Brainwave Project: Image Processing

CPSC 499

Spring 2020

Written by:
Aatib Abdullah, Aarib Abdullah
Advisors:
Dr. Doina Bein

Abstract:

The emotion of a human is a vital part of whether or not an automobile accident would occur. There has to be a system that would offer a remedy to this problem. This group project, supervised by Dr. Doina Bein, is part of a broader project that includes image processing, Lidar Data processing, and Brain wave Data processing. We propose to work on the Image Processing part. We, as in Aarib and Aatib Abdullah, will be performing independent study on this section of the project and would develop a program that identifies restaurants or hotel signs. The Images would be collected and filtered from the Internet. The Images would be the dataset that would be used for training. They would be recognized as either restaurants or hotels. These signs in the Images would be classified as restaurants or hotel brands, such as McDonald's or Burger King. These images would be processed using several Image Processing Techniques to recognize whether they are hotels or restaurants and if yes, they could become a possible stop for a driver. The system would help the driver get its needs fulfilled and prevent the likelihood of accidents.

Purpose:

Building an Image Processor:

The purpose is to develop an Image Processing system that would recognize the road signs that lead to restaurants and hotels. When the driver comes across the sign, he would have an option to pursue those restaurant and hotel places that the system recognizes. This system comes into action when the driver feels like it needs to eat or feels tired. The broader Brainwave Processing Device will measure the brainwave of the driver, and it will activate our system if required.

Learn and Improve Skills:

One of the purposes of the project is to learn about how to use Machine learning tools to implement them in our project. It gives us the opportunity to improve our programming skills and give us the opportunity to expand our knowledge of machine learning and artificial intelligence. We use tools like Python, which is a dynamic programming language in which we get to learn how to build an image processor with this language. With Python, we used the TensorFlow library. The library gives us models that we can use for the layers of the neural networks. This will provide us with the opportunity to be exposed to and use CNN (Convolutional Neural Network).

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 the Instructions)

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.

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 (restaurant/hotel signs) 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. What if there was a signboard that shows multiple fast food logos? The Image Classification approach would try to classify the entire sign into one whole class, when there are actually two classes. In Object Detection, it would identify the different logos, and classify them in separate classes. So it is better to implement an Object-Detection oriented approach, where we would treat restaurant/hotel signs 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.

Label IMG for Training:

Label IMG is or Label IMG, where one can drag a box on the image where the food or hotel sign 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.

Outcome:

Image Processor:

The Image Processor that can classify signs of restaurant brands it was trained for, which are Burger King, McDonald's, KFC, In-N-Out. The Image Processor is not able to recognize signs that are under massive lighting like sun or reflection from a cloudy sky. Alteration of the light in the environment could affect the image Processer's ability to recognize a sign. The only way to fix this is by a more diverse training data set where there are varying brightness and darkness. Some restaurants and hotels have not changed into new signs of their company, so we have to add those signs to the training set.

Learning:

Throughout the Project, We have learned many things and learned how to approach our problems. Which includes the following:

- We found that there is a difference between image classification and object detection.
- We used tutorials to help us use TensorFlow Object-Detection API, with a model that is capable of fast real-time video stream but trades off from accuracy. To go around this, we would use more data/images to train the model.
- Environmental factors such as brightness, darkness, and even the potential target logo being in front of the sun can through off the object detector. There is a strong possibility that data augmentation can buff the model. For every image, create a darker and brighter version of the image while still recognizable by the human eye.
- We followed with the idea buy rather than build principle, but in this case it did not cost us. We trained a model which can be ready for real-time object-detection in the future.

Instructions for Creating Custom Object Detection Model

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

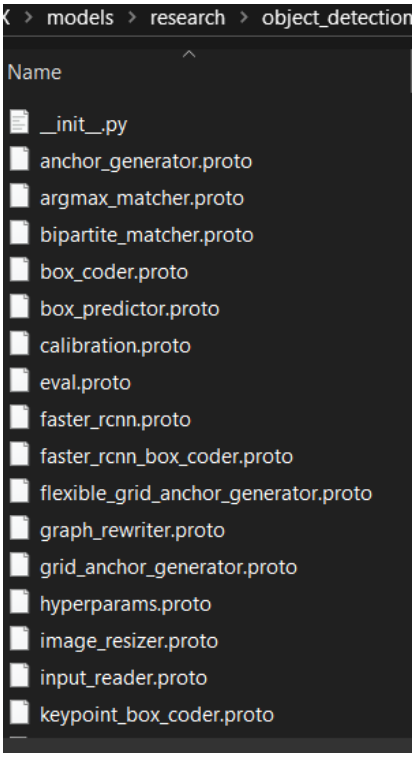
Once that is done, please go to the link for the repository holding the Object-Detection API https://github.com/tensorflow/models. We can also simply use git. Then extract the file.

