PROJECT REPORT

ON

"AUTOMATIC DETECTION OF BIKE RIDER DRIVING WITHOUT HELMET"

Submitted in partial fulfillment of the requirements for the award of the Degree of

Bachelor of Technology

In

Computer Science and Engineering

(2015-2019)

Submitted by

Prahlad Singh Negi

150180101037

Under the guidance of

Dr. Vishal Kumar

Assistant Professor

Department of Computer Science and Engineering

BIPIN TRIPATHI KUMAON INSTITUTE OF TECHNOLOGY

DWARAHAT UTTARAKHAND

DECLARATION

I declare that the project work with the title "**Automatic Detection of Bike Rider driving without Helmet**" is my own work done under **Dr. Vishal Kumar**, Bipin Tripathi Kumaon Institute of Technology, Dwarahat. I have gone through the rules of report writing provided by the institute and have followed all the instructions accordingly. This project work is being submitted in the fulfillment of the requirements for the partial degree of Bachelor of Technology in Computer Science and Engineering at Bipin Tripathi Kumaon Institute of Technology, Dwarahat for the academic session 2015– 2019. All the performed experiments done under this projects and written in this report are properly performed by my own and has not been imitated from any other sources. If there is any reference made for theoretical purpose then due recognition is mentioned for the source of original publication.

Prahlad Singh Negi
150180101037

CERTIFICATE

This is to certify that the project entitled "**Automatic Detection of Bike Rider driving without Helmet**" is submitted by Mr. Prahlad Singh Negi to the department of Computer Science and Engineering, B.T.KIT DWARAHAT, Uttarakhand in partial fulfillment of the requirements of Bachelor of Technology in Computer Science and Engineering during final year. It is authentic and unique record of work carried out under our supervision and guidance and the matter embodied in this project is original and has not been submitted for the award of any other degree.

We wish them success for their future endeavors.

Project Guide & Project Head

Dr. Vishal Kumar

ACKNOWLEDGEMENT

It would be my immense pleasure to take this as a chance to represent my appreciation and earnest because of our regarded Dr. Vishal Kumar for the direction, knowledge, and bolster that he has given all through this work. My present work would never have been achievable without his guidance, inputs and tutoring.

I would also like to thank every one of my friends, faculty team and all staff of prestigious Department of Computer Science and Engineering, B.T.K.I.T. Dwarahat for their amazing help all through our courses I learn at this organization.

Contents

ABSTRACT

Motorcycle accidents are growing throughout the years in all the countries, as there is difference in social, economical and the transport conditions differs from place to place. Mototrcycle is one of the prominent means of transport used by middle class people. Wearing helmet is the main safety equipment of motorcyclists, which might not be followed by all drivers. Accident of a motorcyclist is serious issue. This project aims at prevention of accidents by automatically identifying the drivers wearing helmet or not. For this, a Faster R-CNN descriptor for features extraction is used. Based on FasterR-CNN feature Extraction and Haar Cascade Classier real time images captured by cameras are used. The best result obtained from classification was with an accuracy rate of 0.995, and the best result obtained from helmet detection is with an accuracy rate of 0.96.

1.Introduction

1.1 Background

Two-wheeler is a very popular mode of transportation in almost every country. However, there is a high risk involved because of less protection. To reduce the involved risk, it is highly desirable for bike-riders to use helmet. Observing the usefulness of helmet, Governments have made it a punishable offense to ride a bike without helmet and have adopted manual strategies to catch the violators. However, the existing video surveillance based methods are passive and require significant human assistance. In general, such systems are infeasible due to involvement of humans, whose efficiency decreases over long duration. Automation of this process is highly desirable for reliable and robust monitoring of these violations as well as it also significantly reduces the amount of human resources needed. Also, many countries are adopting systems involving surveillance cameras at public places. So, the solution for detecting violators using the existing infrastructure is also cost-effective.

1.2 Objective

The aim of the project is to identify and detect a person riding a two-wheeler, a helmet and the number plate using Tensorflow. All these processes are done with the help of OpenCV, Anaconda, Jupyter, Python and numerous libraries.

1.3 Procedure

- Install TensorFlow-GPU
- Set up Object Detection directory structure and Anaconda Virtual Environment
- Gather and label pictures
- Generate training data
- Create label map and configure training
- Train object detector
- Test it out!

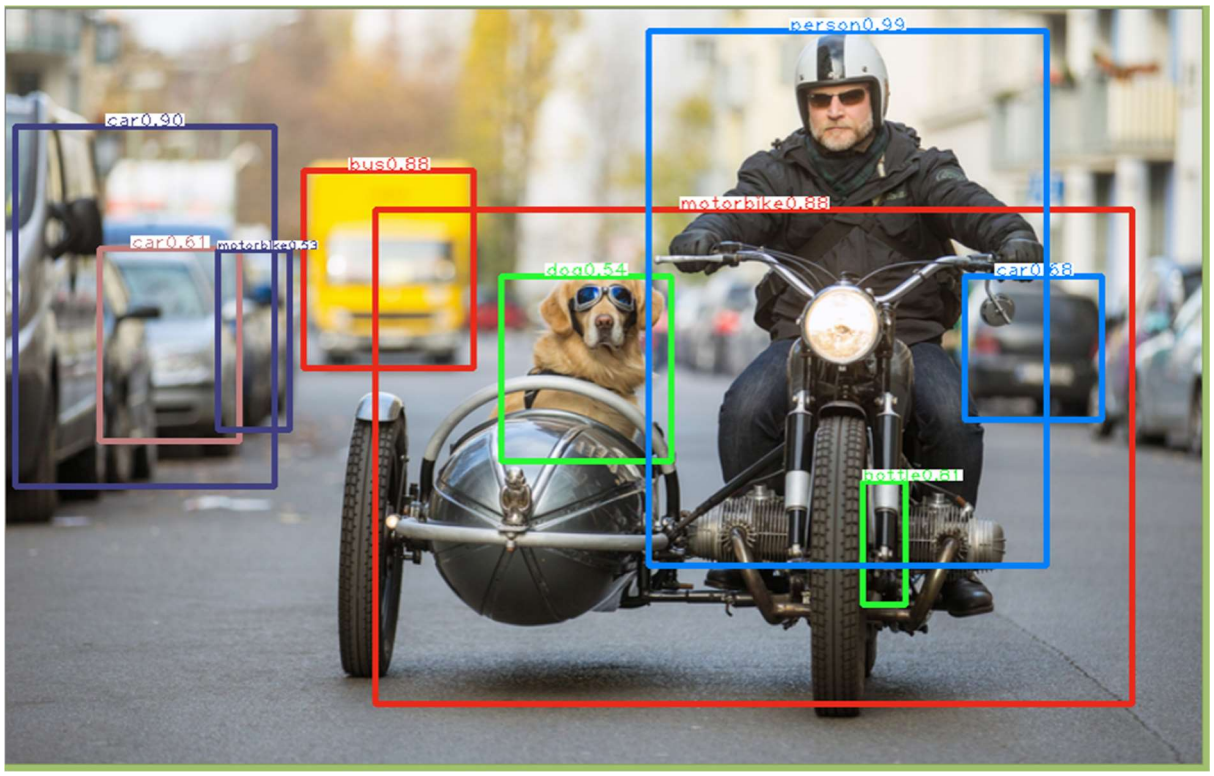
2. Overview of Object Detection

Object detection is a computer technology related to computer vision and image processing that deals with detecting instances of semantic objects of a certain class (such as humans, buildings, or cars) in digital images and videos. Well-researched domains of object detection include face detection and pedestrian detection. Object detection has applications in many areas of computer vision, including image retrieval and video surveillance.

It is widely used in computer vision task such as face detection, face recognition, video object co-segmentation. It is also used in tracking objects, for example tracking a ball during a football match, tracking movement of a cricket bat, tracking a person in a video.

