Artificial Intelligence Final Project:
Car Identification Expert

CPSC 481

Fall 2020

Aatib Abdullah
CPSC 481-05
Saturday, 1:00 – 3:00 pm
Alexander Gauf

Abstract

The Car Identifier is an expert system that would be running on a computer with data such as photos or frames of video so the system could work. This software will identify cars based how the body of the car is structured. This software can be applied in the area of Traffic systems and law enforcement, to catch any vehicle breaking the law or suspicious activity. The software would used trained models for its operation.

Introduction

A. Context

This system is to be implemented in computers connected to the Traffic systems. If would use Photos and Video frames to identify cars. This will help the operator of computers to know what is the make and model of the car. So that way, we can have information about the owners car.

B. Need

This expert system is needed by Traffic Controllers so they do not have to rely on an external sources for their information. The software will assist them of what the make model, and generation is of the car.

C. Implementation Summary

The system will be implemented in a Computer OS and will be coded in Python. It will used APIs like TensorFlow for using some of the functions for our expert system. The TensorFlow API is used for setting and training the models. We will have a folder system that acts like a sample database for “training” and “testing.” The images sample database would have a XML file (created using LabelImg) of the portion of the image that would be used to test and train on. LabelImg is a program that helps us draw a box around a picture, and it would create an XML file version of it [2]. It would focus on that part of the picture. The images along with the XML files will have ten percent of it tested and rest would be in the training file. The car can be in any position when the photo or frame was captured since our data also comprised of different angels of cars. The photo or video frame passed into the program for processing. This includes converting XML files to CSV and converting that into TFRecords as explained in Implementation Details. We would then train our system using train script to make a trained model. After training, we test our model which we would set up a file with a trained and tested model ready for use which would be our main project. The processing, training, and testing of our models would be based on the images. It would come up with make, model, generation in addition to the percentage of accuracy.

Requirements:

Previous Knowledge:

Knowledge in Python at a proficient level is required. Knowing the Syntax, functions, and libraries can aid the development of the project. Be able to understand the basic concepts of Artificial Intelligence and Machine Learning.

Software Requirements:

- TensorFlow 1.15
- Anaconda Distribution
    - Has a Package Management System called Conda
    - Conda, which can be used in the command line, will be used to download TensorFlow 1.15
- Python 3.6
- Jupyter Notebook
    - An online web application that can be used as a compiler.
    - Downloaded through Conda command line
- Label IMG script (provided in Implementation Details)

Our program must recognize the restaurant or hotel signs by the camera pointing towards the sign. All we need is Images that have restaurant or hotel signs on them. The images must contain the logo of the food chain on a road sign or at least provide a logo.

Installation and Set-Up

We will be using a CPU version of TensorFlow V1.15. If you have a CUDA enabled GPU, you can download the GPU version of TensorFlow, which will make computing a lot faster compared to the CPU version. Note: We used an Anaconda environment with Python 3.6 and using a Windows 10 environment.

Please download the following packages:

- pip install pillow
- pip install lxml
- pip install jupyter
- pip install matplotlib

Once that is done, please go to the link for the repository holding the Object-Detection API https://github.com/tensorflow/models. We can also simply use git. Then extract the file [2].

We will then need to download the protoc executable at https://github.com/google/protobuf/releases. Since we are using Windows 10, we will download the protoc3.4.0-win32.zip. We may need to be scrolling and hitting the next tab since there now a newer version out there, but we need the V3.4.0. Once it is downloaded, extract it to a directory to where you plan to put your Object-Detection file named models. Then what we need to do is execute the protoc program (assuming you put it in the same directory) with the windows CMD opened at the same directory. Type the command:

"protoc-3.4.0-win32/bin/protoc" models\research\object_detection\protos*.proto --python_out=.

In a result, you should see some newly generated python files. The following two images show what files are there in the *protos* directory before and after executing the code.

File Organization/Set-up

We now have "models" directory, but let's create a new directory say "object-detection" (not to be confused with the directory in models/research/object_detection). Create a directory "data", "images", and "training". And add "train" and "test" directories in the "image" directory.

