TensorFlow setup Documentation

Lyudmil Vladimirov

Sep 15, 2020

CONTENTS:

Important: This tutorial is intended for TensorFlow 1.14, which (at the time of writing this tutorial) is the latest stable version before TensorFlow 2.x.

Tensorflow 1.15 has also been released, but seems to be exhibiting instability issues.

A version for Tensorflow 1.9 can be found here.

A version for Tensorflow 2.x is in the making and a link will be added here when ready.

This is a step-by-step tutorial/guide to setting up and using TensorFlow's Object Detection API to perform, namely, object detection in images/video.

The software tools which we shall use throughout this tutorial are listed in the table below:

Target Software versions	
OS	Windows, Linux
Python	3.7
TensorFlow	1.14
CUDA Toolkit	10.0
CuDNN	7.6.5
Anaconda	Python 3.7 (Optional)

CHAPTER

ONE

INSTALLATION

1.1 General Remarks

- There are two different variations of TensorFlow that you might wish to install, depending on whether you would like TensorFlow to run on your CPU or GPU, namely *TensorFlow CPU* and *TensorFlow GPU*. I will proceed to document both and you can choose which one you wish to install.
- If you wish to install both TensorFlow variants on your machine, ideally you should install each variant under a different (virtual) environment. If you attempt to install both *TensorFlow CPU* and *TensorFlow GPU*, without making use of virtual environments, you will either end up failing, or when we later start running code there will always be an uncertainty as to which variant is being used to execute your code.
- To ensure that we have no package conflicts and/or that we can install several different versions/variants of TensorFlow (e.g. CPU and GPU), it is generally recommended to use a virtual environment of some sort. For the purposes of this tutorial we will be creating and managing our virtual environments using Anaconda, but you are welcome to use the virtual environment manager of your choice (e.g. virtualenv).

1.2 Install Anaconda Python 3.7 (Optional)

Although having Anaconda is not a requirement in order to install and use TensorFlow, I suggest doing so, due to it's intuitive way of managing packages and setting up new virtual environments. Anaconda is a pretty useful tool, not only for working with TensorFlow, but in general for anyone working in Python, so if you haven't had a chance to work with it, now is a good chance.

Windows

- Go to https://www.anaconda.com/download/
- Download Anaconda Python 3.7 version for Windows
- Run the downloaded executable (`.exe`) file to begin the installation. See here for more details.
- (Optional) In the next step, check the box "Add Anaconda to my PATH environment variable". This will make Anaconda your default Python distribution, which should ensure that you have the same default Python distribution across all editors.

Linux

- Go to https://www.anaconda.com/download/
- Download Anaconda Python 3.7 version for Linux
- Run the downloaded bash script (`.sh`) file to begin the installation. See here for more details.

- When prompted with the question "Do you wish the installer to prepend the Anaconda<2 or 3> install location to PATH in your /home/<user>/.bashrc ?", answer "Yes". If you enter "No", you must manually add the path to Anaconda or conda will not work.

1.3 TensorFlow Installation

As mentioned in the Remarks section, there exist two generic variants of TensorFlow, which utilise different hardware on your computer to run their computationally heavy Machine Learning algorithms.

1. The simplest to install, but also in most cases the slowest in terms of performance, is *TensorFlow CPU*, which runs directly on the CPU of your machine.
2. Alternatively, if you own a (compatible) Nvidia graphics card, you can take advantage of the available CUDA cores to speed up the computations performed by TensorFlow, in which case you should follow the guidelines for installing *TensorFlow GPU*.

1.3.1 TensorFlow CPU

Getting setup with an installation of TensorFlow CPU can be done in 3 simple steps.

Important: The term *Terminal* will be used to refer to the Terminal of your choice (e.g. Command Prompt, Powershell, etc.)

1.3.1.1 Create a new Conda virtual environment (Optional)

- Open a new *Terminal* window
- Type the following command:

```
conda create -n tensorflow_cpu pip python=3.7
```

- The above will create a new virtual environment with name `tensorflow_cpu`
- Now lets activate the newly created virtual environment by running the following in the *Terminal* window:

```
activate tensorflow_cpu
```

Once you have activated your virtual environment, the name of the environment should be displayed within brackets at the beggining of your cmd path specifier, e.g.:

```
(tensorflow_cpu) C:\Users\sglvladi>
```

1.3.1.2 Install TensorFlow CPU for Python

- Open a new *Terminal* window and activate the *tensorflow_cpu* environment (if you have not done so already)
- Once open, type the following on the command line:

```
pip install --ignore-installed --upgrade tensorflow==1.14
```

- Wait for the installation to finish

1.3.1.3 Test your Installation

- Open a new *Terminal* window and activate the *tensorflow_cpu* environment (if you have not done so already)
- Start a new Python interpreter session by running:

```
python
```

- Once the interpreter opens up, type:

```
>>> import tensorflow as tf
```

- If the above code shows an error, then check to make sure you have activated the *tensorflow_cpu* environment and that tensorflow_cpu was successfully installed within it in the previous step.
- Then run the following:

```
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
```

- Once the above is run, if you see a print-out similar (or identical) to the one below, it means that you could benefit from installing TensorFlow by building the sources that correspond to you specific CPU. Everything should still run as normal, but potentially slower than if you had built TensorFlow from source.

```
2019-02-28 11:59:25.810663: I T:\src\github\tensorflow\tensorflow\core\
↪platform\cpu_feature_guard.cc:141] Your CPU supports instructions that␣
↪this TensorFlow binary was not compiled to use: AVX2
```

- Finally, run the following:

```
>>> print(sess.run(hello))
b'Hello, TensorFlow!'
```

1.3.2 TensorFlow GPU

The installation of *TensorFlow GPU* is slightly more involved than that of *TensorFlow CPU*, mainly due to the need of installing the relevant Graphics and CUDE drivers. There's a nice Youtube tutorial (see here), explaining how to install TensorFlow GPU. Although it describes different versions of the relevant components (including TensorFlow itself), the installation steps are generally the same with this tutorial.

Before proceeding to install TensorFlow GPU, you need to make sure that your system can satisfy the following requirements:

Prerequisites
Nvidia GPU (GTX 650 or newer)
CUDA Toolkit v10.0
CuDNN 7.6.5
Anaconda with Python 3.7 (Optional)

1.3.2.1 Install CUDA Toolkit

Windows

Follow this link to download and install CUDA Toolkit 10.0.

Linux

Follow this link to download and install CUDA Toolkit 10.0 for your Linux distribution.

1.3.2.2 Install CUDNN

Windows

- Go to https://developer.nvidia.com/rdp/cudnn-download
- Create a user profile if needed and log in
- Select cuDNN v7.6.5 (Nov 5, 2019), for CUDA 10.0
- Download cuDNN v7.6.5 Library for Windows 10
- Extract the contents of the zip file (i.e. the folder named `cuda`) inside `<INSTALL_PATH>\NVIDIA GPU Computing Toolkit\CUDA\v10.0\`, where `<INSTALL_PATH>` points to the installation directory specified during the installation of the CUDA Toolkit. By default `<INSTALL_PATH>` = `C:\Program Files`.

Linux

- Go to https://developer.nvidia.com/rdp/cudnn-download
- Create a user profile if needed and log in
- Select cuDNN v7.6.5 (Nov 5, 2019), for CUDA 10.0
- Download cuDNN v7.6.5 Library for Linux
- Follow the instructions under Section 2.3.1 of the CuDNN Installation Guide to install CuDNN.

1.3.2.3 Environment Setup

Windows

- Go to *Start* and Search "environment variables"
- Click "Edit the system environment variables". This should open the "System Properties" window
- In the opened window, click the "Environment Variables…" button to open the "Environment Variables" window.
- Under "System variables", search for and click on the `Path` system variable, then click "Edit…"
- Add the following paths, then click "OK" to save the changes:
  - `<INSTALL_PATH>\NVIDIA GPU Computing Toolkit\CUDA\v10.0\bin`

  - `<INSTALL_PATH>\NVIDIA GPU Computing Toolkit\CUDA\v10.0\libnvvp`
  - `<INSTALL_PATH>\NVIDIA GPU Computing Toolkit\CUDA\v10.0\extras\CUPTI\libx64`
  - `<INSTALL_PATH>\NVIDIA GPU Computing Toolkit\CUDA\v10.0\cuda\bin`

Linux

As per Section 7.1.1 of the CUDA Installation Guide for Linux, append the following lines to `~/.bashrc`:

```
# CUDA related exports
export PATH=/usr/local/cuda-10.0/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda-10.0/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_
↪PATH}}
```

1.3.2.4 Update your GPU drivers (Optional)

If during the installation of the CUDA Toolkit (see *Install CUDA Toolkit*) you selected the *Express Installation* option, then your GPU drivers will have been overwritten by those that come bundled with the CUDA toolkit. These drivers are typically NOT the latest drivers and, thus, you may wish to updte your drivers.