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.



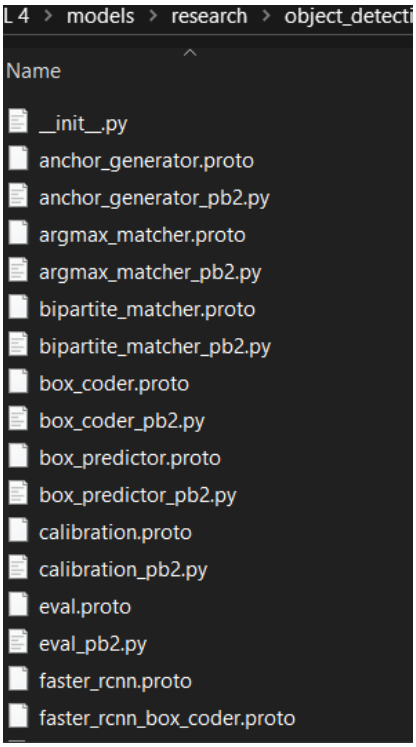


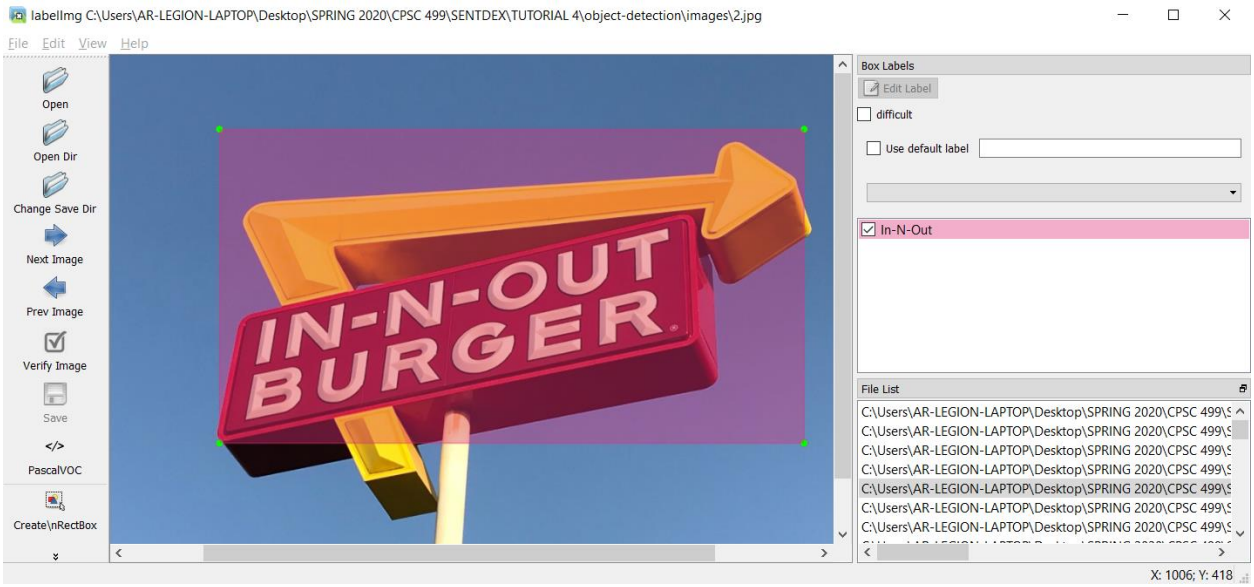
Image on the left is before the protoc program executed, the image on the right is after the protoc program executed.

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.