Approach:

Image Classification and Object Detection:

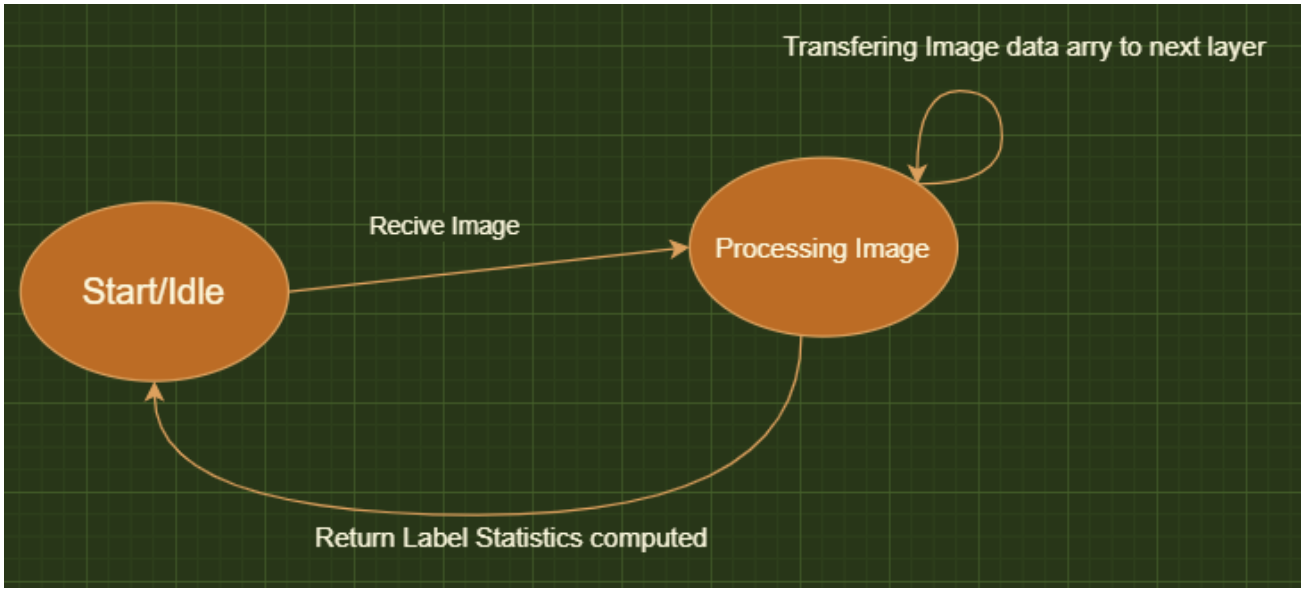
We had two approaches, Image Classification, and Object Detection. In an Image Classification approach, we identify an image as one class as in the restaurant/hotel signs. In the Object Detection approach, the program would detect multiple classes (cars) within an image. We initially went with the Image Classification route due to being relatively simple to implement and use. However, this approach was not feasible because, in an Image Classification approach, it would take an entire image/video frame and determine if the whole image/frame is any of the classes. The Image Classification approach would try to classify the entire sign into one whole class. The class would comprise of make, model, and generation. Having year would make detection a lot more harder since a car that is a model year apart are similar. Car models of a certain year range are classified into a generation since they have a similar design[1]. For example, we have a Nissan Altima with a 2009 model and 2010 model which both belong to the fourth generation which started from the 2007 model. Using generation as part of our class would be helpful. Using object detection, it would identify the different cars, and classify them in separate classes. So it is better to implement an Object-Detection oriented approach, where we would treat the cars as objects.

The Use of TensorFlow:

In this approach to the project, we used TensorFlow, a machine-learning platform for training the program on recognizing individual images. It has a wide range of APIs (Application Programming Interfaces), where we can use them to make our software. There is no need to create models by ourselves, and the models are provided for us. That is the reason why we choose TensorFlow, so we can use models that were created before and implement them for our purposes.

