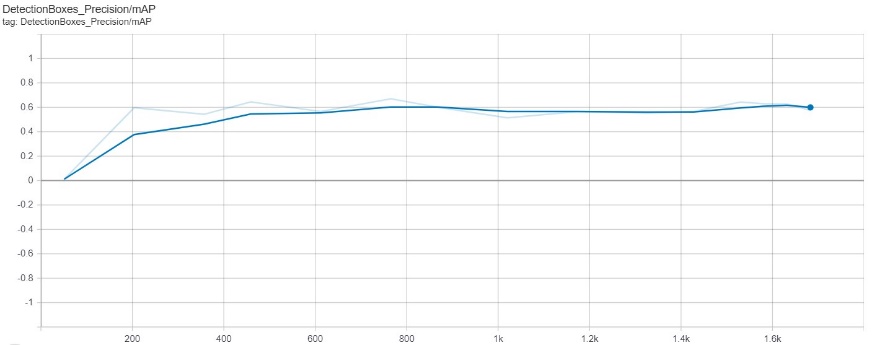


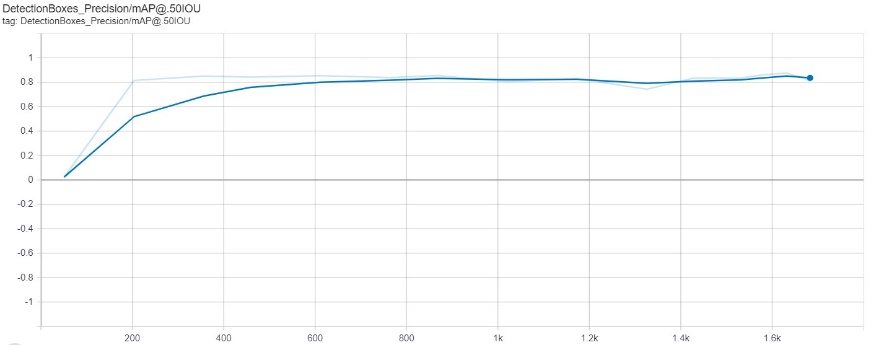


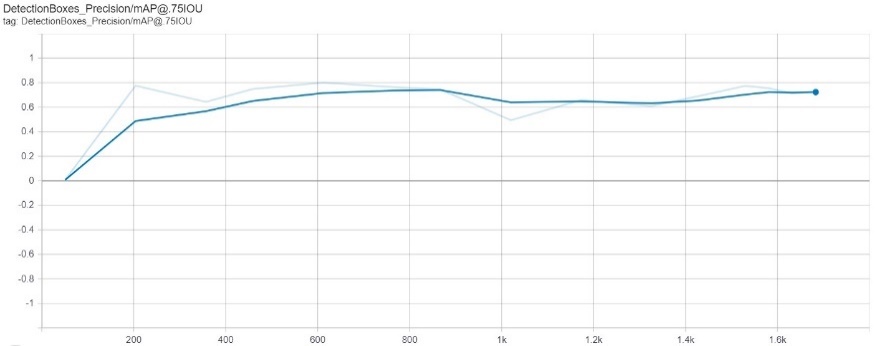




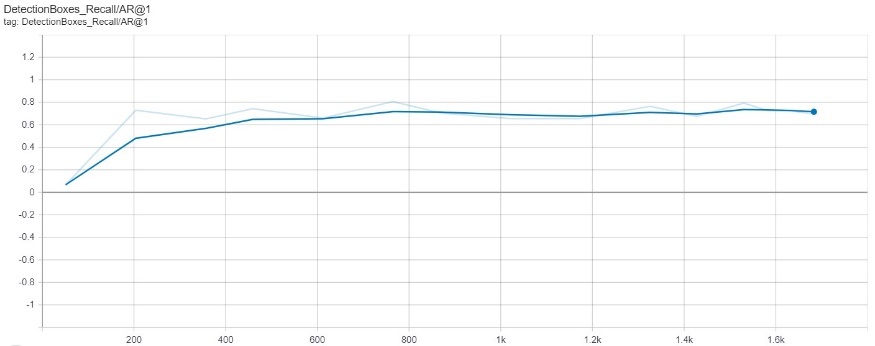
Loss of the SSD MobileNet V2

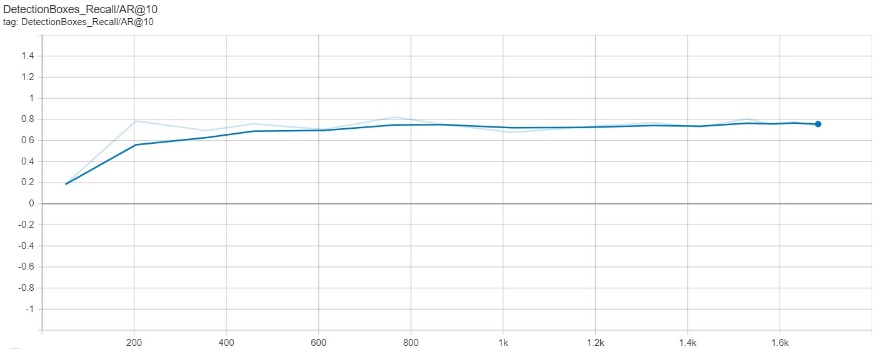