- Go to http://www.nvidia.com/Download/index.aspx
- Select your GPU version to download
- Install the driver for your chosen OS

1.3.2.5 Create a new Conda virtual environment

- Open a new *Terminal* window
- Type the following command:

```
conda create -n tensorflow_gpu pip python=3.7
```

- The above will create a new virtual environment with name `tensorflow_gpu`
- Now lets activate the newly created virtual environment by running the following in the *Anaconda Promt* window:

```
activate tensorflow_gpu
```

Once you have activated your virtual environment, the name of the environment should be displayed within brackets at the beggining of your cmd path specifier, e.g.:

```
(tensorflow_gpu) C:\Users\sglvladi>
```

1.3.2.6 Install TensorFlow GPU for Python

- Open a new *Terminal* window and activate the *tensorflow_gpu* environment (if you have not done so already)
- Once open, type the following on the command line:

```
pip install --upgrade tensorflow-gpu==1.14
```

- Wait for the installation to finish

1.3.2.7 Test your Installation

- Open a new *Terminal* window and activate the *tensorflow_gpu* environment (if you have not done so already)
- Start a new Python interpreter session by running:

```
python
```

- Once the interpreter opens up, type:

```
>>> import tensorflow as tf
```

- If the above code shows an error, then check to make sure you have activated the *tensorflow_gpu* environment and that tensorflow_gpu was successfully installed within it in the previous step.
- Then run the following:

```
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
```

- Once the above is run, you should see a print-out similar (but not identical) to the one bellow:

```
2019-11-25 07:20:32.415386: I tensorflow/stream_executor/platform/default/
↪dso_loader.cc:44] Successfully opened dynamic library nvcuda.dll
2019-11-25 07:20:32.449116: I tensorflow/core/common_runtime/gpu/gpu_
↪device.cc:1618] Found device 0 with properties:
name: GeForce GTX 1070 Ti major: 6 minor: 1 memoryClockRate(GHz): 1.683
pciBusID: 0000:01:00.0
2019-11-25 07:20:32.455223: I tensorflow/stream_executor/platform/default/
↪dlopen_checker_stub.cc:25] GPU libraries are statically linked, skip␣
↪dlopen check.
2019-11-25 07:20:32.460799: I tensorflow/core/common_runtime/gpu/gpu_
↪device.cc:1746] Adding visible gpu devices: 0
2019-11-25 07:20:32.464391: I tensorflow/core/platform/cpu_feature_guard.
↪cc:142] Your CPU supports instructions that this TensorFlow binary was␣
↪not compiled to use: AVX2
2019-11-25 07:20:32.472682: I tensorflow/core/common_runtime/gpu/gpu_
↪device.cc:1618] Found device 0 with properties:
name: GeForce GTX 1070 Ti major: 6 minor: 1 memoryClockRate(GHz): 1.683
pciBusID: 0000:01:00.0
2019-11-25 07:20:32.478942: I tensorflow/stream_executor/platform/default/
↪dlopen_checker_stub.cc:25] GPU libraries are statically linked, skip␣
↪dlopen check.
2019-11-25 07:20:32.483948: I tensorflow/core/common_runtime/gpu/gpu_
↪device.cc:1746] Adding visible gpu devices: 0
2019-11-25 07:20:33.181565: I tensorflow/core/common_runtime/gpu/gpu_
↪device.cc:1159] Device interconnect StreamExecutor with strength 1 edge␣
↪matrix:
```

(continues on next page)

(continued from previous page)

```
2019-11-25 07:20:33.185974: I tensorflow/core/common_runtime/gpu/gpu_
↪device.cc:1165]      0
2019-11-25 07:20:33.189041: I tensorflow/core/common_runtime/gpu/gpu_
↪device.cc:1178] 0:   N
2019-11-25 07:20:33.193290: I tensorflow/core/common_runtime/gpu/gpu_
↪device.cc:1304] Created TensorFlow device (/job:localhost/replica:0/
↪task:0/device:GPU:0 with 6358 MB memory) -> physical GPU (device: 0,␣
↪name: GeForce GTX 1070 Ti, pci bus id: 0000:01:00.0, compute␣
↪capability: 6.1)
```

- Finally, run the following:

```
>>> print(sess.run(hello))
b'Hello, TensorFlow!'
```

1.4 TensorFlow Models Installation

Now that you have installed TensorFlow, it is time to install the models used by TensorFlow to do its magic.

1.4.1 Install Prerequisites

Building on the assumption that you have just created your new virtual environment (whether that's *tensorflow_cpu*, *tensorflow_gpu* or whatever other name you might have used), there are some packages which need to be installed before installing the models.

Prerequisite packages	
Name	Tutorial version-build
pillow	6.2.1-py37hdc69c19_0
lxml	4.4.1-py37h1350720_0
jupyter	1.0.0-py37_7
matplotlib	3.1.1-py37hc8f65d3_0
opencv	3.4.2-py37hc319ecb_0
pathlib	1.0.1-cp37

The packages can be installed using `conda` by running:

```
conda install <package_name>(=<version>) <package_name>(=<version>) ... <package_name>
↪(=<version>)
```

where `<package_name>` can be replaced with the name of the package, and optionally the package version can be specified by adding the optional specifier `=<version>` after `<package_name>`. For example, to simply install all packages at their latest versions you can run:

```
conda install pillow lxml jupyter matplotlib opencv cython
```

Alternatively, if you don't want to use Anaconda you can install the packages using `pip`:

```
pip install <package_name>(==<version>) <package_name>(==<version>) ... <package_name>
↪(==<version>)
```

but you will need to install `opencv-python` instead of `opencv`.

1.4.2 Downloading the TensorFlow Models

Note: To ensure compatibility with the chosen version of Tensorflow (i.e. `1.14.0`), it is generally recommended to use one of the Tensorflow Models releases, as they are most likely to be stable. Release `v1.13.0` is the last unofficial release before `v2.0` and therefore is the one used here.

- Create a new folder under a path of your choice and name it `TensorFlow`. (e.g. `C:\Users\sglvladi\Documents\TensorFlow`).
- From your *Terminal* `cd` into the `TensorFlow` directory.
- To download the models you can either use Git to clone the TensorFlow Models v.1.13.0 release inside the `TensorFlow` folder, or you can simply download it as a ZIP and extract it's contents inside the `TensorFlow` folder. To keep things consistent, in the latter case you will have to rename the extracted folder `models-r1.13.0` to `models`.
- You should now have a single folder named `models` under your `TensorFlow` folder, which contains another 4 folders as such:

```
TensorFlow
└─ models
    ├── official
    ├── research
    ├── samples
    └── tutorials
```

1.4.3 Protobuf Installation/Compilation

The Tensorflow Object Detection API uses Protobufs to configure model and training parameters. Before the framework can be used, the Protobuf libraries must be downloaded and compiled.

This should be done as follows:

- Head to the protoc releases page
- Download the latest `protoc-*-*.zip` release (e.g. `protoc-3.11.0-win64.zip` for 64-bit Windows)
- Extract the contents of the downloaded `protoc-*-*.zip` in a directory <PATH_TO_PB> of your choice (e.g. `C:\Program Files\Google Protobuf`)
- Extract the contents of the downloaded `protoc-*-*.zip`, inside `C:\Program Files\Google Protobuf`
- Add <PATH_TO_PB> to your `Path` environment variable (see *Environment Setup*)
- In a new *Terminal*[1], `cd` into `TensorFlow/models/research/` directory and run the following command:

```
# From within TensorFlow/models/research/
protoc object_detection/protos/*.proto --python_out=.
```

Important: If you are on Windows and using Protobuf 3.5 or later, the multi-file selection wildcard (i.e `*.proto`) may not work but you can do one of the following:

Windows Powershell

[1] NOTE: You MUST open a new *Terminal* for the changes in the environment variables to take effect.

```
# From within TensorFlow/models/research/
Get-ChildItem object_detection/protos/*.proto | foreach {protoc "object_detection/
↪protos/$($_.Name)" --python_out=.}
```

Command Prompt

```
# From within TensorFlow/models/research/
for /f %i in ('dir /b object_detection\protos\*.proto') do protoc object_detection\
↪protos\%i --python_out=.
```

1.4.4 Adding necessary Environment Variables

1. Install the `Tensorflow\models\research\object_detection` package by running the following from `Tensorflow\models\research`:

   ```
   # From within TensorFlow/models/research/
   pip install .
   ```

2. Add *research/slim* to your `PYTHONPATH`:

Windows

- Go to *Start* and Search "environment variables"
- Click "Edit the system environment variables". This should open the "System Properties" window
- In the opened window, click the "Environment Variables…" button to open the "Environment Variables" window.
- Under "System variables", search for and click on the `PYTHONPATH` system variable,
  - If it exists then click "Edit…" and add `<PATH_TO_TF>\TensorFlow\models\research\slim` to the list
  - If it doesn't already exist, then click "New…", under "Variable name" type `PYTHONPATH` and under "Variable value" enter `<PATH_TO_TF>\TensorFlow\models\research\slim`
- Then click "OK" to save the changes:

Linux

The Installation docs suggest that you either run, or add to `~/.bashrc` file, the following command, which adds these packages to your PYTHONPATH:

```
# From within tensorflow/models/research/
export PYTHONPATH=$PYTHONPATH:<PATH_TO_TF>/TensorFlow/models/research/slim
```

where, in both cases, `<PATH_TO_TF>` replaces the absolute path to your `TensorFlow` folder. (e.g. `<PATH_TO_TF> = C:\Users\sglvladi\Documents` if `TensorFlow` resides within your `Documents` folder)

1.4.5 COCO API installation (Optional)

The `pycocotools` package should be installed if you are interested in using COCO evaluation metrics, as discussed in *Evaluating the Model (Optional)*.

Windows

Run the following command to install `pycocotools` with Windows support:

```
pip install git+https://github.com/philferriere/cocoapi.git#subdirectory=PythonAPI
```

Note that, according to the package's instructions, Visual C++ 2015 build tools must be installed and on your path. If they are not, make sure to install them from here.

Linux

Download cocoapi to a directory of your choice, then `make` and copy the pycocotools subfolder to the `Tensorflow/models/research` directory, as such:

```
git clone https://github.com/cocodataset/cocoapi.git
cd cocoapi/PythonAPI
make
cp -r pycocotools <PATH_TO_TF>/TensorFlow/models/research/
```

Note: The default metrics are based on those used in Pascal VOC evaluation.