Implementation Details

A. Finite State Automata Graph



B. Algorithms Used

1) Detection Algorithm

The detection Algorithm is used to detect the object in the image. We would look at the image to look for the vehicle and determine its make and model. The code uses some methods from the Tensorflow API which are called Tensors to help it process the Image. Here is the pseudocode.

```
Def process(imagearray, detection_graph):
With graph set as default():
Start tensforlowsession:
    For key in List = {numdetect, detectionbox, detectscores, classes, detectionmasks}
    Tensorname = key
    If detectionmake in Tensorname:
            Process the image and find the object in the image
            Determine what class does the object belong to
Image_tensor =  get image tensor via accessing Tensorflow function getdefualt graph
Output = tensorflowsession(tensordictionary, image_tensor expanded dimensions by 0 )
Convert output to float
Return Output
```

2) Outputting the Image Algorithm

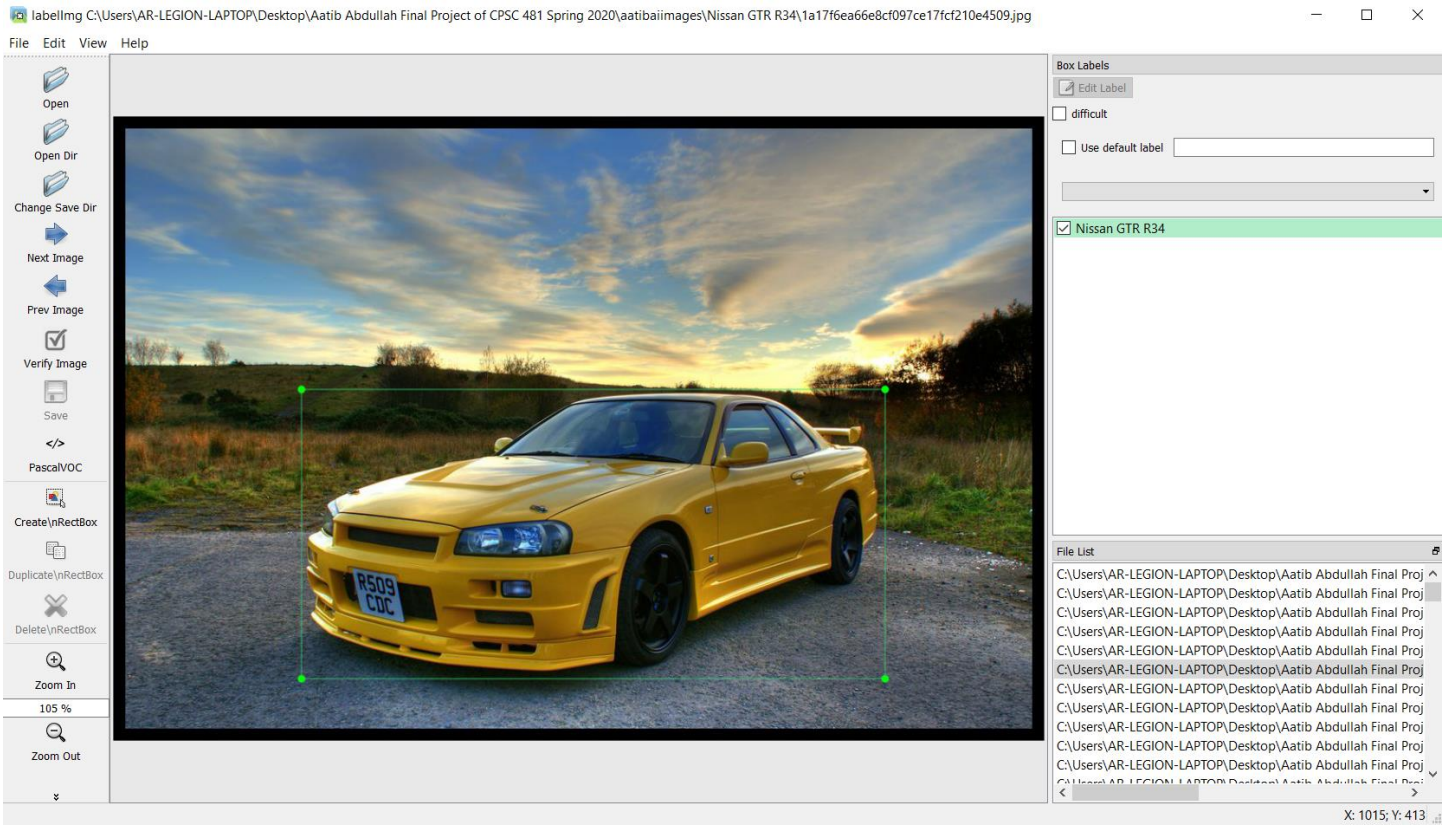
The image gets outputted and shows a box around the car where it shows a percentage rate of what is the make and model. However, if the percentage is high enough, we would consider it that make and model.

```
For imagepath in testimage:
    Load image into numpyarrray
    Numpyarray will be imagenp
    Output = process(imagenp, detection_graph) # this is where the Detection happens
    Imagefordisplay = Putvisualson.(Output.returnImage())
    # the Image would have a box around the object showing what car it is and the percentage of certaintiy which is
#   that car
    Display(Imagefordisplay)
```