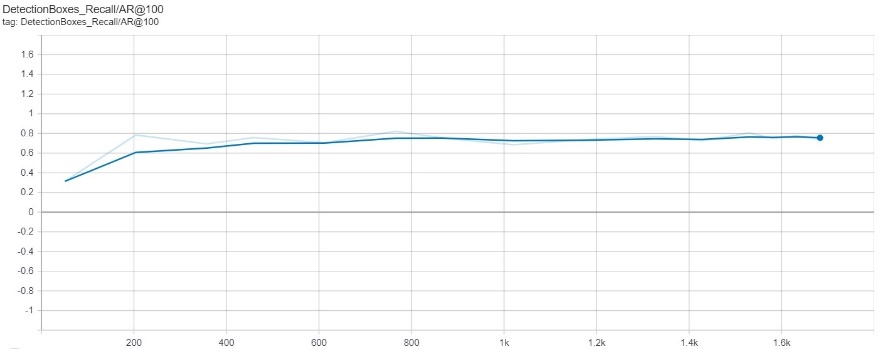


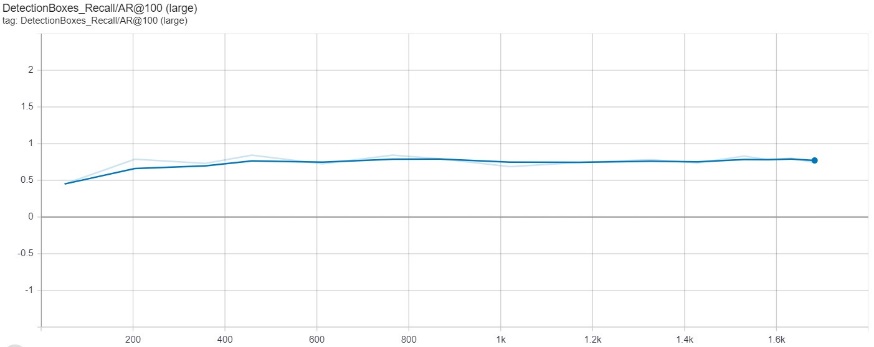


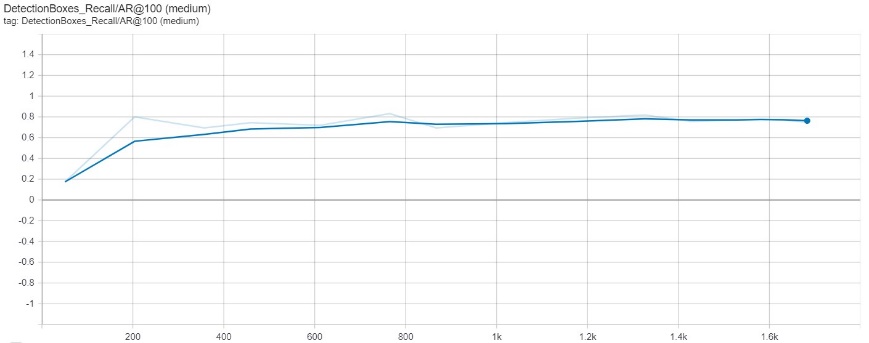
Precision of the SSD MobileNet V2

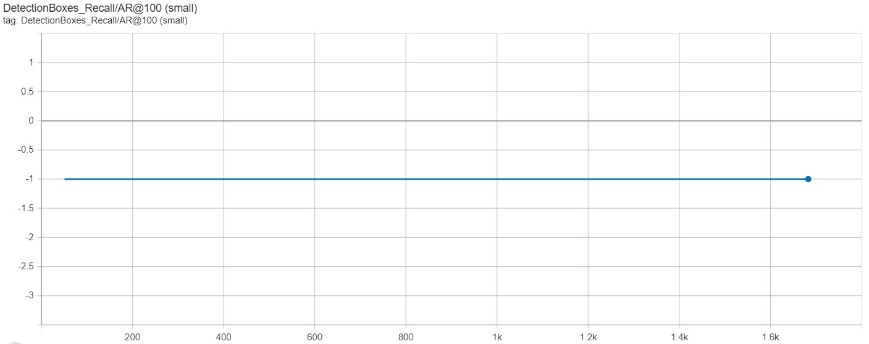












Precision of the SSD MobileNet V2