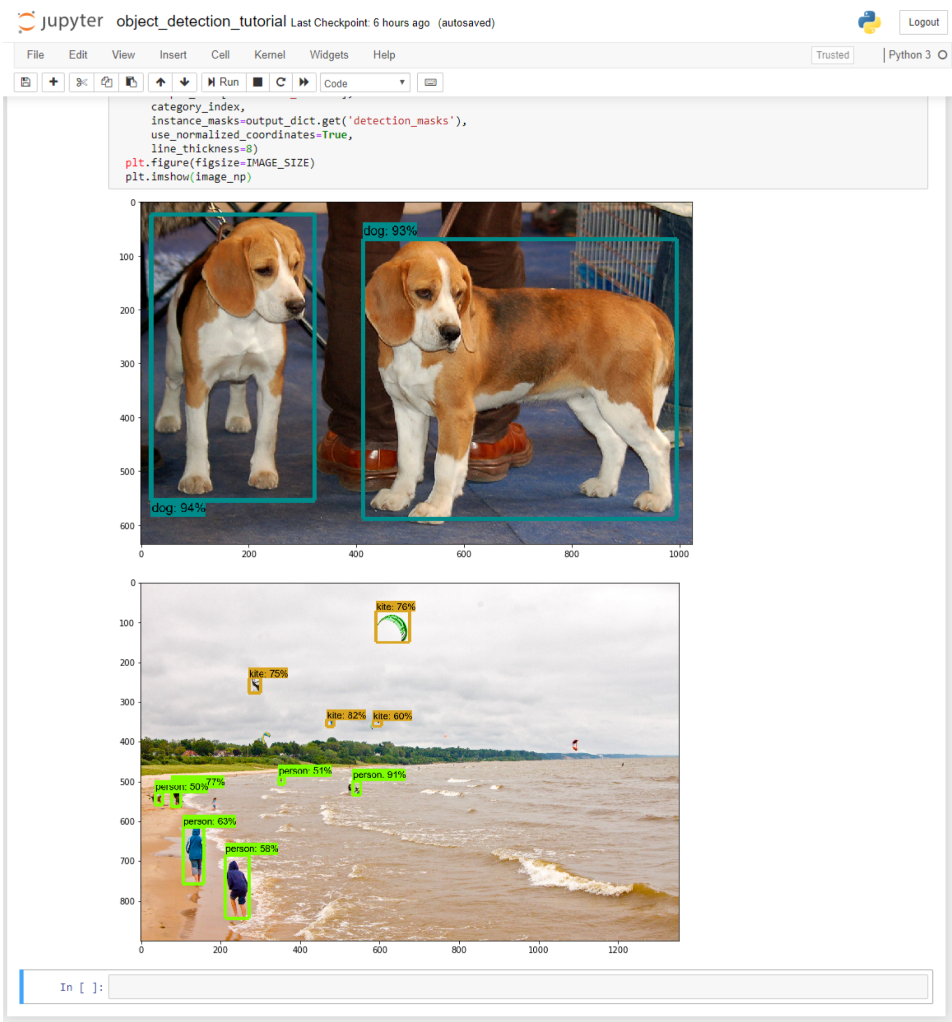
- To use the COCO object detection metrics add `metrics_set: "coco_detection_metrics"` to the `eval_config` message in the config file.
- To use the COCO instance segmentation metrics add `metrics_set: "coco_mask_metrics"` to the `eval_config` message in the config file.

1.4.6 Test your Installation

- Open a new *Terminal* window and activate the *tensorflow_gpu* environment (if you have not done so already)
- `cd` into `TensorFlow\models\research\object_detection` and run the following command:

```
# From within TensorFlow/models/research/object_detection
jupyter notebook
```

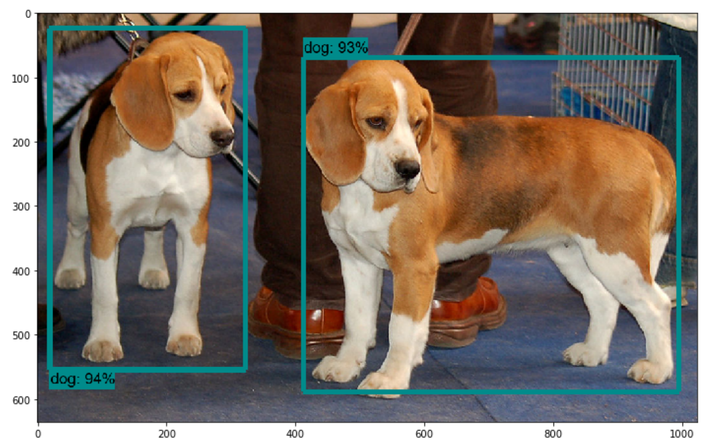
- This should start a new `jupyter notebook` server on your machine and you should be redirected to a new tab of your default browser.
- Once there, simply follow sentdex's Youtube video to ensure that everything is running smoothly.
- When done, your notebook should look similar to the image bellow:



Important:

1. If no errors appear, but also no images are shown in the notebook, try adding `%matplotlib inline` at the start of the last cell, as shown by the highlighted text in the image bellow:

In [11]:
```
# This is needed to display the images.
%matplotlib inline
for image_path in TEST_IMAGE_PATHS:
  image = Image.open(image_path)
  # the array based representation of the image will be used later in order to prepare the
  # result image with boxes and labels on it.
  image_np = load_image_into_numpy_array(image)
  # Expand dimensions since the model expects images to have shape: [1, None, None, 3]
  image_np_expanded = np.expand_dims(image_np, axis=0)
  # Actual detection.
  output_dict = run_inference_for_single_image(image_np, detection_graph)
  # Visualization of the results of a detection.
  vis_util.visualize_boxes_and_labels_on_image_array(
      image_np,
      output_dict['detection_boxes'],
      output_dict['detection_classes'],
      output_dict['detection_scores'],
      category_index,
      instance_masks=output_dict.get('detection_masks'),
      use_normalized_coordinates=True,
      line_thickness=8)
  plt.figure(figsize=IMAGE_SIZE)
  plt.imshow(image_np)
```



2. If Python crashes when running the last cell, have a look at the *Terminal* window you used to run `jupyter notebook` and check for an error similar (maybe identical) to the one below:

```
2018-03-22 03:07:54.623130: E C:\tf_jenkins\workspace\rel-win\M\windows-
↪gpu\PY\36\tensorflow\stream_executor\cuda\cuda_dnn.cc:378] Loaded␣
↪runtime CuDNN library: 7101 (compatibility version 7100) but source was␣
↪compiled with 7003 (compatibility version 7000).  If using a binary␣
↪install, upgrade your CuDNN library to match.  If building from sources,
↪ make sure the library loaded at runtime matches a compatible version␣
↪specified during compile configuration.
```

   • If the above line is present in the printed debugging, it means that you have not installed the correct version of the cuDNN libraries. In this case make sure you re-do the *Install CUDNN* step, making sure you instal cuDNN v7.6.5.

1.5 LabelImg Installation

There exist several ways to install `labelImg`. Below are 3 of the most common.

1.5.1 Get from PyPI (Recommended)

1. Open a new *Terminal* window and activate the *tensorflow_gpu* environment (if you have not done so already)
2. Run the following command to install `labelImg`:

```
pip install labelImg
```

3. `labelImg` can then be run as follows:

```
labelImg
# or
labelImg [IMAGE_PATH] [PRE-DEFINED CLASS FILE]
```

1.5.2 Use precompiled binaries (Easy)

Precompiled binaries for both Windows and Linux can be found here .

Installation is the done in three simple steps:

1. Inside you `TensorFlow` folder, create a new directory, name it `addons` and then `cd` into it.
2. Download the latest binary for your OS from here. and extract its contents under `Tensorflow/addons/labelImg`.
3. You should now have a single folder named `addons\labelImg` under your `TensorFlow` folder, which contains another 4 folders as such:

```
TensorFlow
├─ addons
│   └── labelImg
└─ models
    ├── official
    ├── research
    ├── samples
    └── tutorials
```

4. `labelImg` can then be run as follows:

```
# From within Tensorflow/addons/labelImg
labelImg
# or
labelImg [IMAGE_PATH] [PRE-DEFINED CLASS FILE]
```

1.5.3 Build from source (Hard)

The steps for installing from source follow below.

1. Download labelImg

- Inside you `TensorFlow` folder, create a new directory, name it `addons` and then `cd` into it.
- To download the package you can either use Git to clone the labelImg repo inside the `TensorFlow\addons` folder, or you can simply download it as a ZIP and extract it's contents inside the `TensorFlow\addons` folder. To keep things consistent, in the latter case you will have to rename the extracted folder `labelImg-master` to `labelImg`.[2]
- You should now have a single folder named `addons\labelImg` under your `TensorFlow` folder, which contains another 4 folders as such:

```
TensorFlow
├─ addons
│   └── labelImg
└─ models
    ├── official
    ├── research
    ├── samples
    └── tutorials
```

2. Install dependencies and compiling package

- Open a new *Terminal* window and activate the *tensorflow_gpu* environment (if you have not done so already)
- `cd` into `TensorFlow\addons\labelImg` and run the following commands:

  Windows

```
conda install pyqt=5
pyrcc5 -o libs/resources.py resources.qrc
```

  Linux

```
sudo apt-get install pyqt5-dev-tools
sudo pip install -r requirements/requirements-linux-python3.txt
make qt5py3
```

3. Test your installation

- Open a new *Terminal* window and activate the *tensorflow_gpu* environment (if you have not done so already)
- `cd` into `TensorFlow\addons\labelImg` and run the following command:

```
# From within Tensorflow/addons/labelImg
python labelImg.py
# or
python  labelImg.py [IMAGE_PATH] [PRE-DEFINED CLASS FILE]
```

[2] The latest repo commit when writing this tutorial is 8d1bd68.

CHAPTER

TWO

DETECT OBJECTS USING YOUR WEBCAM

Hereby you can find an example which allows you to use your camera to generate a video stream, based on which you can perform object_detection.