C. Scripts and Programs used

1) LabelImg for Training

At mentioned in the Implementation Summary, Label IMG is or Label IMG, where one can drag a box on the image where the car is, and label it as such. It will use that portion of the image to focus on when training them. This aspect made our training robust and more accurate, with the assistance of Label IMG. This will help training the model and reduce the effect of viewpoint variation relative to the camera's view. This program comes from the link at https://github.com/tzutalin/labelImg and the program is in “labelImg directory” [2]. You can also just say "git clone https://github.com/tzutalin/labelImg". Inside its directory there is a script "lablelImg.py", run it and you will activate the program. You should see a PyQT5 style GUI, from that application open the directory containing your photos. Select your first image and click "Create/nRectbox" and highlight the area in the image containing your custom class. The image below is how the application looks like:



2) Converting xml to csv files

The images the system trains on are csv files that where converted from xml files. The xml files are images that have an area of the photo selected where that area is used for training and testing the system before processing a photo or video frame [2]. The reason why it was converted to a csv file is so that the software could read the data easily and processes it. We would use a script call xml_to_csv.py [3]. From line 28 to 33 is the most crucial part with defines the directory where our we put our converted csv files.

```
def main():
    for directory in ['train','test']:
        image_path = os.path.join(os.getcwd(), 'images/{}'.format(directory))
        xml_df = xml_to_csv(image_path)
        xml_df.to_csv('data/{}_labels.csv'.format(directory), index=None)
        print('Successfully converted xml to csv.')
```

With this code, we would need a directory of this organization for this script to work.

```
object-detection
├── data
│   ├── Test_labels.csv
│   └── Train_labels.csv
├── images
│   ├── Test
│   │   ├── Nissan_GTR_R34.jpg
│   │   ├── Nissan_GTR_R34.xml
│   │   |___ ….
│   └── Train
│       ├── Porsche_911_992.jpg
```

```
│       ├── Porsche_911_992.xml
│       └── …
│
│
├── training
└── xml_to_csv.py
```

4) Generating TFRecords from CSV data

After converting the XML (eXtensivle Markup Langage) files to CSV(comma separated value) files, we will then convert the data from the CSV files into a TFRecords file. The TFRecords file is a file in a format which is easier for storing a sequence of binary records that contain the strings. Generating the TF records file require to define the value of the Row_Labels which are the labels made on the photos. We run generate_tfrecord.py (from [4]) to get our binary data so we can use the product of this code for training (using train.py which is defined in the training algorithm section above). We make TFRecords for both Train and Test files. Here is what I did:

You set up the row label values in the Generate TFRecords file so we can record the labels that were made during the creation of the XML files.

```python
# TO-DO replace this with label map
def class_text_to_int(row_label):
    if row_label == 'Nissan GTR R34':
        return 1
    if row_label == 'Nissan GTR R35':
        return 2
    if row_label == 'Porsche 911 992':
        return 3
    else:
        None
```

We first run the setput.py script located at mdoels/research/directory to download missing dependencies.

