WHITE PAPER

Transportation
Artificial Intelligence
Intel AI® Builders

Traffic Light Detection Using the TensorFlow* Object Detection API

Table of Contents

Abstract

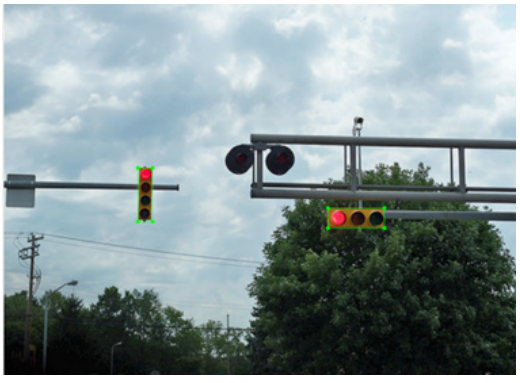
This case study evaluates the ability of the TensorFlow* Object Detection API to solve a real-time problem such as traffic light detection. The experiment uses the Microsoft Common Objects in Context (COCO) pre-trained model called Single Shot Multibox Detector MobileNet from the TensorFlow Zoo for transfer learning. Intel® Xeon® processor-based machines were used for the study. At the end of this experiment, we obtained an accurate model that was able to identify the traffic signals at more than 90 percent accuracy.

Introduction

With the advancements in technology, there has been a rapid increase in the development of autonomous cars or smart cars. Accurate detection and recognition of traffic lights is a crucial part in the development of such cars. The concept involves enabling autonomous cars to automatically detect traffic lights using the least amount of human interaction. Automating the process of traffic light detection in cars would also help to reduce accidents.

Traditional approaches in machine learning for traffic light detection and classification are being replaced by deep learning methods to provide state-of-the-art results. However, these methods create various challenges. For example, the distortion or variation in images due to orientation, illumination, and speed fluctuation of vehicles could result in false recognition.

The experiment was implemented using transfer learning of the Microsoft Common Objects in Context (COCO) pre-trained model called Single Shot Multibox Detector (SSD) with MobileNet. A subset of the ImageNet* dataset, which contains traffic lights, was used for further training to improve the performance. For this particular experiment, the entire training and the inferencing was done on an Intel® Xeon® processor.



Hardware Details

The hardware configuration of Intel Xeon® machine is as follows:

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	88
On-line CPU(s) list:	0-87
Thread(s) per core:	2
Core(s) per socket:	22
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	79
Model name::	Intel(R) Xeon (R) CPU E5-2699 v4 @ 2.20 GHz
Stepping:	1
CPU MHz:	1200.117
CPU max MHz:	3600
CPU min MHz:	1200
BogoMIPS:	4391.09
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	56320K
NUMA node0 CPU(s):	0-21,44-65
NUMA node1 CPU(s):	22-43,66-87

Table 1: Intel® Xeon® processor configuration.

Software Configuration

The development of this use case had the following dependencies as shown in Table 2.

Library	Version
TensorFlow*	1.4.0 (built from source)
Python*	3 or later
Operating system	CentOS* 7.3.1
Protobuf	2.6
Pillow	1.0
Lxml	4.1.1
Matplotlib	2.1.0
MoviePy	0.2
GCC* (GNU Compiler Collection*)	6+

Table 2: Software configuration

Installation

Building and Installing TensorFlow Optimized for Intel® Architecture

TensorFlow can be installed and used with several combinations of development tools and libraries on a variety of platforms. The following are the steps to build and install TensorFlow optimized for Intel® architecture[1] with the Intel® Math Kernel Library 2017 on Ubuntu*-based systems.

```
git clone https://github.com/tensorflow/tensorflow
cd tensorflow
git checkout r1.4
wget https://github.com/bazelbuild/bazel/releases/download/0.7.0/bazel-0.7.0-installer-linux-x86_64.sh
wget --no-check-certificate -c --header "Cookie: oraclelicense=accept-securebackupcookie" http://download.
oracle.com/otn-pub/java/jdk/8u151-b12/e758a0de34e24606bca991d704f6dcbf/jdk-8u151-linux-x64.tar.gz
export PATH=/opt/intel/intelpython3.5/bin/:${PATH}
conda create -n tensorflow python=3.5
source activate tensorflow
tar -zxvf jdk-8u151-linux-x64.tar.gz
export JAVA_HOME=$WRKDIR/jdk1.8.0_151
export PATH=$JAVA_HOME/bin:$PATH
export PATH=$PATH:$JAVA_HOME/bin:/home/intel-user3/bazel/output
chmod 755 bazel-0.7.0-installer-linux-x86_64
./bazel-0.7.0-installer-linux-x86_64 --user --prefix=~/bazel
./configure
bazel build --config=mkl -c opt //tensorflow/tools/pip_package:build_pip_package
bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
sudo pip install /tmp/tensorflow_pkg/tensorflow-1.4.0
```

Installing LabelImg

Download the latest version of LabelImg, an annotation tool for Microsoft Windows*[2]. Extract the zip file, and then rename the folder as LabelImg.