To run the example, simply create a new file under `<PATH_TO_TF>/TensorFlow/models/research/object_detection` and paste the code below.

```
import numpy as np
import os
import six.moves.urllib as urllib
import sys
import tarfile
import tensorflow as tf
import zipfile
import cv2

from collections import defaultdict
from io import StringIO
from matplotlib import pyplot as plt
from PIL import Image
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as vis_util

# Define the video stream
cap = cv2.VideoCapture(0)  # Change only if you have more than one webcams

# What model to download.
# Models can bee found here: https://github.com/tensorflow/models/blob/master/
↪research/object_detection/g3doc/detection_model_zoo.md
MODEL_NAME = 'ssd_inception_v2_coco_2017_11_17'
MODEL_FILE = MODEL_NAME + '.tar.gz'
DOWNLOAD_BASE = 'http://download.tensorflow.org/models/object_detection/'

# Path to frozen detection graph. This is the actual model that is used for the␣
↪object detection.
PATH_TO_CKPT = MODEL_NAME + '/frozen_inference_graph.pb'

# List of the strings that is used to add correct label for each box.
PATH_TO_LABELS = os.path.join('data', 'mscoco_label_map.pbtxt')

# Number of classes to detect
NUM_CLASSES = 90

# Download Model
if not os.path.exists(os.path.join(os.getcwd(), MODEL_FILE)):
```

(continues on next page)

(continued from previous page)

```
    print("Downloading model")
    opener = urllib.request.URLopener()
    opener.retrieve(DOWNLOAD_BASE + MODEL_FILE, MODEL_FILE)
    tar_file = tarfile.open(MODEL_FILE)
    for file in tar_file.getmembers():
        file_name = os.path.basename(file.name)
        if 'frozen_inference_graph.pb' in file_name:
            tar_file.extract(file, os.getcwd())


# Load a (frozen) Tensorflow model into memory.
detection_graph = tf.Graph()
with detection_graph.as_default():
    od_graph_def = tf.compat.v1.GraphDef()
    with tf.io.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
        serialized_graph = fid.read()
        od_graph_def.ParseFromString(serialized_graph)
        tf.import_graph_def(od_graph_def, name='')


# Loading label map
# Label maps map indices to category names, so that when our convolution network␣
↪predicts `5`, we know that this corresponds to `airplane`.  Here we use internal␣
↪utility functions, but anything that returns a dictionary mapping integers to␣
↪appropriate string labels would be fine
label_map = label_map_util.load_labelmap(PATH_TO_LABELS)
categories = label_map_util.convert_label_map_to_categories(
    label_map, max_num_classes=NUM_CLASSES, use_display_name=True)
category_index = label_map_util.create_category_index(categories)


# Helper code
def load_image_into_numpy_array(image):
    (im_width, im_height) = image.size
    return np.array(image.getdata()).reshape(
        (im_height, im_width, 3)).astype(np.uint8)


# Detection
with detection_graph.as_default():
    with tf.compat.v1.Session(graph=detection_graph) as sess:
        while True:
            # Read frame from camera
            ret, image_np = cap.read()
            # Expand dimensions since the model expects images to have shape: [1,␣
↪None, None, 3]
            image_np_expanded = np.expand_dims(image_np, axis=0)
            # Extract image tensor
            image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')
            # Extract detection boxes
            boxes = detection_graph.get_tensor_by_name('detection_boxes:0')
            # Extract detection scores
            scores = detection_graph.get_tensor_by_name('detection_scores:0')
            # Extract detection classes
            classes = detection_graph.get_tensor_by_name('detection_classes:0')
            # Extract number of detectionsd
            num_detections = detection_graph.get_tensor_by_name(
```

(continues on next page)

(continued from previous page)

```
            'num_detections:0')
        # Actual detection.
        (boxes, scores, classes, num_detections) = sess.run(
            [boxes, scores, classes, num_detections],
            feed_dict={image_tensor: image_np_expanded})
        # Visualization of the results of a detection.
        vis_util.visualize_boxes_and_labels_on_image_array(
            image_np,
            np.squeeze(boxes),
            np.squeeze(classes).astype(np.int32),
            np.squeeze(scores),
            category_index,
            use_normalized_coordinates=True,
            line_thickness=8)

        # Display output
        cv2.imshow('object detection', cv2.resize(image_np, (800, 600)))

        if cv2.waitKey(25) & 0xFF == ord('q'):
            cv2.destroyAllWindows()
            break
```

CHAPTER

THREE

TRAINING CUSTOM OBJECT DETECTOR

So, up to now you should have done the following:

- Installed TensorFlow, either CPU or GPU (See *TensorFlow Installation*)
- Installed TensorFlow Models (See *TensorFlow Models Installation*)
- Installed labelImg (See *LabelImg Installation*)

Now that we have done all the above, we can start doing some cool stuff. Here we will see how you can train your own object detector, and since it is not as simple as it sounds, we will have a look at:

1. How to organise your workspace/training files
2. How to prepare/annotate image datasets
3. How to generate tf records from such datasets
4. How to configure a simple training pipeline
5. How to train a model and monitor it's progress
6. How to export the resulting model and use it to detect objects.

3.1 Preparing workspace

1. If you have followed the tutorial, you should by now have a folder `Tensorflow`, placed under `<PATH_TO_TF>` (e.g. `C:\Users\sglvladi\Documents`), with the following directory tree:

```
TensorFlow
├─ addons
│   └── labelImg
└─ models
    ├── official
    ├── research
    ├── samples
    └── tutorials
```

2. Now create a new folder under `TensorFlow` and call it `workspace`. It is within the `workspace` that we will store all our training set-ups. Now let's go under workspace and create another folder named `training_demo`. Now our directory structure should be as so:

```
TensorFlow
├─ addons
│   └─ labelImg
├─ models
```

(continues on next page)

(continued from previous page)

```
│   ├─ official
│   ├─ research
│   ├─ samples
│   └─ tutorials
└─ workspace
    └─ training_demo
```

3. The `training_demo` folder shall be our *training folder*, which will contain all files related to our model training. It is advisable to create a separate training folder each time we wish to train a different model. The typical structure for training folders is shown below.

```
training_demo
├─ annotations
├─ images
│   ├─ test
│   └─ train
├─ pre-trained-model
├─ training
└─ README.md
```

Here's an explanation for each of the folders/filer shown in the above tree:

- `annotations`: This folder will be used to store all `*.csv` files and the respective TensorFlow `*.record` files, which contain the list of annotations for our dataset images.
- `images`: This folder contains a copy of all the images in our dataset, as well as the respective `*.xml` files produced for each one, once `labelImg` is used to annotate objects.
  - `images\train`: This folder contains a copy of all images, and the respective `*.xml` files, which will be used to train our model.
  - `images\test`: This folder contains a copy of all images, and the respective `*.xml` files, which will be used to test our model.
- `pre-trained-model`: This folder will contain the pre-trained model of our choice, which shall be used as a starting checkpoint for our training job.
- `training`: This folder will contain the training pipeline configuration file `*.config`, as well as a `*.pbtxt` label map file and all files generated during the training of our model.
- `README.md`: This is an optional file which provides some general information regarding the training conditions of our model. It is not used by TensorFlow in any way, but it generally helps when you have a few training folders and/or you are revisiting a trained model after some time.

If you do not understand most of the things mentioned above, no need to worry, as we'll see how all the files are generated further down.

3.2 Annotating images

To annotate images we will be using the labelImg package. If you haven't installed the package yet, then have a look at *LabelImg Installation*.

- Once you have collected all the images to be used to test your model (ideally more than 100 per class), place them inside the folder `training_demo\images`.
- Open a new *Anaconda/Command Prompt* window and `cd` into `Tensorflow\addons\labelImg`.

- If (as suggested in *LabelImg Installation*) you created a separate Conda environment for `labelImg` then go ahead and activate it by running:

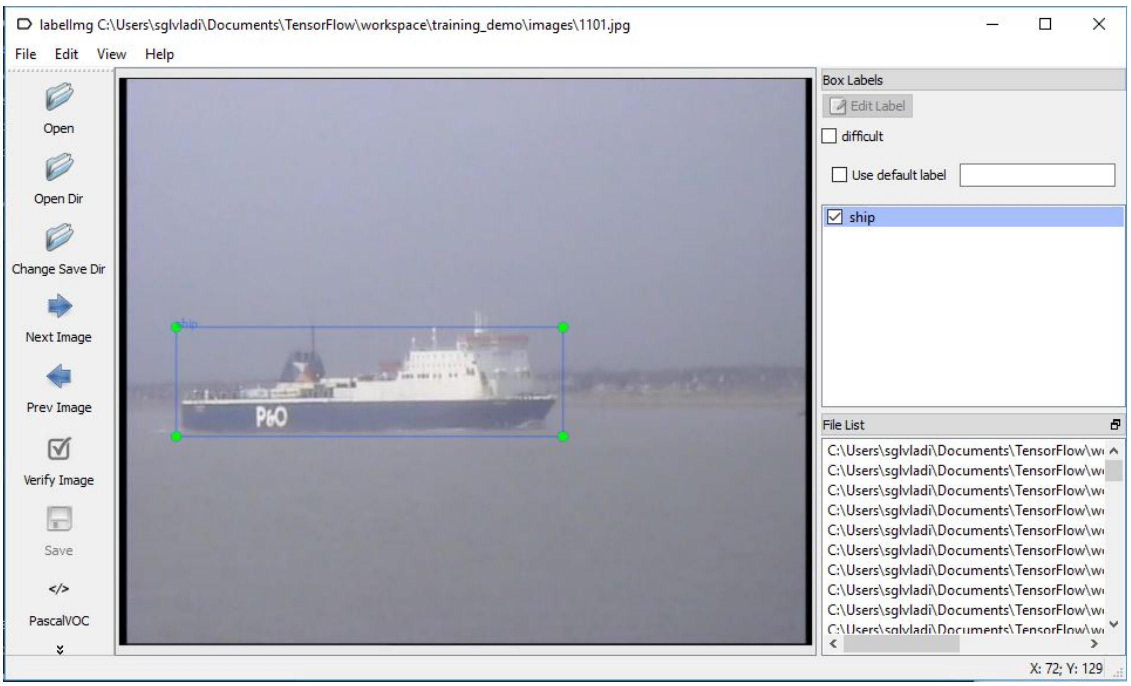
```
activate labelImg
```

- Next go ahead and start `labelImg`, pointing it to your `training_demo\images` folder.

```
python labelImg.py ..\..\workspace\training_demo\images
```

- A File Explorer Dialog windows should open, which points to the `training_demo\images` folder.
- Press the "Select Folder" button, to start annotating your images.