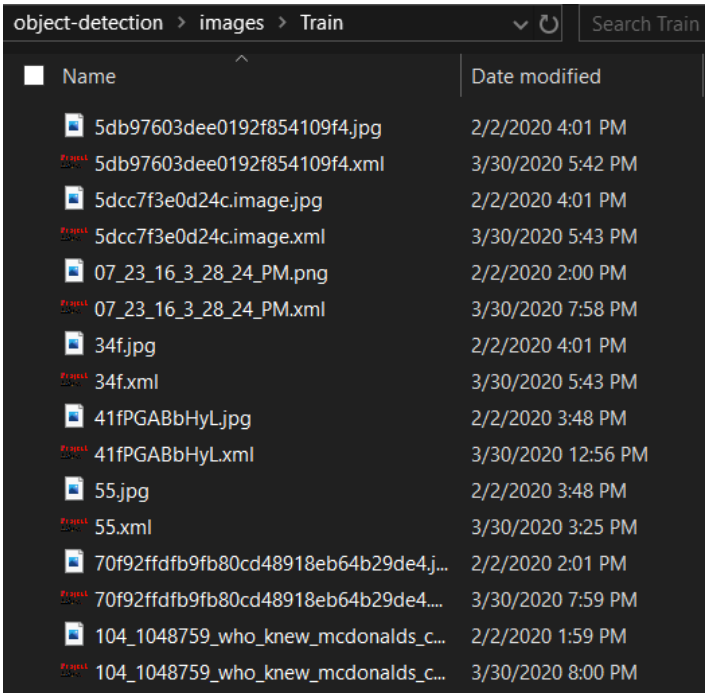
Labeling and Tracking Custom Objects

Assuming you have already collected your images per each class (an image can contain one or more classes, EX: highway signboard), now we will begin labeling the classes in each image. Download the LabelImg program script at https://github.com/tzutalin/labelImg. You can also just say "git clone https://github.com/tzutalin/labelImg". Inside its directory there is a script "lablelImg.py", run it. 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. It should look like the following image.



This process will be done for every photo for every class. The labeling program will generate an XML file containing the images' filename, their dimensions and the coordinates of where the custom class(es) are in the images.

Once done with labeling the custom classes in the images, separate the images with their corresponding XML files into Train and a Test directory. The rule I followed is the 90% – 10%. Now put them in the *object-detection/images/Train* and *object-detection/images/Test* directories. The directory containing the files should now look like the following image for both *Test* and *Train*.



Creating Generating CSV data from XML Files

Now we have XML data which tells where each custom class are in the images. We then need to use the XML files and generate two CSV files, one holding data for training images and the other for testing images. Lets grab a script called *xml_to_csv.py* from https://github.com/datitran/raccoon_dataset/tree/386a8f4f1064ea0fe90cfac8644e0dba48f0387b.

Put this script in the *object-detection* directory. Open the file and change the following section of the code (line 28 – 32) from (the first following image) to the (second following image).

```
28  def main():
29      image_path = os.path.join(os.getcwd(), 'annotations')
30      xml_df = xml_to_csv(image_path)
31      xml_df.to_csv('raccoon_labels.csv', index=None)
32      print('Successfully converted xml to csv.')
```

```
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.')
```

Then run the script with command "python xml_to_csv.py".

Just making sure, your file structure should look like this (not the directory in models/research/object_detection):

```
object-detection
├── data
│   ├── Test_labels.csv
│   └── Train_labels.csv
├── images
│   ├── Test
│   │   ├── 00OBESITY_ghana_street_articleLarge.jpg
│   │   ├── 00OBESITY_ghana_street_articleLarge.xml
│   │   ├── 0123_biz_wire_app.jpg
│   │   ├── 0123_biz_wire_app.xml
│   │   └── ...
│   └── Train
│       ├── 0711_kfc_jpg.jpg
│       ├── 0711_kfc_jpg.xml
│       ├── 07_23_16_3_28_24_PM.png
│       ├── 07_23_16_3_28_24_PM.xml
│       └── …
├── training
└── xml_to_csv.py
```

Generating TFRecords from CSV Data

Download the script *generate_tfrecord.py* from https://github.com/datitran/raccoon_dataset/blob/master/generate_tfrecord.py. The following section of code will change from the following first code snippet to the second code snippet.

```
31  def class_text_to_int(row_label):
32      if row_label == 'raccoon':
33          return 1
34      else:
35          None
```

```
def class_text_to_int(row_label): #Ke
    # if row_label == 'raccoon':
    #     return 1
    # else:
    #     None
    if row_label == 'McDonalds':
        return 1
    elif row_label == 'Burger King':
        return 2
    elif row_label == 'KFC':
        return 3
    elif row_label == 'In-N-Out':
        return 4
    else:
        None
```

Please make sure the *row_label* values are the exact values as the custom classes mentioned during labeling (look at *Labeling and Tracking Custom Objects* section of this report).

We then need run the script *setup.py* located at models/research/ directory to download any missing dependencies.

Now to generate TFRecords for training and testing data, run the following commands while at the object-detection directory:

```
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/
```

You now have generated files train.record and test.record.