2.1 Concept

Every object class has its own special features that helps in classifying the class – for example all circles are round. Object class detection uses these special features. For example, when looking for circles, objects that are at a particular distance from a point (i.e. the center) are sought. Similarly, when looking for squares, objects that are perpendicular at corners and have equal side lengths are needed. A similar approach is used for face identification where eyes, nose, and lips can be found and features like skin color and distance between eyes can be found.



2.2 Methods

Methods for object detection generally fall into either machine learning-based approaches or deep learning-based approaches. For Machine Learning approaches, it becomes necessary to first define features using one of the methods below, then using a technique such as support vector machine (SVM) to do the classification. On the other hand, deep learning techniques that are able to do end-to-end object detection without specifically defining features, and are typically based on convolutional neural networks (CNN).

2.2.1 Machine Learning approaches:

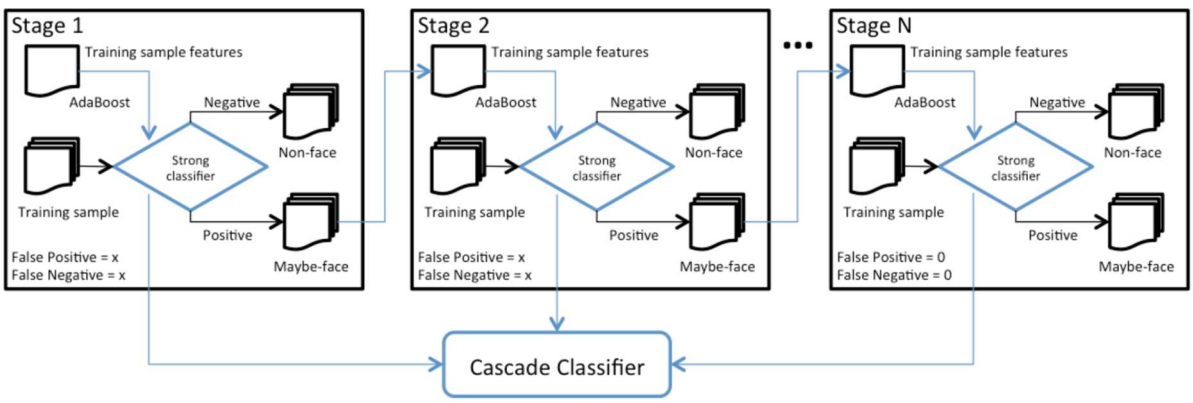
2.2.1.1 Viola–Jones object detection framework based on Haar features

The **Viola–Jones object detection framework** is the first object detection framework to provide competitive object detection rates in real-time proposed in 2001 by Paul Viola and Michael Jones. Although it can be trained to detect a variety of object classes, it was motivated primarily by the problem of face detection.

Feature types and evaluation

The characteristics of Viola–Jones algorithm which make it a good detection algorithm are:

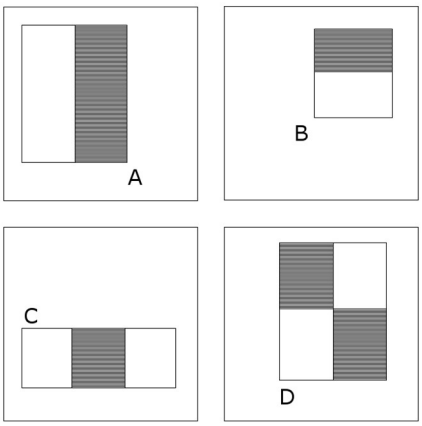
- Robust – very high detection rate (true-positive rate) & very low false-positive rate always.
- Real time – For practical applications at least 2 frames per second must be processed.
- Face detection only (not recognition) - The goal is to distinguish faces from non-faces (detection is the first step in the recognition process).



The algorithm has four stages:

1. Haar Feature Selection
2. Creating an Integral Image
3. Adaboost Training
4. Cascading Classifiers

The features sought by the detection framework universally involve the sums of image pixels within rectangular areas. As such, they bear some resemblance to Haar basis functions, which have been used previously in the realm of image-based object detection.[3] However, since the features used by Viola and Jones all rely on more than one rectangular area, they are generally more complex. The figure on the right illustrates the four different types of features used in the framework. The value of any given feature is the sum of the pixels within clear rectangles subtracted from the sum of the pixels within shaded rectangles. Rectangular features of this sort are primitive when compared to alternatives such as steerable filters. Although they are sensitive to vertical and horizontal features, their feedback is considerably coarser.



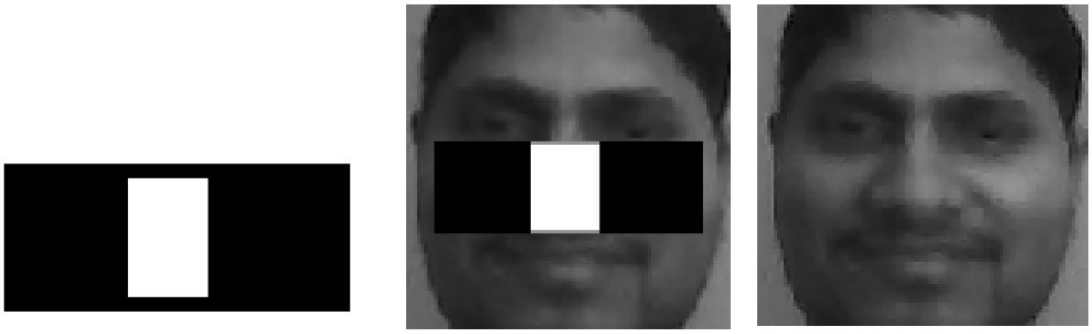
Example rectangle features shown relative to the enclosing detection window

Haar Features

All human faces share some similar properties. These regularities may be matched using **Haar Features**.

A few properties common to human faces:

- The eye region is darker than the upper-cheeks.
- The nose bridge region is brighter than the eyes.



Haar Feature that looks similar to the bridge of the nose is applied onto the face

Composition of properties forming matchable facial features:

- Location and size: eyes, mouth, bridge of nose
- Value: oriented gradients of pixel intensities

The four features matched by this algorithm are then sought in the image of a face (shown at right).

Haar Feature that looks similar to the eye region which is darker than the upper cheeks is applied onto a face

Rectangle features:

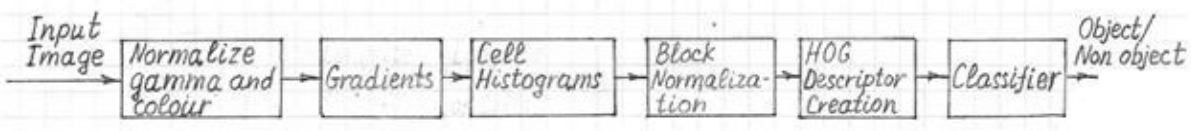
- Value = Σ (pixels in black area) - Σ (pixels in white area)
- Three types: two-, three-, four-rectangles, Viola & Jones used two-rectangle features
- For example: the difference in brightness between the white &black rectangles over a specific area
- Each feature is related to a special location in the sub-window

3rd and 4th kind of Haar Feature

2.2.1.2 histogram of oriented gradients (HOG) features

The **histogram of oriented gradients (HOG)** is a feature descriptor used in computer vision and image processing for the purpose of object detection. The technique counts occurrences of gradient orientation in localized portions of an image. This method is similar to that of edge orientation histograms, scale-invariant feature transform descriptors, and shape contexts, but differs in that it is computed on a dense grid of uniformly spaced cells and uses overlapping local contrast normalization for improved accuracy.

Histogram of Oriented Gradients (HOG) is a feature descriptor widely employed on several domains to characterize objects through their shapes. Local object appearance and shape can often be described by the distribution of local intensity gradients or edge directions.