Here are the commands we used to run generate_tfrecords.py:

```
1. python generate_tfrecord.py
   --csv_input=data/Train_labels.csv
   --output_path=data/train.record --image_dir=images/
2. python generate_tfrecord.py
   --csv_input=data/Test_labels.csv
   --output_path=data/test.record --image_dir=images/
```

Now we will use the Train our model.

5) Training our Model

We grab (copy it) the *ssd_mobilenet_v1_pets.config* from *models/research/object_detection/sameples/configs* directory and paste it in the object-detection directory. From the object-detection directory, open a command line (I used command-line bash application on Windows to unzip the tar.gz folder) and run command:

```
   wget
http://download.tensorflow.org/models/object_detection/ssd_mobilenet_
v1_coco_11_06_2017.tar.gz
```

Then run command:

```
tar -xvzf ssd_mobilenet_v1_coco_11_06_2017.tar.gz
```

If I have forgot to mention to create a *training* directory in the object-detection, please do so. In the *training* create file “object-detection.pbtxt”, and type in it the following statements:

```
item {
    id: 1
    name: 'Nissan GTR R34'
}
item {
    id: 2
    name: 'Nissan GTR R35'
}
item {
    id: 3
    name: 'Porsche 911 992'
}
```

And place *ssd_mobilenet_v1_pets.config* in the *training* directory. I recommend copying and pasting files in a new directory (I am calling my new directory "CAR_IDENTIFIER") with you current (object-detection) directory. Now change the high-lighted sections of the config file (or you could just copy and paste it cause finding where to edit can be a pain). The following is my current *ssd_mobilenet_v1_pets.config* file.

```
# SSD with Mobilenet v1, configured for Oxford-IIIT Pets Dataset.
# Users should configure the fine_tune_checkpoint field in the train config as
# well as the label_map_path and input_path fields in the train_input_reader and
# eval_input_reader. Search for "PATH_TO_BE_CONFIGURED" to find the fields that
# should be configured.

model {
  ssd {
    num_classes: 3 #number of custom classes
    box_coder {
      faster_rcnn_box_coder {
        y_scale: 10.0
        x_scale: 10.0
        height_scale: 5.0
        width_scale: 5.0
      }
    }
    matcher {
      argmax_matcher {
        matched_threshold: 0.5
        unmatched_threshold: 0.5
        ignore_thresholds: false
        negatives_lower_than_unmatched: true
        force_match_for_each_row: true
      }
    }
    similarity_calculator {
      iou_similarity {
      }
    }
    anchor_generator {
      ssd_anchor_generator {
        num_layers: 6
        min_scale: 0.2
```

```
    max_scale: 0.95
    aspect_ratios: 1.0
    aspect_ratios: 2.0
    aspect_ratios: 0.5
    aspect_ratios: 3.0
    aspect_ratios: 0.3333
  }
}
image_resizer {
  fixed_shape_resizer {
    height: 300
    width: 300
  }
}
box_predictor {
  convolutional_box_predictor {
    min_depth: 0
    max_depth: 0
    num_layers_before_predictor: 0
    use_dropout: false
    dropout_keep_probability: 0.8
    kernel_size: 1
    box_code_size: 4
    apply_sigmoid_to_scores: false
    conv_hyperparams {
      activation: RELU_6,
      regularizer {
        l2_regularizer {
          weight: 0.00004
        }
      }
      initializer {
        truncated_normal_initializer {
          stddev: 0.03
          mean: 0.0
        }
      }
      batch_norm {
        train: true,
        scale: true,
        center: true,
        decay: 0.9997,
        epsilon: 0.001,
      }
    }
  }
}
feature_extractor {
  type: 'ssd_mobilenet_v1'
  min_depth: 16
  depth_multiplier: 1.0