Solution Design

The solution was implemented with the TensorFlow Object Detection API using Intel architecture. The detection pipeline is given below.

Traffic detection pipeline

```
Algorithm 1: Detection Pipeline
boxAssigned← false
while true do
        f ←nextFrame
        while boxAssigned == false do
                 InvokeDetection(f)
        if Bounding Box is detected then
                boxAssigned ← true
                class ← identfiedClass
                if class is Trafficlight then
                    drawBoundingBox
                end if
        end if
end while
```

Why choose TensorFlow Object Detection API?

TensorFlow's Object Detection API is a powerful tool that makes it easy to construct, train, and deploy object detection models[3]. In most of the cases, training an entire convolutional network from scratch is time consuming and requires large datasets. This problem can be solved by using the advantage of transfer learning with a pre-trained model using the TensorFlow API. Before getting into the technical details of implementing the API, let's discuss the concept of transfer learning.

Transfer learning is a research problem in machine learning that focuses on storing the knowledge gained from solving one problem and applying it to a different but related problem. Transfer learning can be applied three major ways[4]:

Convolutional neural network (ConvNet) as a fixed feature extractor: In this method the last fully connected layer of a ConvNet is removed, and the rest of the ConvNet is treated as a fixed feature extractor for the new dataset.

Fine-tuning the ConvNet: This method is similar to the previous method, but the difference is that the weights of the pre-trained network are fine-tuned by continuing backpropagation.

Pre-trained models: Since modern ConvNets takes weeks to train from scratch, it is common to see people release their final ConvNet checkpoints for the benefit of others who can use the networks for fine-tuning. For example, TensorFlow Zoo[5] is one such place where people share their trained models/checkpoints.

In this experiment, we used a pre-trained model for the transfer learning. The advantage of using a pre-trained model is that instead of building the model from scratch, a model trained for a similar problem can be used as a starting point for training the network. Many pre-trained models are available. This experiment used the COCO pre-trained model/checkpoints SSD MobileNet from the TensorFlow Zoo. This model was used as an initialization checkpoint for training. The model was further trained with images of traffic lights from ImageNet. This fine-tuned model was used for inference.

Now let's look at how to implement the solution. The TensorFlow Object Detection API has a series of steps to follow, as shown in Figure 1.

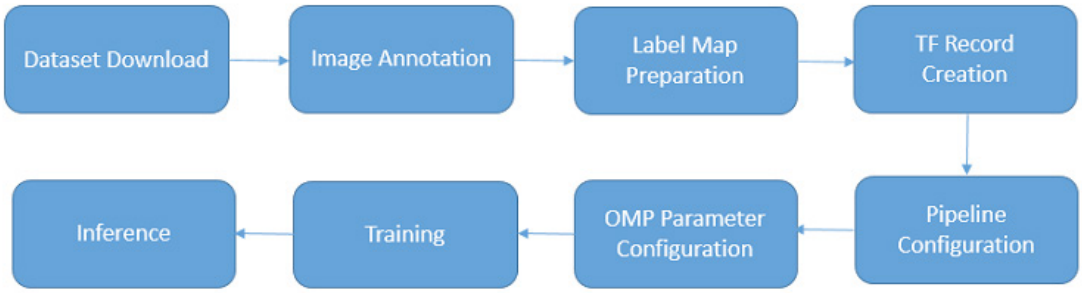


Figure 1: Solution design

1. Dataset download

The dataset for fine-tuning the pre-trained model was prepared using over 600 traffic light images from ImageNet[6]. The dataset contains over ten million URLS of images from various classes. The traffic light images were downloaded from the URLs and saved for annotation.

2. Image Annotation

1. Configuring the LabelImg tool. Before starting with the annotation of images, the classes for labelling needs to be defined in the `LabelImg/data/predefined_classes.txt` file. In this case, there's only one class which is `trafficlight`.
2. Launch labelimg.exe and then select the dataset folder by clicking the **OpenDir** icon on the left pane.
3. For each image that appears, draw a rectangular box across each traffic light by clicking the **Create RectBox** icon. These rectangular boxes are known as bounding boxes. Select the category **trafficlight** from the drop-down list that appears.
4. Repeat this process for every traffic light present in the image. Figure 2 shows an example of a completely annotated image.

Figure 2: Annotated image

Once the annotations for an image are completed, save the image to any folder.