HOG is widely utilized as a feature described image region for object detection such as human face or human body detection. To increase the efficiency of the object searching, gamma and colors of the image should be normalized. The object search is based on the detection technique applied for the small images defined by sliding detector window that probes region by region of the original input image and its scaled versions.

The first step in HOG detection is to divide the source image into blocks (for example 16×16 pixels). Each block is divided by small regions, called cells (for example 8×8 pixels). Usually blocks overlap each other, so that the same cell may be in several blocks. For each pixel within the cell the vertical and horizontal

gradients are obtained. The simplest method to do that is to use 1-D Sobel vertical and horizontal operators:

$$\mathbf{Gx(y,x) = Y(y,x+1) - Y(y,x-1); Gy(y,x) = Y(y+1,x) - Y(y-1,x)}$$

Y(y,x) is the pixel intensity at coordinates x and y. Gx(y,x) is the horizontal gradient, and Gy(y,x) is the vertical gradient. The magnitude and phase of the gradient are determined as

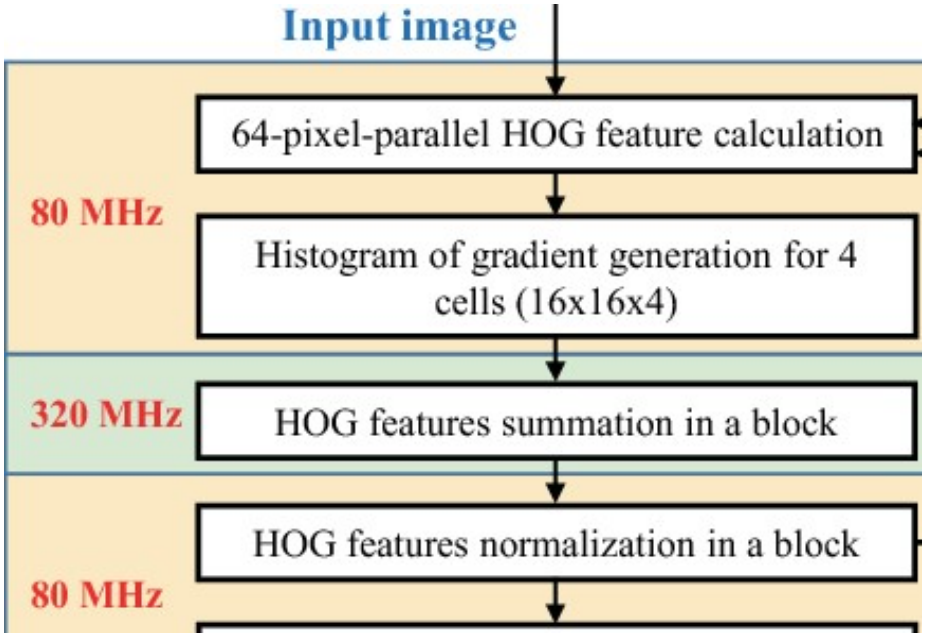
$$G(y,x) = \sqrt{G_x(y,x)^2 + G_y(y,x)^2}, \theta(y,x) = \arctan\left(\frac{G_y(y,x)}{G_x(y,x)}\right)$$

Next, the HOG is created for each cell. For the histogram, Q bins for the angle are chosen (for example Q=9). Usually unsigned orientation is used, so angles below 0o are increased by 180o.

Since different images may have different contrast, contrast normalization can be very useful. Normalization is done on the histogram vector v within a block. One of the following norms could be used

$$L1\text{-norm } norm = \frac{v}{\sqrt{|v|_2^2+\varepsilon^2}}; L2\text{-norm } norm = \frac{v}{|v|_2+\varepsilon}; L1\text{sqrt-norm } norm = \sqrt{\frac{v}{|v|_2+\varepsilon}}$$

A descriptor is assigned to each detector window. This descriptor consists of all the cell histograms for each block in the detector window. The detector window descriptor is used as information for object recognition. Training and testing happens using this descriptor. Many possible methods exist to classify objects using the descriptor such as SVM (support vector machine), neural networks, etc.



2.2.2 Deep Learning approaches:

2.2.2.1 Single Shot Multibox Detection

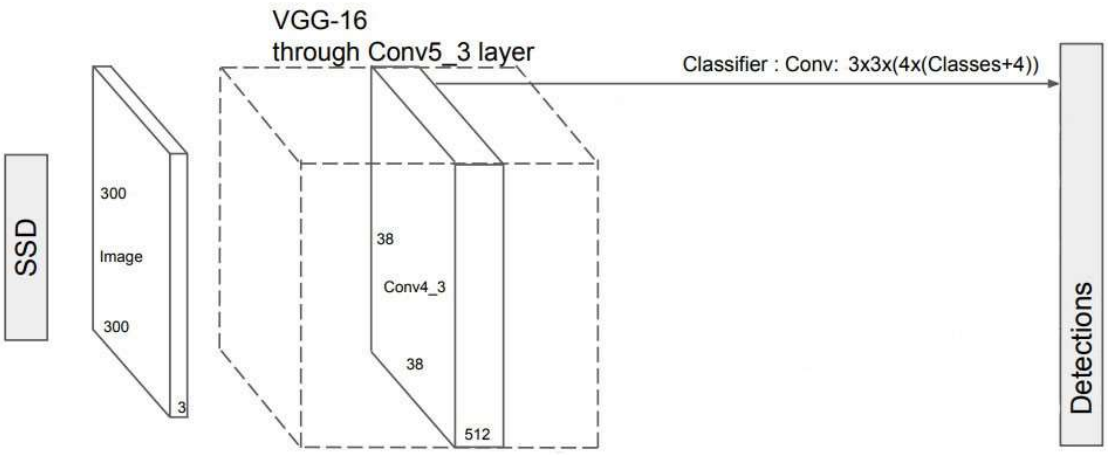
The paper about SSD: Single Shot MultiBox Detector (by C. Szegedy et al.) was released at the end of November 2016 and reached new records in terms of performance and precision for object detection tasks, scoring over 74% mAP (*mean Average Precision*) at 59 frames per second on standard datasets such as PascalVOC and COCO.

To better understand SSD, let's start by explaining where the name of this architecture comes from:

- **Single Shot:** this means that the tasks of object localization and classification are done in a *single forward pass* of the network
- **MultiBox:** this is the name of a technique for bounding box regression developed by Szegedy et al. (we will briefly cover it shortly)
- **Detector:** The network is an object detector that also classifies those detected objects

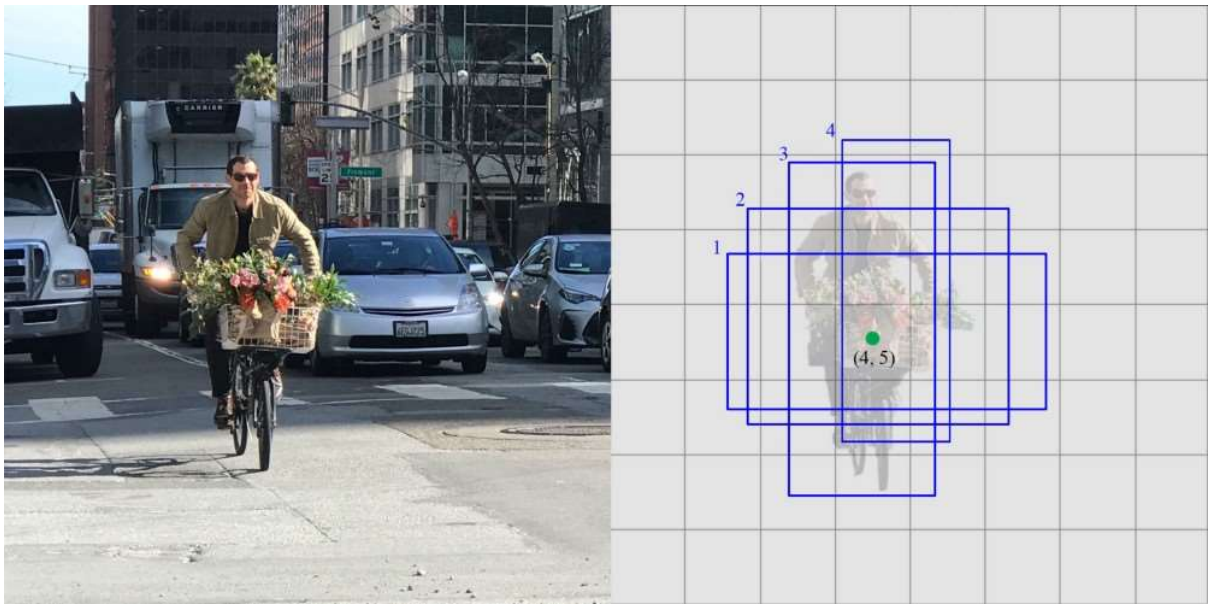
The SSD object detection composes of 2 parts:

- Extract feature maps, and
- Apply convolution filters to detect objects.