Once open, you should see a window similar to the one below:



I won't be covering a tutorial on how to use `labelImg`, but you can have a look at labelImg's repo for more details. A nice Youtube video demonstrating how to use `labelImg` is also available here. What is important is that once you annotate all your images, a set of new `*.xml` files, one for each image, should be generated inside your `training_demo\images` folder.

3.3 Partitioning the images

Once you have finished annotating your image dataset, it is a general convention to use only part of it for training, and the rest is used for evaluation purposes (e.g. as discussed in *Evaluating the Model (Optional)*).

Typically, the ratio is 90%/10%, i.e. 90% of the images are used for training and the rest 10% is maintained for testing, but you can chose whatever ratio suits your needs.

Once you have decided how you will be splitting your dataset, copy all training images, together with their corresponding `*.xml` files, and place them inside the `training_demo\images\train` folder. Similarly, copy all testing images, with their `*.xml` files, and paste them inside `training_demo\images\test`.

For lazy people like myself, who cannot be bothered to do the above, I have put tugether a simple script that automates the above process:

```
""" usage: partition_dataset.py [-h] [-i IMAGEDIR] [-o OUTPUTDIR] [-r RATIO] [-x]

Partition dataset of images into training and testing sets

optional arguments:
  -h, --help            show this help message and exit
  -i IMAGEDIR, --imageDir IMAGEDIR
                        Path to the folder where the image dataset is stored. If not␣
↪specified, the CWD will be used.
  -o OUTPUTDIR, --outputDir OUTPUTDIR
                        Path to the output folder where the train and test dirs␣
↪should be created. Defaults to the same directory as IMAGEDIR.
  -r RATIO, --ratio RATIO
                        The ratio of the number of test images over the total number␣
↪of images. The default is 0.1.
  -x, --xml             Set this flag if you want the xml annotation files to be␣
↪processed and copied over.
"""
import os
import re
import shutil
from PIL import Image
from shutil import copyfile
import argparse
import glob
import math
import random
import xml.etree.ElementTree as ET


def iterate_dir(source, dest, ratio, copy_xml):
    source = source.replace('\\', '/')
    dest = dest.replace('\\', '/')
    train_dir = os.path.join(dest, 'train')
    test_dir = os.path.join(dest, 'test')

    if not os.path.exists(train_dir):
        os.makedirs(train_dir)
    if not os.path.exists(test_dir):
        os.makedirs(test_dir)

    images = [f for f in os.listdir(source)
              if re.search(r'([a-zA-Z0-9\s_\\.\-\(\):])+(.jpg|.jpeg|.png)$', f)]

    num_images = len(images)
    num_test_images = math.ceil(ratio*num_images)

    for i in range(num_test_images):
        idx = random.randint(0, len(images)-1)
        filename = images[idx]
        copyfile(os.path.join(source, filename),
                 os.path.join(test_dir, filename))
        if copy_xml:
            xml_filename = os.path.splitext(filename)[0]+'.xml'
            copyfile(os.path.join(source, xml_filename),
                     os.path.join(test_dir,xml_filename))
        images.remove(images[idx])
```

(continues on next page)

(continued from previous page)

```
    for filename in images:
        copyfile(os.path.join(source, filename),
                 os.path.join(train_dir, filename))
        if copy_xml:
            xml_filename = os.path.splitext(filename)[0]+'.xml'
            copyfile(os.path.join(source, xml_filename),
                     os.path.join(train_dir, xml_filename))


def main():

    # Initiate argument parser
    parser = argparse.ArgumentParser(description="Partition dataset of images into training and testing sets",
                                     formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument(
        '-i', '--imageDir',
        help='Path to the folder where the image dataset is stored. If not specified, the CWD will be used.',
        type=str,
        default=os.getcwd()
    )
    parser.add_argument(
        '-o', '--outputDir',
        help='Path to the output folder where the train and test dirs should be created. '
             'Defaults to the same directory as IMAGEDIR.',
        type=str,
        default=None
    )
    parser.add_argument(
        '-r', '--ratio',
        help='The ratio of the number of test images over the total number of images. The default is 0.1.',
        default=0.1,
        type=float)
    parser.add_argument(
        '-x', '--xml',
        help='Set this flag if you want the xml annotation files to be processed and copied over.',
        action='store_true'
    )
    args = parser.parse_args()

    if args.outputDir is None:
        args.outputDir = args.imageDir

    # Now we are ready to start the iteration
    iterate_dir(args.imageDir, args.outputDir, args.ratio, args.xml)


if __name__ == '__main__':
    main()
```

To use the script, simply copy and paste the code above in a script named `partition_dataset.py`. Then, assuming you have all your images and `*.xml` files inside `training_demo\images`, just run the following command:

```
python partition_dataser.py -x -i training_demo\images -r 0.1
```

Once the script has finished, there should exist two new folders under `training_demo\images`, namely `training_demo\images\train` and `training_demo\images\test`, containing 90% and 10% of the images (and `*.xml` files), respectively. To avoid loss of any files, the script will not delete the images under `training_demo\images`. Once you have checked that your images have been safely copied over, you can delete the images under `training_demo\images` manually.

3.4 Creating Label Map

TensorFlow requires a label map, which namely maps each of the used labels to an integer values. This label map is used both by the training and detection processes.

Below I show an example label map (e.g `label_map.pbtxt`), assuming that our dataset containes 2 labels, `dogs` and `cats`:

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

item {
    id: 2
    name: 'dog'
}
```

Label map files have the extention `.pbtxt` and should be placed inside the `training_demo\annotations` folder.

3.5 Creating TensorFlow Records

Now that we have generated our annotations and split our dataset into the desired training and testing subsets, it is time to convert our annotations into the so called `TFRecord` format.

There are two steps in doing so:

- Converting the individual `*.xml` files to a unified `*.csv` file for each dataset.
- Converting the `*.csv` files of each dataset to `*.record` files (TFRecord format).

Before we proceed to describe the above steps, let's create a directory where we can store some scripts. Under the `TensorFlow` folder, create a new folder `TensorFlow\scripts`, which we can use to store some useful scripts. To make things even tidier, let's create a new folder `TensorFlow\scripts\preprocessing`, where we shall store scripts that we can use to preprocess our training inputs. Below is out `TensorFlow` directory tree structure, up to now:

```
TensorFlow
├─ addons
│   └─ labelImg
├─ models
│   ├─ official
│   ├─ research
```

(continues on next page)

(continued from previous page)

```
│   ├─ samples
│   └─ tutorials
├─ scripts
│   └─ preprocessing
└─ workspace
    └─ training_demo
```

3.5.1 Converting `*.xml` to `*.csv`

To do this we can write a simple script that iterates through all `*.xml` files in the `training_demo\images\train` and `training_demo\images\test` folders, and generates a `*.csv` for each of the two.

Here is an example script that allows us to do just that:

```
"""
Usage:
# Create train data:
python xml_to_csv.py -i [PATH_TO_IMAGES_FOLDER]/train -o [PATH_TO_ANNOTATIONS_FOLDER]/
↪train_labels.csv

# Create test data:
python xml_to_csv.py -i [PATH_TO_IMAGES_FOLDER]/test -o [PATH_TO_ANNOTATIONS_FOLDER]/
↪test_labels.csv
"""

import os
import glob
import pandas as pd
import argparse
import xml.etree.ElementTree as ET


def xml_to_csv(path):
    """Iterates through all .xml files (generated by labelImg) in a given directory␣
↪and combines them in a single Pandas datagrame.

    Parameters:
    ----------
    path : {str}
        The path containing the .xml files
    Returns
    -------
    Pandas DataFrame
        The produced dataframe
    """

    xml_list = []
    for xml_file in glob.glob(path + '/*.xml'):
        tree = ET.parse(xml_file)
        root = tree.getroot()
        for member in root.findall('object'):
            value = (root.find('filename').text,
                    int(root.find('size')[0].text),
                    int(root.find('size')[1].text),
                    member[0].text,
```

(continues on next page)

(continued from previous page)

```
                    int(member[4][0].text),
                    int(member[4][1].text),
                    int(member[4][2].text),
                    int(member[4][3].text)
                    )
            xml_list.append(value)
    column_name = ['filename', 'width', 'height',
                'class', 'xmin', 'ymin', 'xmax', 'ymax']
    xml_df = pd.DataFrame(xml_list, columns=column_name)
    return xml_df


def main():
    # Initiate argument parser
    parser = argparse.ArgumentParser(
        description="Sample TensorFlow XML-to-CSV converter")
    parser.add_argument("-i",
                        "--inputDir",
                        help="Path to the folder where the input .xml files are stored
↪",
                        type=str)
    parser.add_argument("-o",
                        "--outputFile",
                        help="Name of output .csv file (including path)", type=str)
    args = parser.parse_args()

    if(args.inputDir is None):
        args.inputDir = os.getcwd()
    if(args.outputFile is None):
        args.outputFile = args.inputDir + "/labels.csv"

    assert(os.path.isdir(args.inputDir))

    xml_df = xml_to_csv(args.inputDir)
    xml_df.to_csv(
        args.outputFile, index=None)
    print('Successfully converted xml to csv.')


if __name__ == '__main__':
    main()
```

- Create a new file with name `xml_to_csv.py` under `TensorFlow\scripts\preprocessing`, open it, paste the above code inside it and save.
- Install the `pandas` package:

```
conda install pandas # Anaconda
                     # or
pip install pandas   # pip
```

- Finally, `cd` into `TensorFlow\scripts\preprocessing` and run:

```
# Create train data:
python xml_to_csv.py -i [PATH_TO_IMAGES_FOLDER]/train -o [PATH_TO_
↪ANNOTATIONS_FOLDER]/train_labels.csv
```

(continues on next page)

(continued from previous page)