Finally, Training our Model

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 Ubuntu 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: 'McDonalds'
}
item {
    id: 2
    name: 'Burger King'
}
item {
    id: 3
    name: 'KFC'
}
item {
    id: 4
    name: 'In-N-Out'
}
```

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 "GUTS_2") 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: 4 #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:
"GUTS_2/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: "GUTS_2/data/Train.record"
  }
  label_map_path: "GUTS_2/training/object-detection.pbtxt"
}

eval_config: {
  metrics_set: "coco_detection_metrics"
  num_examples: 26 #Number of test images
}

eval_input_reader: {
  tf_record_input_reader {
    input_path: "GUTS_2/data/Test.record"
  }
  label_map_path: "GUTS_2/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 *GUTS_2* directory (you name your directory whatever you want). We will copy this directory and paste it in the *models/research/object_detection* directory.

```
GUTS_2
├── data
│   ├── Test.record
│   ├── Test_labels.csv
│   ├── Train.record
│   └── Train_labels.csv
├── images
│   ├── Test
│   │   ├── 00OBESITY_ghana_street_articleLarge.jpg
│   │   ├── 00OBESITY_ghana_street_articleLarge.xml
│   │   └── …
│   ├── Train
│   │   ├── 0711_kfc_jpg.jpg
│   │   ├── 0711_kfc_jpg.xml
│   │   └── …
├── 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

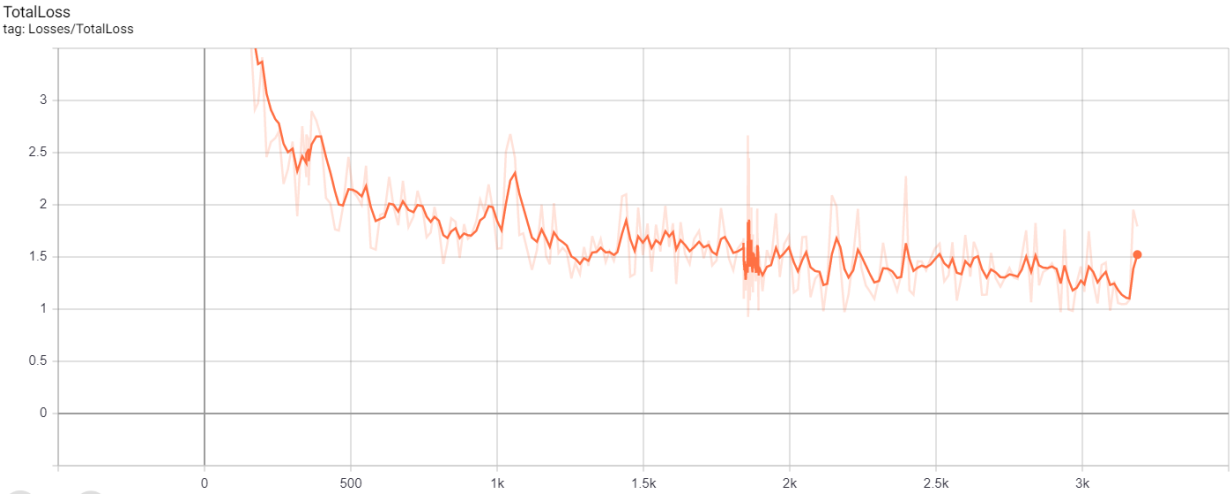
It will output some warnings, but it should start outputting steps and losses like this snippet bellow:

```
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 TotalLoss 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. You can see a graph representation of you training by running the command while in the *models/research/object_detection* directory:

tensorboard --logdir GUTS_2\training

You will get a graph similar to the image below.



As one can see, from step 1 to 3,188,000 steps it rarely touches below 1. Had to settle for less since the model was already training for 10 hours. This would be less painful if added more image data. But wanted to try to only use images which contain the perspective of a car looking on the road which have food-chain logos/signs near it.

To stop training, just press "Ctrl + C".

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 GUTS_2/training/ssd_mobilenet_v1_pets.config
--trained_checkpoint_prefix GUTS_2/training/model.ckpt-3187
--output_directory GUTS_2/Food_chainz
```

You should point the *trained_checkpoint_prefix* where the step number of the model checkpoint (ckpt) is highest in the *training* directory. Now we have a model name "Food_chainz". Please look at the Jupyter Notebook "Testing_Food_Chainz_recognition.ipynb" for how to read frozen_inference_graph.pb into a TensorFlow model in memory and use *object-detection.pbtxt* as label maps to give value regarding which class exists in the image.

The Jupyter Notebook is coded to output the test images with the labels on an area of the image where a class exists per the model. That's about it.