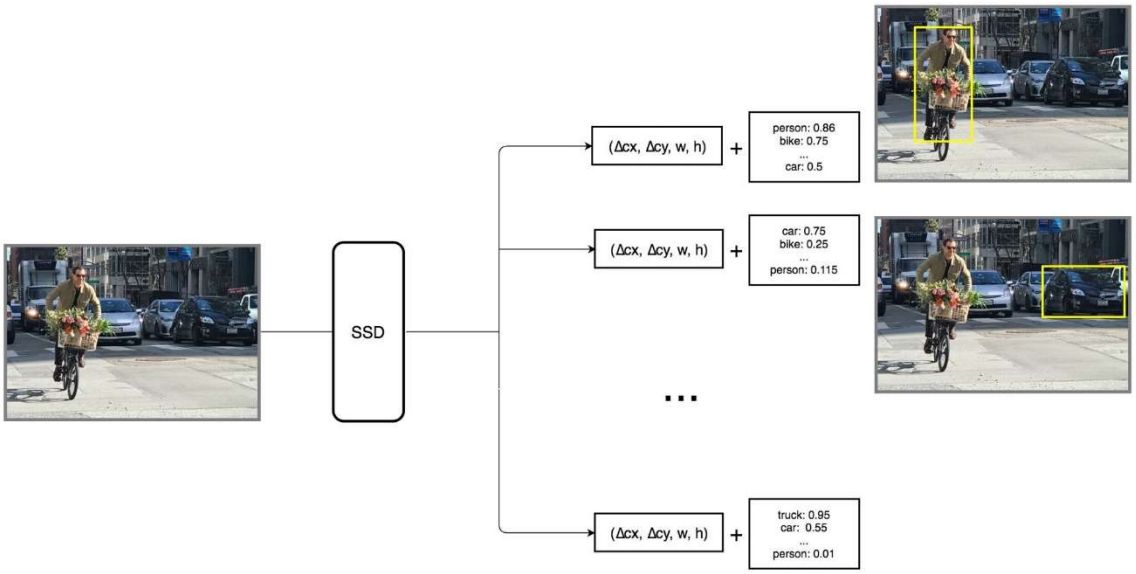
Modified from SSD: Single Shot MultiBox Detector

SSD uses **VGG16** to extract feature maps. Then it detects objects using the **Conv4_3** layer. For illustration, we draw the Conv4_3 to be 8 × 8 spatially (it should be 38 × 38). For each cell (also called location), it makes 4 object predictions.



Each prediction composes of a boundary box and 21 scores for each class (one extra class for no object), and we pick the highest score as the class for the bounded object. Conv4_3 makes a total of 38 × 38 × 4 predictions: four predictions per cell

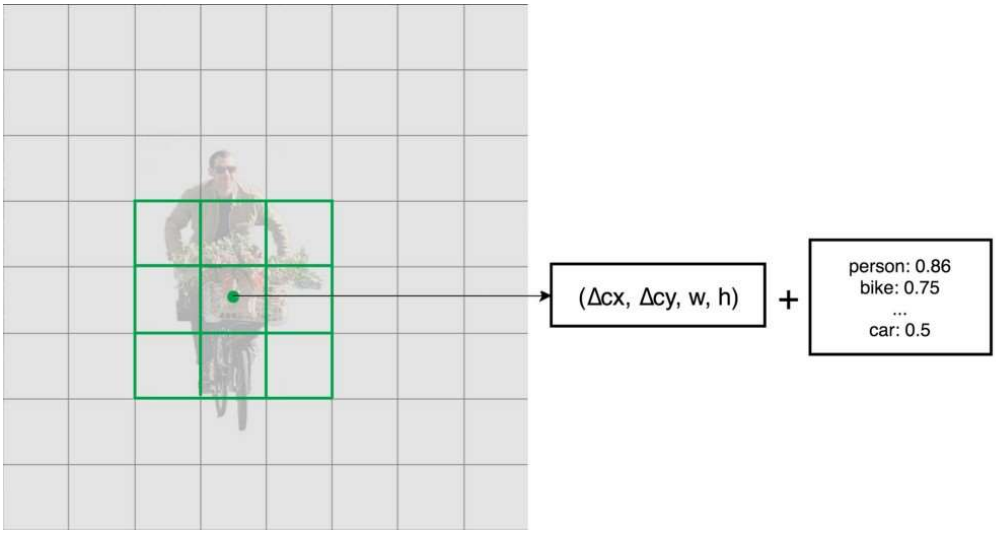
regardless of the depth of the feature maps. As expected, many predictions contain no object. SSD reserves a class "0" to indicate it has no objects.



Each prediction includes a boundary box and 21 scores for 21 classes (one class for no object).

Convolutional predictors for object detection

SSD does not use a delegated region proposal network. Instead, it resolves to a very simple method. It computes both the location and class scores using **small convolution filters**. After extracting the feature maps, SSD applies 3 × 3 convolution filters for each cell to make predictions. (These filters compute the results just like the regular CNN filters.) Each filter outputs 25 channels: 21 scores for each class plus one boundary box.



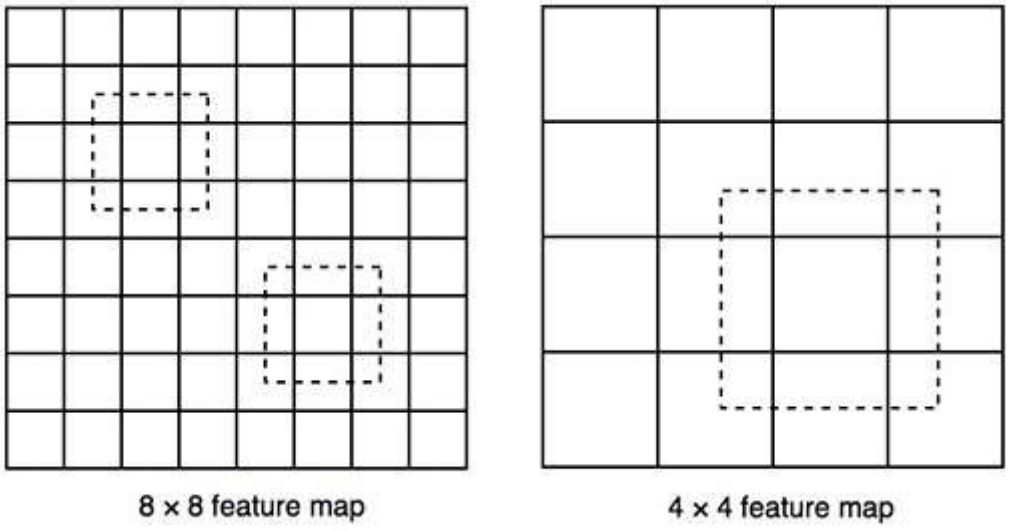
Apply a 3x3 convolution filter to make a prediction for the location and the class.