```
# Create test data:
python xml_to_csv.py -i [PATH_TO_IMAGES_FOLDER]/test -o [PATH_TO_
↪ANNOTATIONS_FOLDER]/test_labels.csv

# For example
# python xml_to_csv.py -i C:\Users\sglvladi\Documents\TensorFlow\
↪workspace\training_demo\images\train -o C:\Users\sglvladi\Documents\
↪TensorFlow\workspace\training_demo\annotations\train_labels.csv
# python xml_to_csv.py -i C:\Users\sglvladi\Documents\TensorFlow\
↪workspace\training_demo\images\test -o C:\Users\sglvladi\Documents\
↪TensorFlow\workspace\training_demo\annotations\test_labels.csv
```

Once the above is done, there should be 2 new files under the `training_demo\annotations` folder, named `test_labels.csv` and `train_labels.csv`, respectively.

3.5.2 Converting from `*.csv` to `*.record`

Now that we have obtained our `*.csv` annotation files, we will need to convert them into TFRecords. Below is an example script that allows us to do just that:

```
"""
Usage:

# Create train data:
python generate_tfrecord.py --label=<LABEL> --csv_input=<PATH_TO_ANNOTATIONS_FOLDER>/
↪train_labels.csv  --output_path=<PATH_TO_ANNOTATIONS_FOLDER>/train.record

# Create test data:
python generate_tfrecord.py --label=<LABEL> --csv_input=<PATH_TO_ANNOTATIONS_FOLDER>/
↪test_labels.csv  --output_path=<PATH_TO_ANNOTATIONS_FOLDER>/test.record
"""

from __future__ import division
from __future__ import print_function
from __future__ import absolute_import

import os
import io
import pandas as pd
import tensorflow as tf
import sys
sys.path.append("../../models/research")

from PIL import Image
from object_detection.utils import dataset_util
from collections import namedtuple, OrderedDict

flags = tf.app.flags
flags.DEFINE_string('csv_input', '', 'Path to the CSV input')
flags.DEFINE_string('output_path', '', 'Path to output TFRecord')
flags.DEFINE_string('label', '', 'Name of class label')
# if your image has more labels input them as
# flags.DEFINE_string('label0', '', 'Name of class[0] label')
# flags.DEFINE_string('label1', '', 'Name of class[1] label')
# and so on.
```

(continues on next page)

(continued from previous page)

```
flags.DEFINE_string('img_path', '', 'Path to images')
FLAGS = flags.FLAGS


# TO-DO replace this with label map
# for multiple labels add more else if statements
def class_text_to_int(row_label):
    if row_label == FLAGS.label:  # 'ship':
        return 1
    # comment upper if statement and uncomment these statements for multiple labelling
    # if row_label == FLAGS.label0:
    #   return 1
    # elif row_label == FLAGS.label1:
    #   return 0
    else:
        None


def split(df, group):
    data = namedtuple('data', ['filename', 'object'])
    gb = df.groupby(group)
    return [data(filename, gb.get_group(x)) for filename, x in zip(gb.groups.keys(),␣
↪gb.groups)]


def create_tf_example(group, path):
    with tf.gfile.GFile(os.path.join(path, '{}'.format(group.filename)), 'rb') as fid:
        encoded_jpg = fid.read()
    encoded_jpg_io = io.BytesIO(encoded_jpg)
    image = Image.open(encoded_jpg_io)
    width, height = image.size

    filename = group.filename.encode('utf8')
    image_format = b'jpg'
    # check if the image format is matching with your images.
    xmins = []
    xmaxs = []
    ymins = []
    ymaxs = []
    classes_text = []
    classes = []

    for index, row in group.object.iterrows():
        xmins.append(row['xmin'] / width)
        xmaxs.append(row['xmax'] / width)
        ymins.append(row['ymin'] / height)
        ymaxs.append(row['ymax'] / height)
        classes_text.append(row['class'].encode('utf8'))
        classes.append(class_text_to_int(row['class']))

    tf_example = tf.train.Example(features=tf.train.Features(feature={
        'image/height': dataset_util.int64_feature(height),
        'image/width': dataset_util.int64_feature(width),
        'image/filename': dataset_util.bytes_feature(filename),
        'image/source_id': dataset_util.bytes_feature(filename),
        'image/encoded': dataset_util.bytes_feature(encoded_jpg),
        'image/format': dataset_util.bytes_feature(image_format),
```

(continues on next page)

(continued from previous page)

```
        'image/object/bbox/xmin': dataset_util.float_list_feature(xmins),
        'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs),
        'image/object/bbox/ymin': dataset_util.float_list_feature(ymins),
        'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs),
        'image/object/class/text': dataset_util.bytes_list_feature(classes_text),
        'image/object/class/label': dataset_util.int64_list_feature(classes),
    }))
    return tf_example


def main(_):
    writer = tf.python_io.TFRecordWriter(FLAGS.output_path)
    path = os.path.join(os.getcwd(), FLAGS.img_path)
    examples = pd.read_csv(FLAGS.csv_input)
    grouped = split(examples, 'filename')
    for group in grouped:
        tf_example = create_tf_example(group, path)
        writer.write(tf_example.SerializeToString())

    writer.close()
    output_path = os.path.join(os.getcwd(), FLAGS.output_path)
    print('Successfully created the TFRecords: {}'.format(output_path))


if __name__ == '__main__':
    tf.app.run()
```

- Create a new file with name `generate_tfrecord.py` under `TensorFlow\scripts\preprocessing`, open it, paste the above code inside it and save.
- Once this is done, `cd` into `TensorFlow\scripts\preprocessing` and run:

```
# Create train data:
python generate_tfrecord.py --label=<LABEL> --csv_input=<PATH_TO_
↪ANNOTATIONS_FOLDER>/train_labels.csv
--img_path=<PATH_TO_IMAGES_FOLDER>/train  --output_path=<PATH_TO_
↪ANNOTATIONS_FOLDER>/train.record

# Create test data:
python generate_tfrecord.py --label=<LABEL> --csv_input=<PATH_TO_
↪ANNOTATIONS_FOLDER>/test_labels.csv
--img_path=<PATH_TO_IMAGES_FOLDER>/test
--output_path=<PATH_TO_ANNOTATIONS_FOLDER>/test.record

# For example
# python generate_tfrecord.py --label=ship --csv_input=C:\Users\sglvladi\
↪Documents\TensorFlow\workspace\training_demo\annotations\train_labels.
↪csv --output_path=C:\Users\sglvladi\Documents\TensorFlow\workspace\
↪training_demo\annotations\train.record --img_path=C:\Users\sglvladi\
↪Documents\TensorFlow\workspace\training_demo\images\train
# python generate_tfrecord.py --label=ship --csv_input=C:\Users\sglvladi\
↪Documents\TensorFlow\workspace\training_demo\annotations\test_labels.
↪csv --output_path=C:\Users\sglvladi\Documents\TensorFlow\workspace\
↪training_demo\annotations\test.record --img_path=C:\Users\sglvladi\
↪Documents\TensorFlow\workspace\training_demo\images\test
```

Once the above is done, there should be 2 new files under the `training_demo\annotations` folder, named `test.record` and `train.record`, respectively.

3.6 Configuring a Training Pipeline

For the purposes of this tutorial we will not be creating a training job from the scratch, but rather we will go through how to reuse one of the pre-trained models provided by TensorFlow. If you would like to train an entirely new model, you can have a look at TensorFlow's tutorial.

The model we shall be using in our examples is the `ssd_inception_v2_coco` model, since it provides a relatively good trade-off between performance and speed, however there are a number of other models you can use, all of which are listed in TensorFlow's detection model zoo. More information about the detection performance, as well as reference times of execution, for each of the available pre-trained models can be found here.

First of all, we need to get ourselves the sample pipeline configuration file for the specific model we wish to re-train. You can find the specific file for the model of your choice here. In our case, since we shall be using the `ssd_inception_v2_coco` model, we shall be downloading the corresponding ssd_inception_v2_coco.config file.

Apart from the configuration file, we also need to download the latest pre-trained NN for the model we wish to use. This can be done by simply clicking on the name of the desired model in the tables found in TensorFlow's detection model zoo. Clicking on the name of your model should initiate a download for a `*.tar.gz` file.

Once the `*.tar.gz` file has been downloaded, open it using a decompression program of your choice (e.g. 7zip, WinZIP, etc.). Next, open the folder that you see when the compressed folder is opened (typically it will have the same name as the compressed folded, without the `*.tar.gz` extension), and extract it's contents inside the folder `training_demo\pre-trained-model`.

Now that we have downloaded and extracted our pre-trained model, let's have a look at the changes that we shall need to apply to the downloaded `*.config` file (highlighted in yellow):

```
# SSD with Inception v2 configuration for MSCOCO 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: 1 # Set this to the number of different label 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 {
```

(continues on next page)

(continued from previous page)

```
            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
                reduce_boxes_in_lowest_layer: true
            }
        }
        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: 3
                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
                        }
                    }
                }
            }
        }
        feature_extractor {
            type: 'ssd_inception_v2' # Set to the name of your chosen pre-trained␣
↪model
            min_depth: 16
            depth_multiplier: 1.0
            conv_hyperparams {
                activation: RELU_6,
                regularizer {
                    l2_regularizer {
```

(continues on next page)

(continued from previous page)

```
                        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,
                }
            }
            override_base_feature_extractor_hyperparams: true
        }
        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: 12 # Increase/Decrease this value depending on the available memory (Higher values require more memory and vice-versa)
    optimizer {
        rms_prop_optimizer: {
            learning_rate: {
```

(continues on next page)

(continued from previous page)