The corresponding eXtensible Markup Language (XML) files will be generated for each image in the specified folder. XML files contain the coordinates of the bounding boxes, filename, category, and so on for each object within the image. These annotations are the ground truth boxes for comparison. Figure 3 represents the XML file of the corresponding image in Figure 2.

```
<?xml version="1.0"?>
- <annotation>
    <folder>New folder1</folder>
    <filename>1130297356_52720b15a7.jpg</filename>
    <path>  PATH OF THE IMAGE FILE  </path>
  - <source>
      <database>Unknown</database>
    </source>
  - <size>
      <width>500</width>
      <height>375</height>
      <depth>3</depth>
    </size>
    <segmented>0</segmented>
  - <object>
      <name>trafficlight</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>129</xmin>
        <ymin>162</ymin>
        <xmax>144</xmax>
        <ymax>216</ymax>
      </bndbox>
    </object>
  - <object>
      <name>trafficlight</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>312</xmin>
        <ymin>201</ymin>
        <xmax>369</xmax>
        <ymax>222</ymax>
      </bndbox>
    </object>
</annotation>
```

Figure 3: XML file structure

3. Label map preparation

Each dataset requires a label map associated with it, which defines a mapping from string class names to integer class IDs. Label maps should always start from ID 1.

As there is only one class, the label map for this experiment file has the following structure:

```
item {
    id: 1
    name: 'trafficlight'
}
```

4. TensorFlow records (TFRecords) generation

TensorFlow accepts inputs in a standard format called a TFRecord file, which is a simple record-oriented binary format. Eighty percent of the input data is used for training and 20 percent is used for testing. The split dataset of images and ground truth boxes are converted to train and test TFRecords. Here, the XML files are converted to csv, and then the TFRecords are created. Sample scripts for generation are available here.

5. Pipeline configuration

This section discusses the configuration of the hyperparameters, and the path to the model checkpoints, ft. records, and label map. The protosun files are used to configure the training process that has a few major configurations to be modified. A detailed explanation is given in Configuring the Object Detection Training Pipeline. The following are the major settings to be changed for the experiment.

- In the `model` config, the major setting to be changed is the `num_classes` that specifies the number of classes in the dataset.
- The train config is used to provide model parameters such as `batch_size`, `learning_rate` and `fine_tune_checkpoint`. `fine_tune_checkpoint` field is used to provide path to the pre-existing checkpoint.
- The `train_input_config` and `eval_input_config` fields are used to provide paths to the TFRecords and the label map for both train as well as test data.

Table 4 depicts the observations of hyperparameter tuning for various trials of `batch_size` and `learning_rate`.

6. OpenMP* (OMP) parameters configuration

There are various optimization parameters that can be configured to improve the system performance. The experiment was attempted with `OMP_NUM_THREADS` equal to 8. However the experiment could be tried with `OMP_NUM_THREADS` up to four less than the number of cores.

7. Training

The final task is to assemble all that has been configured so far and run the training job (see Figure 4). Once the optimization parameters like `OMP_NUM_THREADS`, `KMP_AFFININTY`, and the rest are set, the training file is executed. By default, the training job will continue to run until the user terminates it explicitly. The models will be saved at various checkpoints.

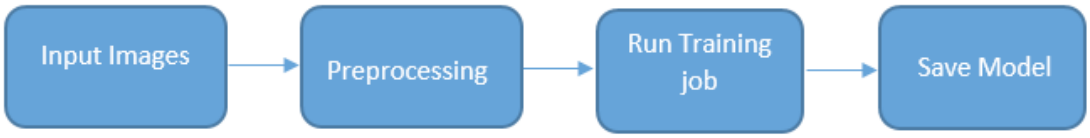


Figure 4: Training pipeline

8. Inference

The inferencing video was first converted into frames using MoviePy, a Python* module for video editing. These sets of frames are given to our model trained using transfer learning. After the frames pass through the Object Detection pipeline, the bounding boxes will be drawn on the detected frames. These frames are finally merged to form the inferred video (see Figure 5).

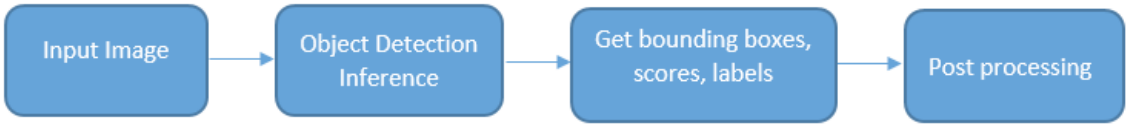


Figure 5: Inference pipeline

Experimental Results

The following detection (see Figures 6 and 7) was obtained when the inference use case was run on a sample YouTube* video available at: https://www.youtube.com/watch?v=BMYsRd7Qq0I

Figure 6: Raw frame

Figure 7: Inferenced frame

Conclusion and Future Work

From the results, we observed that the traffic lights were detected with a high level of accuracy. Future work involves parallel inferencing across multiple cores.

References

1. Build and install TensorFlow on Intel architecture:

https://software.intel.com/en-us/articles/build-and-install-tensorflow-on-intel-architecture

2. LabelImg

https://github.com/tzutalin/labelImg

3. TensorFlow Object Detection API

https://github.com/tensorflow/models/tree/master/research/object_detection

4. Transfer learning

http://cs231n.github.io/transfer-learning

5. TensorFlow detection model zoo

https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md

6. ImageNet

http://imagenet.stanford.edu/synset?wnid=n06874185