For example, in Conv4_3, we apply four 3 × 3 filters to map 512 input channels to 25 output channels.

$$(38 \times 38 \times 512) \xrightarrow{(4\times3\times3\times512\times(21+4))} (38 \times 38 \times 4 \times (21 + 4))$$

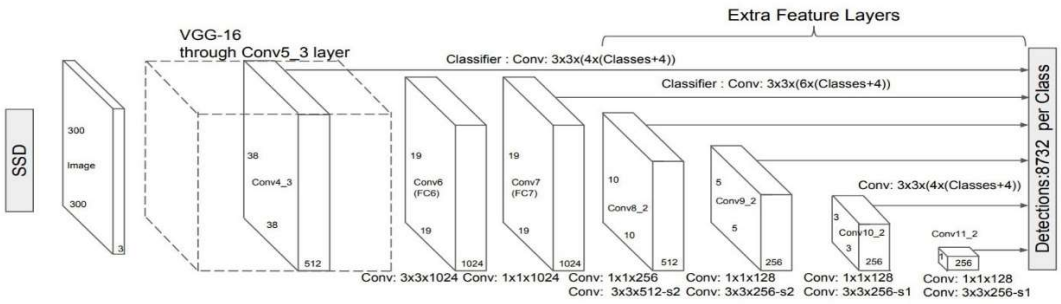
Multi-scale feature maps for detection

At first, we describe how SSD detects objects from a single layer. Actually, it uses multiple layers (**multi-scale feature maps)** to detect objects independently. As CNN reduces the spatial dimension gradually, the resolution of the feature maps also decrease. SSD uses lower resolution layers to detect larger scale objects. For example, the 4× 4 feature maps are used for larger scale object.



Lower resolution feature maps (right) detects larger scale objects.

SSD adds 6 more auxiliary convolution layers after the VGG16. Five of them will be added for object detection. In three of those layers, we make 6 predictions instead of 4. In total, SSD makes 8732 predictions using 6 layers.



SSD: Single Shot MultiBox Detector

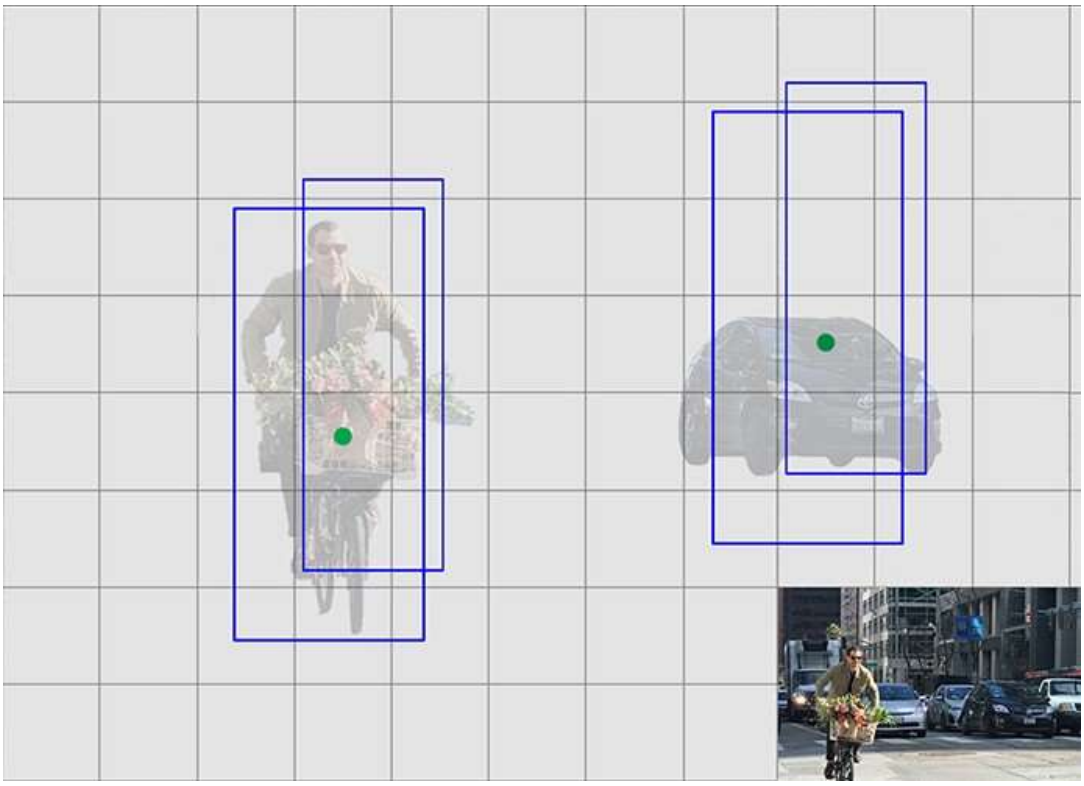
Multi-scale feature maps improve accuracy significantly. Here is the accuracy with different number of feature map layers used for object detection.

Prediction source layers from:						mAP use boundary boxes?		# Boxes
38 × 38	19 × 19	10 × 10	5 × 5	3 × 3	1 × 1	Yes	No	
✔	✔	✔	✔	✔	✔	74.3	63.4	8732
✔	✔	✔				70.7	69.2	9864
	✔					62.4	64.0	8664

Default boundary box

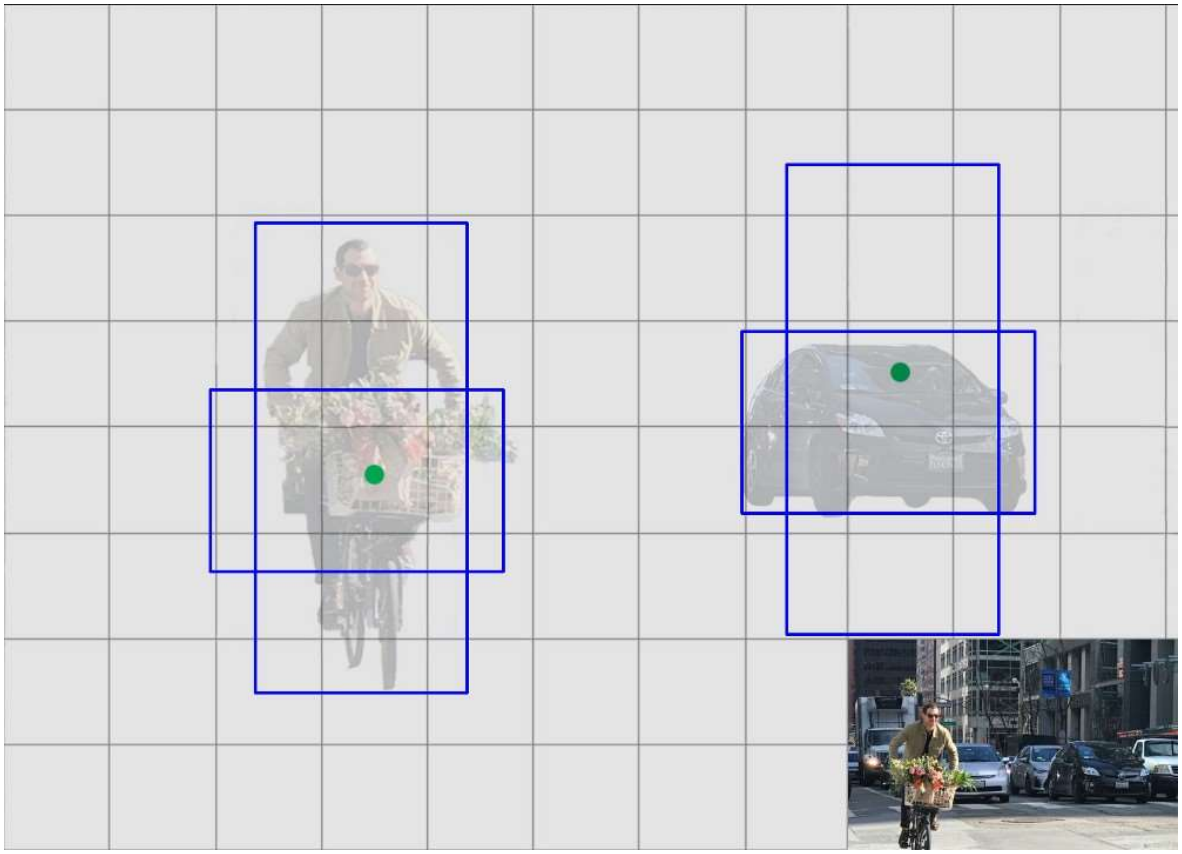
*The default boundary boxes are equivalent to **anchors i**n Faster R-CNN.*