```
                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: "pre-trained-model/model.ckpt" # Path to extracted files of pre-trained model
    from_detection_checkpoint: 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: "annotations/train.record" # Path to training TFRecord file
    }
    label_map_path: "annotations/label_map.pbtxt" # Path to label map file
}

eval_config: {
    # (Optional): Uncomment the line below if you installed the Coco evaluation tools
    # and you want to also run evaluation
    # metrics_set: "coco_detection_metrics"
    # (Optional): Set this to the number of images in your <PATH_TO_IMAGES_FOLDER>/train
    # if you want to also run evaluation
    num_examples: 8000
    # Note: The below line limits the evaluation process to 10 evaluations.
    # Remove the below line to evaluate indefinitely.
    max_evals: 10
}

eval_input_reader: {
    tf_record_input_reader {
        input_path: "annotations/test.record" # Path to testing TFRecord
    }
    label_map_path: "annotations/label_map.pbtxt" # Path to label map file
    shuffle: false
    num_readers: 1
}
```

It is worth noting here that the changes to lines `178` and `181` above are optional. These should only be used if you installed the COCO evaluation tools, as outlined in the *COCO API installation (Optional)* section, and you intend to run evaluation (see *Evaluating the Model (Optional)*).

Once the above changes have been applied to our config file, go ahead and save it under `training_demo/training`.

3.7 Training the Model

Standard

Note: This tab describes the training process using Tensorflow's new model training script, namely `model_main.py`, as suggested by the Tensorflow Object Detection docs. The advantage of using this script is that it interleaves training and evaluation, essentially combining the `train.py` and `eval.py` Legacy scripts.

If instead you would like to use the legacy `train.py` script, switch to the Legacy tab.

Before we begin training our model, let's go and copy the `TensorFlow/models/research/object_detection/model_main.py` script and paste it straight into our `training_demo` folder. We will need this script in order to train our model.

Now, to initiate a new training job, `cd` inside the `training_demo` folder and type the following:

```
python model_main.py --alsologtostderr --model_dir=training/ --pipeline_config_
↪path=training/ssd_inception_v2_coco.config
```

Once the training process has been initiated, you should see a series of print outs similar to the one below (plus/minus some warnings):

```
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:Restoring parameters from ssd_inception_v2_coco_2017_11_17/model.ckpt
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Saving checkpoints for 0 into training\model.ckpt.
INFO:tensorflow:loss = 16.100115, step = 0
...
```

Important: The output will normally look like it has "frozen" after the loss for step 0 has been logged, but DO NOT rush to cancel the process. The training outputs logs only every 100 steps by default, therefore if you wait for a while, you should see a log for the loss at step 100.

The time you should wait can vary greatly, depending on whether you are using a GPU and the chosen value for `batch_size` in the config file, so be patient.

Legacy

Before we begin training our model, let's go and copy the `TensorFlow/models/research/object_detection/legacy/train.py` script and paste it straight into our `training_demo` folder. We will need this script in order to train our model.

Now, to initiate a new training job, `cd` inside the `training_demo` folder and type the following:

```
python train.py --logtostderr --train_dir=training/ --pipeline_config_path=training/
↪ssd_inception_v2_coco.config
```

Once the training process has been initiated, you should see a series of print outs similar to the one below (plus/minus some warnings):

```
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:depth of additional conv before box predictor: 0
INFO:tensorflow:Restoring parameters from ssd_inception_v2_coco_2017_11_17/model.ckpt
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Starting Session.
INFO:tensorflow:Saving checkpoint to path training\model.ckpt
INFO:tensorflow:Starting Queues.
INFO:tensorflow:global_step/sec: 0
INFO:tensorflow:global step 1: loss = 13.8886 (12.339 sec/step)
INFO:tensorflow:global step 2: loss = 16.2202 (0.937 sec/step)
INFO:tensorflow:global step 3: loss = 13.7876 (0.904 sec/step)
INFO:tensorflow:global step 4: loss = 12.9230 (0.894 sec/step)
INFO:tensorflow:global step 5: loss = 12.7497 (0.922 sec/step)
INFO:tensorflow:global step 6: loss = 11.7563 (0.936 sec/step)
INFO:tensorflow:global step 7: loss = 11.7245 (0.910 sec/step)
INFO:tensorflow:global step 8: loss = 10.7993 (0.916 sec/step)
INFO:tensorflow:global step 9: loss = 9.1277 (0.890 sec/step)
INFO:tensorflow:global step 10: loss = 9.3972 (0.919 sec/step)
INFO:tensorflow:global step 11: loss = 9.9487 (0.897 sec/step)
INFO:tensorflow:global step 12: loss = 8.7954 (0.884 sec/step)
INFO:tensorflow:global step 13: loss = 7.4329 (0.906 sec/step)
INFO:tensorflow:global step 14: loss = 7.8270 (0.897 sec/step)
INFO:tensorflow:global step 15: loss = 6.4877 (0.894 sec/step)
...
```

If you ARE observing a similar output to the above, then CONGRATULATIONS, you have successfully started your first training job. Now you may very well treat yourself to a cold beer, as waiting on the training to finish is likely to take a while. Following what people have said online, it seems that it is advisable to allow you model to reach a `TotalLoss` of at least 2 (ideally 1 and lower) if you want to achieve "fair" detection results. Obviously, lower `TotalLoss` is better, however very low `TotalLoss` should be avoided, as the model may end up overfitting the dataset, meaning that it will perform poorly when applied to images outside the dataset. To monitor `TotalLoss`, as well as a number of other metrics, while your model is training, have a look at *Monitor Training Job Progress using TensorBoard*.

If you ARE NOT seeing a print-out similar to that shown above, and/or the training job crashes after a few seconds, then have a look at the issues and proposed solutions, under the *Common issues* section, to see if you can find a solution. Alternatively, you can try the issues section of the official Tensorflow Models repo.

Note: Training times can be affected by a number of factors such as:

- The computational power of you hardware (either CPU or GPU): Obviously, the more powerful your PC is, the faster the training process.
- Whether you are using the TensorFlow CPU or GPU variant: In general, even when compared to the best CPUs, almost any GPU graphics card will yield much faster training and detection speeds. As a matter of fact, when I

first started I was running TensorFlow on my *Intel i7-5930k* (6/12 cores @ 4GHz, 32GB RAM) and was getting step times of around *12 sec/step*, after which I installed TensorFlow GPU and training the very same model -using the same dataset and config files- on a *EVGA GTX-770* (1536 CUDA-cores @ 1GHz, 2GB VRAM) I was down to *0.9 sec/step*!!! A 12-fold increase in speed, using a "low/mid-end" graphics card, when compared to a "mid/high-end" CPU.

- How big the dataset is: The higher the number of images in your dataset, the longer it will take for the model to reach satisfactory levels of detection performance.
- The complexity of the objects you are trying to detect: Obviously, if your objective is to track a black ball over a white background, the model will converge to satisfactory levels of detection pretty quickly. If on the other hand, for example, you wish to detect ships in ports, using Pan-Tilt-Zoom cameras, then training will be a much more challenging and time-consuming process, due to the high variability of the shape and size of ships, combined with a highly dynamic background.
- And many, many, many, more. . . .

3.8 Evaluating the Model (Optional)

By default, the training process logs some basic measures of training performance. These seem to change depending on the installed version of Tensorflow and the script used for training (i.e. `model_main.py` (Standard) or `train.py` (Legacy)).

As you will have seen in various parts of this tutorial, we have mentioned a few times the optional utilisation of the COCO evaluation metrics. Also, under section _image_partitioning_sec we partitioned our dataset in two parts, where one was to be used for training and the other for evaluation. In this section we will look at how we can use these metrics, along with the test images, to get a sense of the performance achieved by our model as it is being trained.

Firstly, let's start with a brief explanation of what the evaluation process does. While the training process runs, it will occasionally create checkpoint files inside the `training_demo/training` folder, which correspond to snapshots of the model at given steps. When a set of such new checkpoint files is generated, the evaluation process uses these files and evaluates how well the model performs in detecting objects in the test dataset. The results of this evaluation are summarised in the form of some metrics, which can be examined over time.

The steps to run the evaluation are outlined below:

1. Firstly we need to download and install the metrics we want to use.

- For a description of the supported object detection evaluation metrics, see here.
- The process of installing the COCO evaluation metrics is described in *COCO API installation (Optional)*.

2. Secondly, we must modify the configuration pipeline (`*.config` script).

- See lines 178 and 181 of the script in *Configuring a Training Pipeline*.

3. The third step depends on what method (script) was used when staring the training in *Training the Model*. See below for details:

    Standard

    The `model_main.py` script interleaves training and evaluation. Therefore, assuming that the following two steps were followed correctly, nothing else needs to be done.

    Legacy

    When using the Legacy scripts, evaluation is run using the `eval.py` script. This is done as follows:

    - Copy the `TensorFlow/models/research/object_detection/legacy/eval.py` script and paste it inside the `training_demo` folder.

- Now, to initiate an evaluation job, `cd` inside the `training_demo` folder and type the following:

```
python eval.py --logtostderr --pipeline_config_path=training/
↪ssd_inception_v2_coco.config --checkpoint_dir=training/ --
↪eval_dir=training/eval_0
```

While the evaluation process is running, it will periodically (every 300 sec by default) check and use the latest `training/model.ckpt-*` checkpoint files to evaluate the performance of the model. The results are stored in the form of tf event files (`events.out.tfevents.*`) inside `training/eval_0`. These files can then be used to monitor the computed metrics, using the process described by the next section.

3.9 Monitor Training Job Progress using TensorBoard

A very nice feature of TensorFlow, is that it allows you to coninuously monitor and visualise a number of different training/evaluation metrics, while your model is being trained. The specific tool that allows us to do all that is Tensorboard.

To start a new TensorBoard server, we follow the following steps:

- Open a new *Anaconda/Command Prompt*
- Activate your TensorFlow conda environment (if you have one), e.g.:

```
activate tensorflow_gpu
```

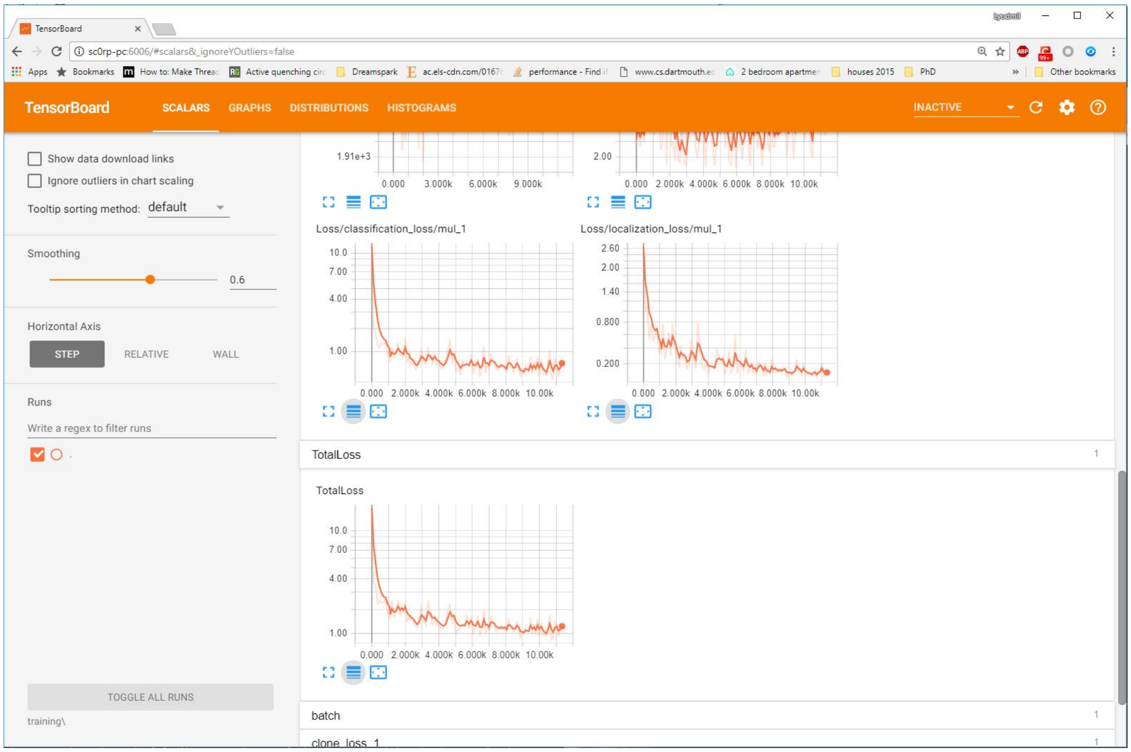
- `cd` into the `training_demo` folder.
- Run the following command:

```
tensorboard --logdir=training\
```

The above command will start a new TensorBoard server, which (by default) listens to port 6006 of your machine. Assuming that everything went well, you should see a print-out similar to the one below (plus/minus some warnings):

```
TensorBoard 1.6.0 at http://YOUR-PC:6006 (Press CTRL+C to quit)
```

Once this is done, go to your browser and type `http://YOUR-PC:6006` in your address bar, following which you should be presented with a dashboard similar to the one shown below (maybe less populated if your model has just started training):



3.10 Exporting a Trained Inference Graph

Once your training job is complete, you need to extract the newly trained inference graph, which will be later used to perform the object detection. This can be done as follows:

- Open a new *Anaconda/Command Prompt*
- Activate your TensorFlow conda environment (if you have one), e.g.:

```
activate tensorflow_gpu
```

- Copy the `TensorFlow/models/research/object_detection/export_inference_graph.py` script and paste it straight into your `training_demo` folder.
- Check inside your `training_demo/training` folder for the `model.ckpt-*` checkpoint file with the highest number following the name of the dash e.g. `model.ckpt-34350`). This number represents the training step index at which the file was created.
- Alternatively, simply sort all the files inside `training_demo/training` by descending time and pick the `model.ckpt-*` file that comes first in the list.
- Make a note of the file's name, as it will be passed as an argument when we call the `export_inference_graph.py` script.
- Now, `cd` inside your `training_demo` folder, and run the following command:

```
python export_inference_graph.py --input_type image_tensor --pipeline_config_path␣
↪training/ssd_inception_v2_coco.config --trained_checkpoint_prefix training/model.
↪ckpt-13302 --output_directory trained-inference-graphs/output_inference_graph_v1.pb
```

CHAPTER

FOUR

COMMON ISSUES

Below is a list of common issues encountered while using TensorFlow for objects detection.

4.1 Python crashes - TensorFlow GPU

If you are using *TensorFlow GPU* and when you try to run some Python object detection script (e.g. *Test your Installation*), after a few seconds, Windows reports that Python has crashed then have a look at the *Anaconda/Command Prompt* window you used to run the script and check for a line similar (maybe identical) to the one below:

```
2018-03-22 03:07:54.623130: E C:\tf_jenkins\workspace\rel-win\M\windows-gpu\
↪PY\36\tensorflow\stream_executor\cuda\cuda_dnn.cc:378] Loaded runtime␣
↪CuDNN library: 7101 (compatibility version 7100) but source was compiled␣
↪with 7003 (compatibility version 7000).  If using a binary install,␣
↪upgrade your CuDNN library to match.  If building from sources, make sure␣
↪the library loaded at runtime matches a compatible version specified␣
↪during compile configuration.
```

If the above line is present in the printed debugging, it means that you have not installed the correct version of the cuDNN libraries. In this case make sure you re-do the *Install CUDNN* step, making sure you instal cuDNN v7.0.5.

4.2 Cleaning up Nvidia containers (TensorFlow GPU)

Sometimes, when terminating a TensorFlow training process, the Nvidia containers associated to the process are not cleanly terminated. This can lead to bogus errors when we try to run a new TensorFlow process.

Some known issues caused by the above are presented below:

- Failure to restart training of a model. Look for the following errors in the debugging:

```
2018-03-23 03:03:10.326902: E C:\tf_jenkins\workspace\rel-win\M\windows-
↪gpu\PY\36\tensorflow\stream_executor\cuda\cuda_dnn.cc:385] could not␣
↪create cudnn handle: CUDNN_STATUS_ALLOC_FAILED
2018-03-23 03:03:10.330475: E C:\tf_jenkins\workspace\rel-win\M\windows-
↪gpu\PY\36\tensorflow\stream_executor\cuda\cuda_dnn.cc:352] could not␣
↪destroy cudnn handle: CUDNN_STATUS_BAD_PARAM
2018-03-23 03:03:10.333797: W C:\tf_jenkins\workspace\rel-win\M\windows-
↪gpu\PY\36\tensorflow/stream_executor/stream.h:1983] attempting to␣
↪perform DNN operation using StreamExecutor without DNN support
2018-03-23 03:03:10.333807: I C:\tf_jenkins\workspace\rel-win\M\windows-
↪gpu\PY\36\tensorflow\stream_executor\stream.cc:1851] stream␣
↪00000216F05CB660 did not wait for stream: 00000216F05CA6E0
```

(continues on next page)

(continued from previous page)

```
2018-03-23 03:03:10.340765: I C:\tf_jenkins\workspace\rel-win\M\windows-
↪gpu\PY\36\tensorflow\stream_executor\stream.cc:4637] stream␣
↪00000216F05CB660 did not memcpy host-to-device; source: 000000020DB37B00
2018-03-23 03:03:10.343752: F C:\tf_jenkins\workspace\rel-win\M\windows-
↪gpu\PY\36\tensorflow\core\common_runtime\gpu\gpu_util.cc:343] CPU->GPU␣
↪Memcpy failed
```

To solve such issues in Windows, open a *Task Manager* windows, look for Tasks with name `NVIDIA Container` and kill them by selecting them and clicking the *End Task* button at the bottom left corner of the window.

If the issue persists, then you're probably running out of memory. Try closing down anything else that might be eating up your GPU memory (e.g. Youtube videos, webpages etc.)

4.3 labelImg saves annotation files with `.xml.xml` extension

At the time of writing up this document, I haven't managed to identify why this might be happening. I have joined a GitHub issue, at which you can refer in case there are any updates.

One way I managed to fix the issue was by clicking on the "Change Save Dir" button and selecting the directory where the annotations files should be stores. By doing so, you should not longer get a pop-up dialog when you click "Save" (or Ctrl+s), but you can always check if the file was saved by looking at the bottom left corner of `labelImg`.

4.4 "WARNING:tensorflow:Entity `<bound method X of <Y>>` could not be transformed ..."

In some versions of Tensorflow, you may see errors that look similar to the ones below:

```
...
WARNING:tensorflow:Entity <bound method Conv.call of <tensorflow.python.layers.
↪convolutional.Conv2D object at 0x000001E92103EDD8>> could not be transformed and␣
↪will be executed as-is. Please report this to the AutgoGraph team. When filing the␣
↪bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VERBOSITY=10`) and attach␣
↪the full output. Cause: converting <bound method Conv.call of <tensorflow.python.
↪layers.convolutional.Conv2D object at 0x000001E92103EDD8>>: AssertionError: Bad␣
↪argument number for Name: 3, expecting 4
WARNING:tensorflow:Entity <bound method BatchNormalization.call of <tensorflow.python.
↪layers.normalization.BatchNormalization object at 0x000001E9225EBA90>> could not be␣
↪transformed and will be executed as-is. Please report this to the AutgoGraph team.␣
↪When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_
↪VERBOSITY=10`) and attach the full output. Cause: converting <bound method␣
↪BatchNormalization.call of <tensorflow.python.layers.normalization.
↪BatchNormalization object at 0x000001E9225EBA90>>: AssertionError: Bad argument␣
↪number for Name: 3, expecting 4
...
```

These warnings appear to be harmless form my experience, however they can saturate the console with unnecessary messages, which makes it hard to scroll through the output of the training/evaluation process.

As reported here, this issue seems to be caused by a mismatched version of gast. Simply downgrading gast to version `0.2.2` seems to remove the warnings. This can be done by running:

```
pip install gast==0.2.2
```

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search