  conv_hyperparams {
    activation: RELU_6,
    regularizer {
      l2_regularizer {
        weight: 0.00004
      }
    }
    initializer {
```

```
          truncated_normal_initializer {
            stddev: 0.03
            mean: 0.0
          }
        }
        batch_norm {
          train: true,
          scale: true,
          center: true,
          decay: 0.9997,
          epsilon: 0.001,
        }
      }
    }
    loss {
      classification_loss {
        weighted_sigmoid {
        }
      }
      localization_loss {
        weighted_smooth_l1 {
        }
      }
      hard_example_miner {
        num_hard_examples: 3000
        iou_threshold: 0.99
        loss_type: CLASSIFICATION
        max_negatives_per_positive: 3
        min_negatives_per_image: 0
      }
      classification_weight: 1.0
      localization_weight: 1.0
    }
    normalize_loss_by_num_matches: true
    post_processing {
      batch_non_max_suppression {
        score_threshold: 1e-8
        iou_threshold: 0.6
        max_detections_per_class: 100
        max_total_detections: 100
      }
      score_converter: SIGMOID
    }
  }
}

train_config: {
  batch_size: 24
  optimizer {
    rms_prop_optimizer: {
      learning_rate: {
        exponential_decay_learning_rate {
          initial_learning_rate: 0.004
          decay_steps: 800720
          decay_factor: 0.95
        }
      }
      momentum_optimizer_value: 0.9
      decay: 0.9
```

```
      epsilon: 1.0
    }
  }
  fine_tune_checkpoint: "CAR_IDENTIFIER/ssd_mobilenet_v1_coco_11_06_2017/model.ckpt"
  from_detection_checkpoint: true
  load_all_detection_checkpoint_vars: true
  # Note: The below line limits the training process to 200K steps, which we
  # empirically found to be sufficient enough to train the pets dataset. This
  # effectively bypasses the learning rate schedule (the learning rate will
  # never decay). Remove the below line to train indefinitely.
  num_steps: 200000
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
    ssd_random_crop {
    }
  }
}

train_input_reader: {
  tf_record_input_reader {
    input_path: "CAR_IDENTIFIER/data/Train.record"
  }
  label_map_path: " CAR_IDENTIFIER/training/object-detection.pbtxt"
}

eval_config: {
  metrics_set: "coco_detection_metrics"
  num_examples: 1100 #Number of images for taining, should have been lowered to 100
}

eval_input_reader: {
  tf_record_input_reader {
    input_path: " CAR_IDENTIFIER/data/Test.record"
  }
  label_map_path: " CAR_IDENTIFIER/training/object-detection.pbtxt"
  shuffle: false
  num_readers: 1
}
```

Now copy the directories *data*, *images*, *ssd_mobilenet_v1_coco_11_06_2017*, and *training* and past them in *CAR_IDENTIFIER* directory (you name your directory whatever you want). We will copy this directory and paste it in the *models/research/object_detection* directory.

```
CAR_IDENTIFIER
├── data
│   ├── Test.record
│   ├── Test_labels.csv
│   ├── Train.record
│   └── Train_labels.csv
├── images
│   |_Sample Imasges of cars
├── ssd_mobilenet_v1_coco_11_06_2017
│   ├── frozen_inference_graph.pb
│   ├── graph.pbtxt
│   ├── model.ckpt.data-00000-of-00001
│   ├── model.ckpt.index
```

```
│   └── model.ckpt.meta
└── training
    ├── object-detection.pbtxt
    └── ssd_mobilenet_v1_pets.config
```

Grab the *train.py* script from the *models/research/object_detection/legacy* directory and paste it in the *models/research/object_detection*. Run the command:

Python train.py –logtostderr –train_dir=training/
--pipeline_config_path=training/ssd_mobilenet_v1_pets.config