How do we predict boundary boxes? Just like Deep Learning, we can start with random predictions and use gradient descent to optimize the model. However, during the initial training, the model may fight with each other to determine what shapes (pedestrians or cars) to be optimized for which predictions. Empirical results indicate early training can be very unstable. The boundary box predictions below work well for one category but not for others. We want our initial predictions to be diverse and not looking similar.



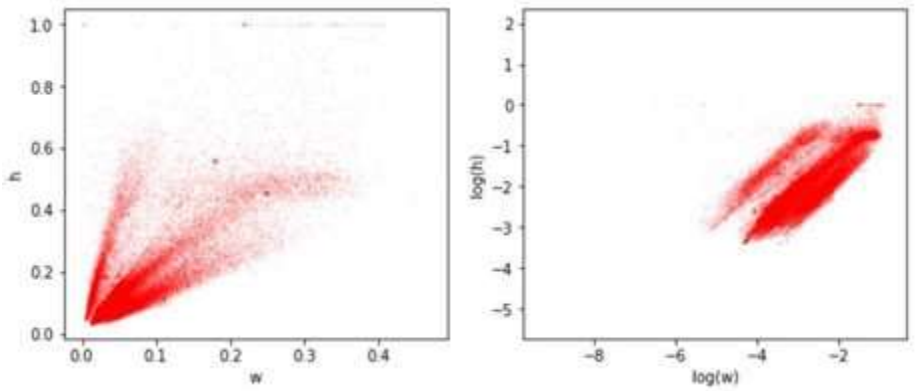
If predictions are not diverse, the model will not perform

If our predictions cover more shapes, like the one below, our model can detect more object types. This kind of head start makes training much easier and more stable.



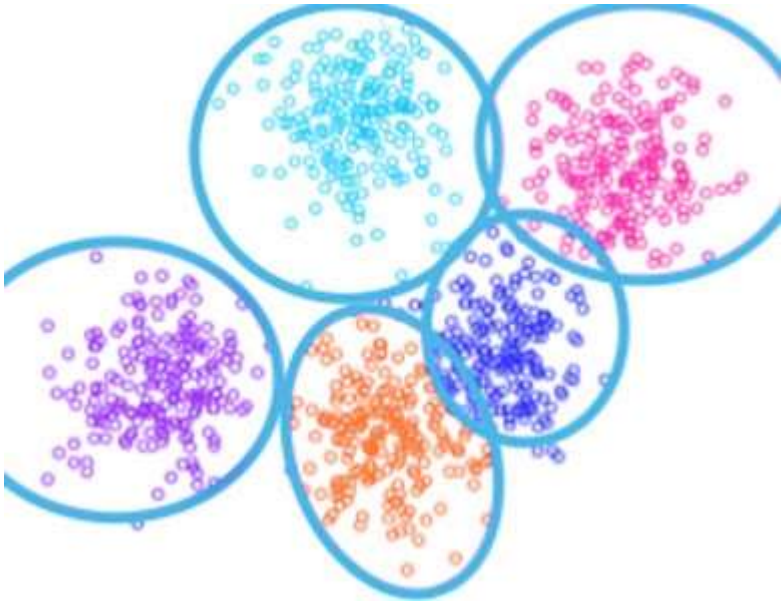
Divese predictions cover more object types.

In real-life, **boundary boxes do not have arbitrary shape** and size. Cars have similar shapes and pedestrians have an approximate aspect ratio of 0.41. In the KITTI dataset used in the autonomous driving, the width and height distributions for the boundary boxes are highly clustered.



Conceptually, the ground truth boundary boxes can be partitioned into clusters with each cluster represented by a **default boundary box** (the centroid of the

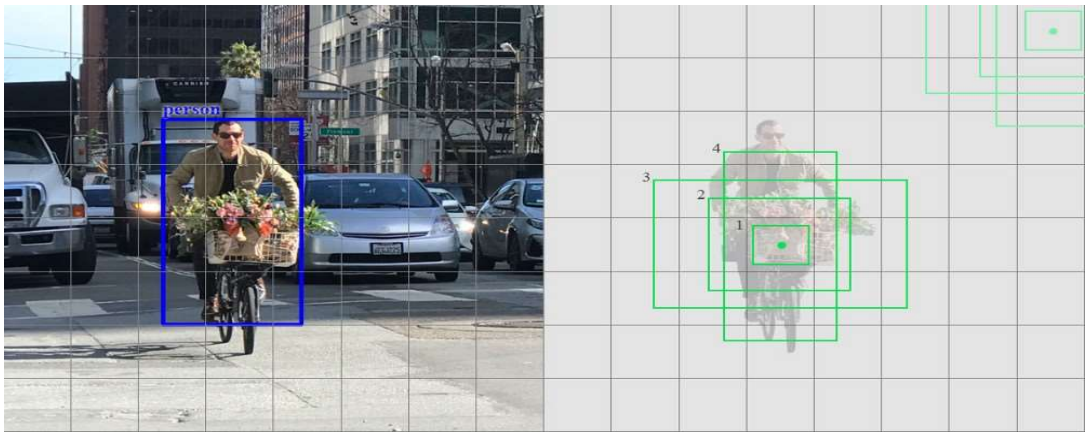
cluster). So, instead of making random guesses, we can start the guesses based on those default boxes.



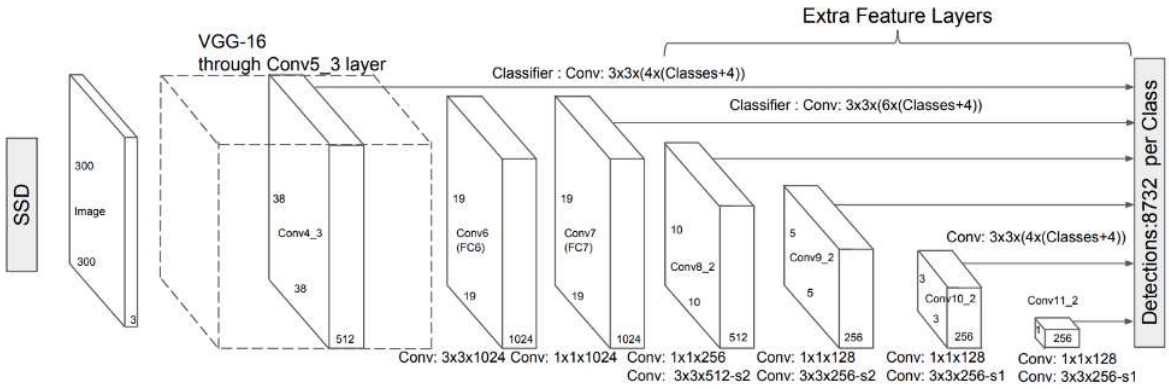
(Image modified form a k-means cluster)

To keep the complexity low, the default boxes are pre-selected manually and carefully to cover a wide spectrum of real life objects. SSD also keeps the default boxes to a minimum (4 or 6) with one prediction per default box. Now, instead of using a global coordination for the box location, the boundary box predictions are relative to the default boundary boxes at each cell (Δcx, Δcy, w, h), i.e. the offsets to the default box at each cell.

For each feature map layers, it shares the same set of default boxes centered at the corresponding cell. But different layers use different sets of default boxes to customize object detections at different resolutions. The 4 green boxes below illustrates 4 default boundary boxes.



Choosing default boundary boxes



Default boundary boxes are chosen manually. SSD defines a scale value for each feature map layer. Starting from the left, Conv4_3 detects objects at the smallest scale 0.2 (or 0.1 sometimes) and then increases linearly to the rightmost layer at a scale of 0.9. Combining the scale value with the target aspect ratios, we compute the width and the height of the default boxes. For layers making 6 predictions, SSD starts with 5 target aspect ratios: 1, 2, 3, 1/2 and 1/3. Then the width and the height of the default boxes are calculated as:

$$w = scale \cdot \sqrt{\text{aspect ratio}}$$

$$h = \frac{scale}{\sqrt{\text{aspect ratio}}}$$

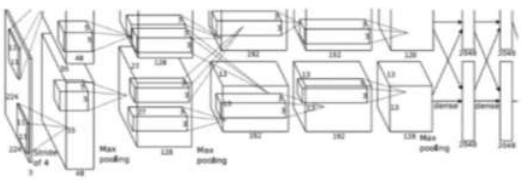
Then SSD adds an extra default box with scale:

$$scale = \sqrt{\text{scale} \cdot \text{scale at next level}}$$

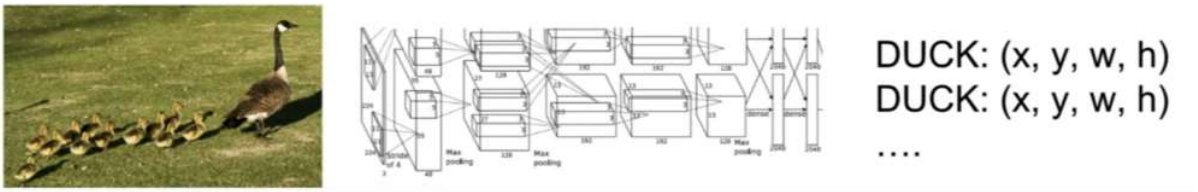
and aspect ratio = 1.

2.2.2.2 Region Proposals (R-CNN, Fast R-CNN, Faster R-CNN)

In this, algorithm try to draw a bounding box around the object of interest to locate it within the image. Also, you might not necessarily draw just one bounding box in an object detection case, there could be many bounding boxes representing different objects of interest within the image and you would not know how many beforehand.



CAT: (x, y, w, h)

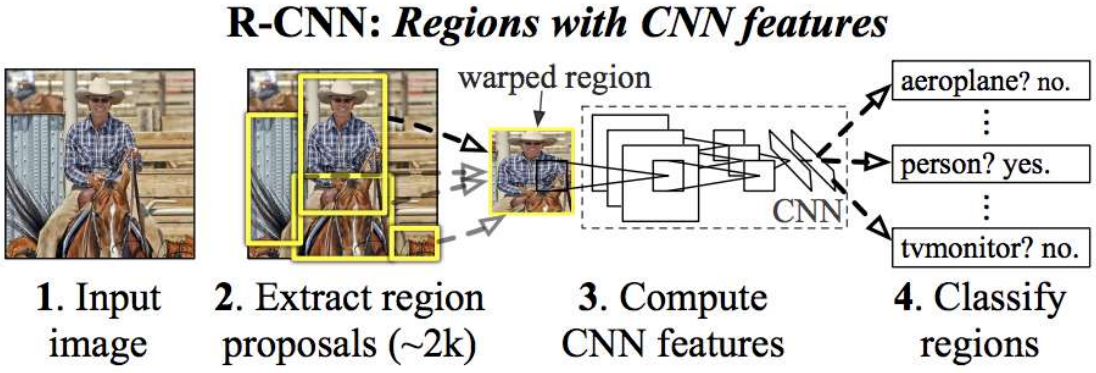


The major reason why you cannot proceed with this problem by building a standard convolutional network followed by a fully connected layer is that, the length of the output layer is variable—not constant, this is because the number of occurrences of the objects of interest is not fixed. A naive approach to solve this problem would be to take different regions of interest from the image, and use a CNN to classify the presence of the object within that region. The problem with this approach is that the objects of interest might have different spatial locations within the image and different aspect ratios. Hence, you would have to select a huge number of regions and this could computationally blow up. Therefore, algorithms like R-CNN, YOLO etc have been developed to find these occurrences and find them fast.

1. R-CNN

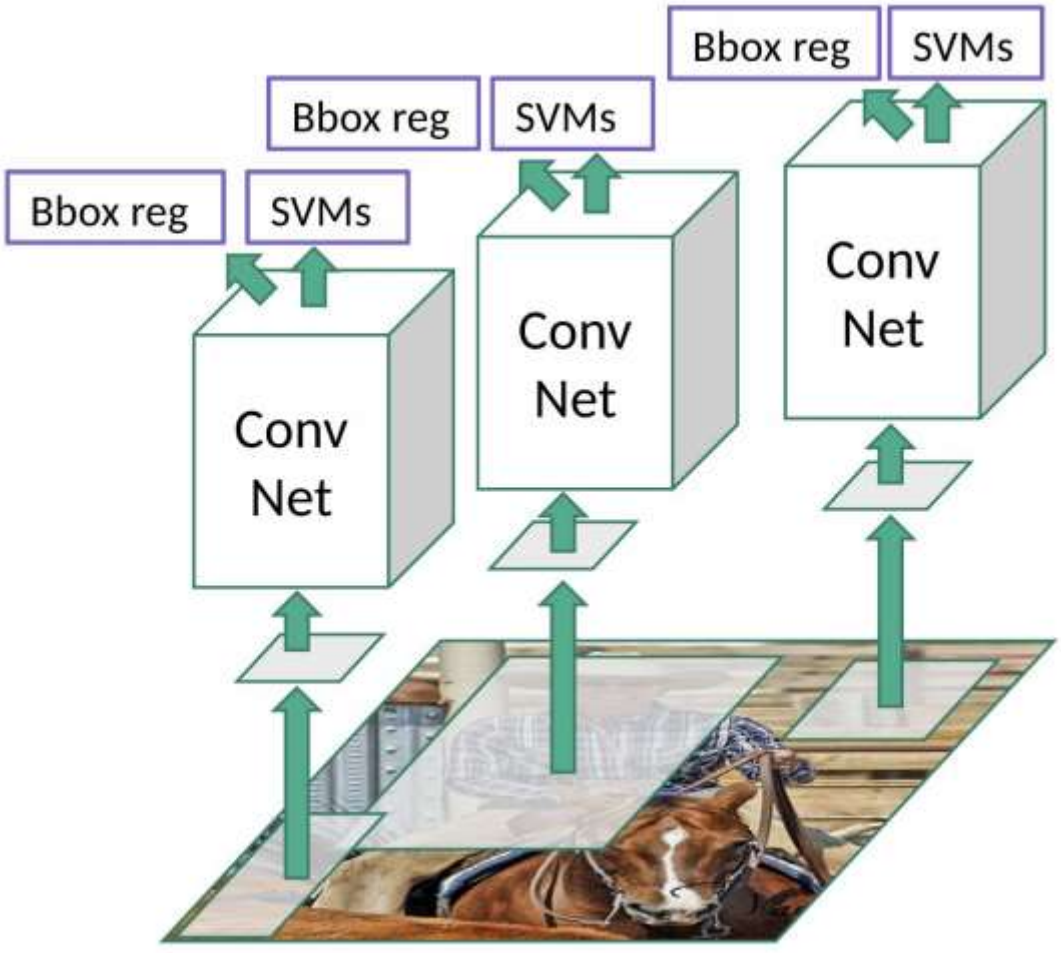
To bypass the problem of selecting a huge number of regions, Ross Girshick et al. proposed a method where we use selective search to extract just 2000 regions from the image and he called them region proposals. Therefore, now, instead of trying to classify a huge number of regions, you can just work with 2000 regions. These 2000 region proposals are generated using the selective search algorithm which is written below.