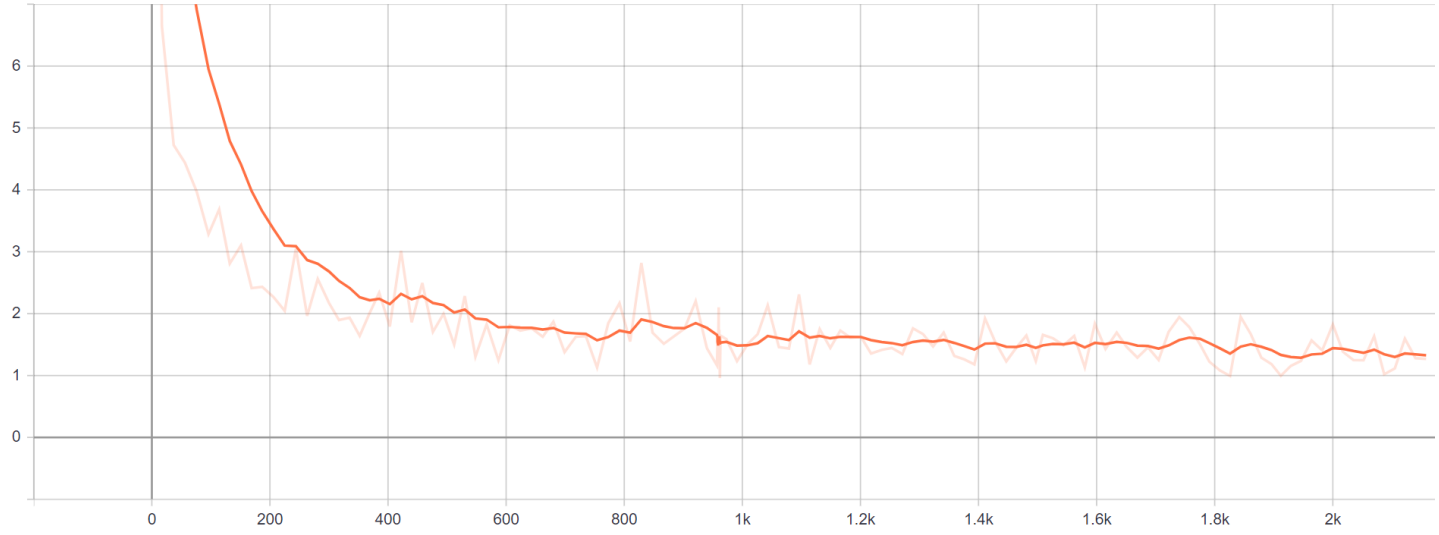
It will output some warnings, but it should start outputting steps and losses like this snippet bellow:

```
I0404 03:04:27.685195 15176 supervisor.py:1050] Recording summary at
step 15.
INFO:tensorflow:global step 16: loss = 8.7406 (19.563 sec/step)
I0404 03:04:27.732058  2216 learning.py:507] global step 16: loss =
8.7406 (19.563 sec/step)
INFO:tensorflow:global step 17: loss = 8.6451 (6.526 sec/step)
I0404 03:04:34.257843  2216 learning.py:507] global step 17: loss =
8.6451 (6.526 sec/step)
INFO:tensorflow:global step 18: loss = 8.4099 (11.514 sec/step)
I0404 03:04:45.787036  2216 learning.py:507] global step 18: loss =
8.4099 (11.514 sec/step)
```

Try to get under Total Loss of 1, if training your model takes too long you can sacrifice a bit by letting the loss come around the average of 1 but must be less than 2. For my case I was able to get under 2 but it was above 1 during the training process. You can see a graph representation of you training by running the command while in the *models/research/object_detection* directory:

tensorboard --logdir CAR_IDENTIFIER\training

You will get a graph similar to the image below.



As one can see, from step 1 to 2158 steps it rarely touches slightly above 1. Had to settle for less since the model was already training for 5 hours. The lower the total loss value, the higher the accuracy in detection. More Images can lower the loss value, which when exposed to more data means more accuracy. I had more than a thousand pictures, but to have a loss less than one I would need a few more thousand pictures. To stop training, I press "Ctrl + C".

6) Exporting and Testing our Model

To export the model, first need to grab (copy) *export_inference_graph.py* from *models\research\build\lib\object_detection* directory and paste it in *models\research\object_detection*. Then run the command:

```
python export_inference_graph.py --input_type image_tensor
--pipeline_config_path CAR_IDENTIFIER /training/ssd_mobilenet_v1_pets.config
--trained_checkpoint_prefix CAR_IDENTIFIER /training/model.ckpt-2158
--output_directory CAR_IDENTIFIER/car_identifier_graph
```

Now we have a model name "car_identifier_graph".CAR_IDENFITIER is the main part of the program. The script to run this program would be to create Jupyter Notebook file which I named "Testing_Car__Identifier.ipynb." The algorithms Aof this script is in pseudocode for detection and Outputting the results as an Image with a percent value on it.

D. Implementation Challenge and Learning Experience: To get more Images for Better Accuracy or Decrease Number of Examples

The amount of images you have determines the accuracy of the system. Having more images in the range of thousands can help reduce the Total loss and make it more accurate. Finding a lot of images can be a challenge because it is very lengthy process. The solution is to find about a thousand images and create new images out of them like flipping a photo form left to right orientation. One can also find more images. If you do not have enough Images, you can decrease “num_examples”(number of examples variable) in “ssd_mobilenet_v1_pets.config” in the eval_config function (evaluation configuration function). The mistake made that I did not edit the number of examples (like about 100) and kept it at the original 1100 size. It made the system less accurate, so if one does not have enough the system would make mistakes in detection. Decreasing the number of examples in the evaluation configuration function can make the detection system more accurate if running on a low number of images. The system would train and test on that amount every step in the training process, although it would take longer because it would look at each of the images for detail.

Working Example:

We have a small-sized that speed through a red light, and we have captured the footage of it. The footage will be processed by our software to see what model it is by looking at its body. We do not know what make and model it is since it was in the afternoon which the sun was setting, affecting the light. So we have to rely on the body of the car. The color of the car is white and using that we can detect was model it is. The software has already trained with a sample photos of different models from variety of brands. By using the layered neural networks, the image would be processed. Since the neural networks are trained, it would be able to use that experience to determine what is in that photo. It turns out that it was a Nissan Altima that broke the red light. This paragraph is meant to show how the system would function.

In Operation:

We have applied our system to seven photos. We have three cars that we want to detect for which is the Nissan GTR R34, Porsche 911 992, and the Nissan GTR R35. In the photo, these cars will have varying angles and colors to see if it can detect even from the backside. These photos are in the Images directory in the

CAR_IDENTIFIER folder where the models are. We run the “Testing_Car_Identifier” file in Jupyter Notebook to see if our system works. Here is the results of running the program:


Porsche 911 992: 99%


Porsche 911 992: 97%

Conclusions:

This system will be in operation by the Beginning of December in the year of 2020. This project would help make identification of cars more easier for Traffic controllers to detect. The status of the project is that it has not been implemented yet, but it will be worked on after the release of this document. The Traffic controllers will be able to get information about a vehicle by looking at the footage of the car once this system would be in effect. This system would be able to assist Law Enforcement and Road Traffic Management to do their Jobs.

Work Cited:

[1] T. O'Sullivan, *CarGurus*. [Online]. Available: https://www.cargurus.com/Cars/articles/understanding_vehicle_generations. [Accessed: 26-Oct-2020].

[2] *Python Programming Tutorials*, 25-Aug-2017. [Online]. Available: https://pythonprogramming.net/custom-objects-tracking-tensorflow-object-detection-api-tutorial/. [Accessed: 26-Oct-2020].

[3] Datitran. (2018, August 21). Datitran/raccoon_dataset. Retrieved December 05, 2020, from https://github.com/datitran/raccoon_dataset/blob/master/generate_tfrecord.py

[4] Datitran. (n.d.). Datitran/raccoon_dataset. Retrieved December 05, 2020, from https://github.com/datitran/raccoon_dataset/tree/386a8f4f1064ea0fe90cfac8644e0dba48f0387b