Selective Search:
1. Generate initial sub-segmentation, we generate many candidate regions
2. Use greedy algorithm to recursively combine similar regions into larger ones
3. Use the generated regions to produce the final candidate region proposals



R-CNN

These 2000 candidate region proposals are warped into a square and fed into a convolutional neural network that produces a 4096-dimensional feature vector as output. The CNN acts as a feature extractor and the output dense layer consists of the features extracted from the image and the extracted features are fed into an SVM to classify the presence of the object within that candidate region proposal. In addition to predicting the presence of an object within the region proposals, the algorithm also predicts four values which are offset values to increase the precision of the bounding box. For example, given a region proposal, the algorithm would have predicted the presence of a person but the face of that person within that region proposal could've been cut in half. Therefore, the offset values help in adjusting the bounding box of the region proposal.

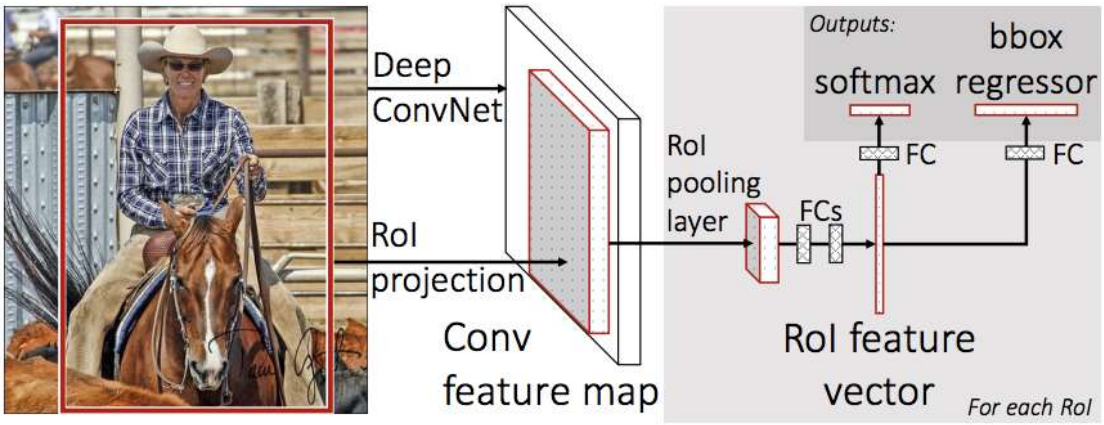


R-CNN

Problems with R-CNN

- It still takes a huge amount of time to train the network as you would have to classify 2000 region proposals per image.
- It cannot be implemented real time as it takes around 47 seconds for each test image.
- The selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage. This could lead to the generation of bad candidate region proposals.

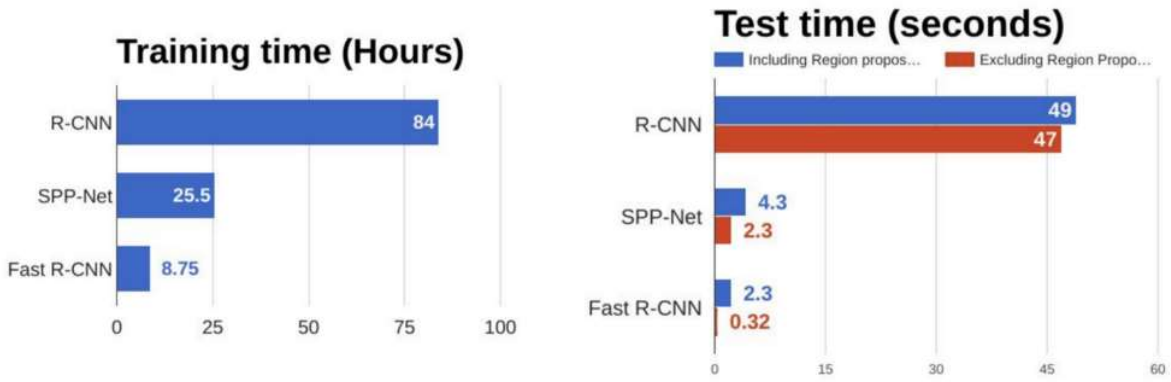
2. Fast R-CNN



R-CNN

The same author of the previous paper(R-CNN) solved some of the drawbacks of R-CNN to build a faster object detection algorithm and it was called Fast R-CNN. The approach is similar to the R-CNN algorithm. But, instead of feeding the region proposals to the CNN, we feed the input image to the CNN to generate a convolutional feature map. From the convolutional feature map, we identify the region of proposals and warp them into squares and by using a RoI pooling layer we reshape them into a fixed size so that it can be fed into a fully connected layer. From the RoI feature vector, we use a softmax layer to predict the class of the proposed region and also the offset values for the bounding box.

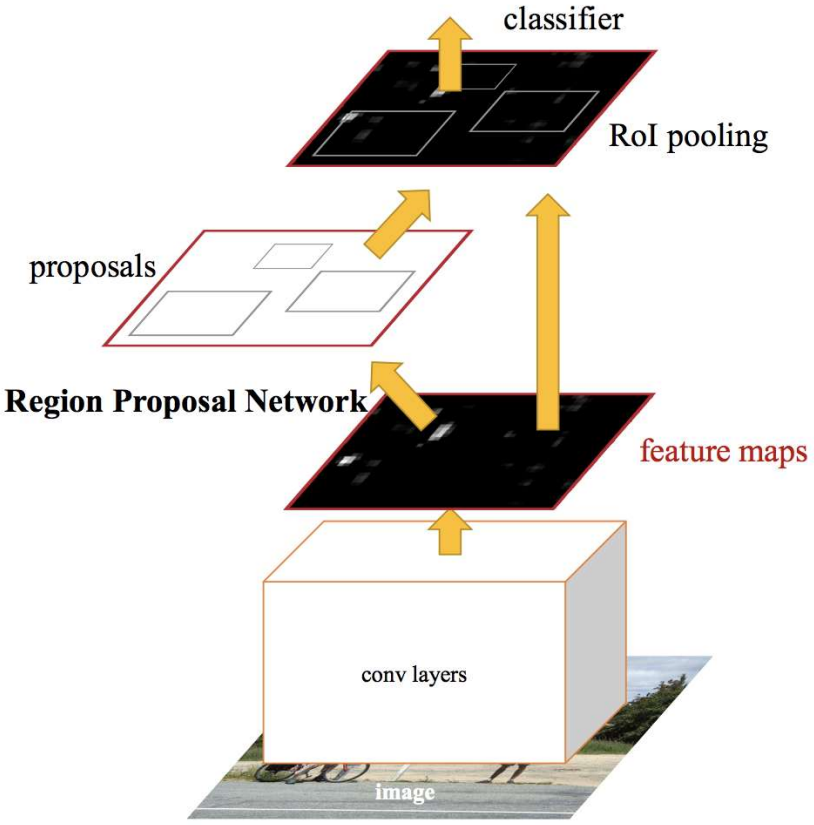
The reason "Fast R-CNN" is faster than R-CNN is because you don't have to feed 2000 region proposals to the convolutional neural network every time. Instead, the convolution operation is done only once per image and a feature map is generated from it.



Comparison of object detection algorithms

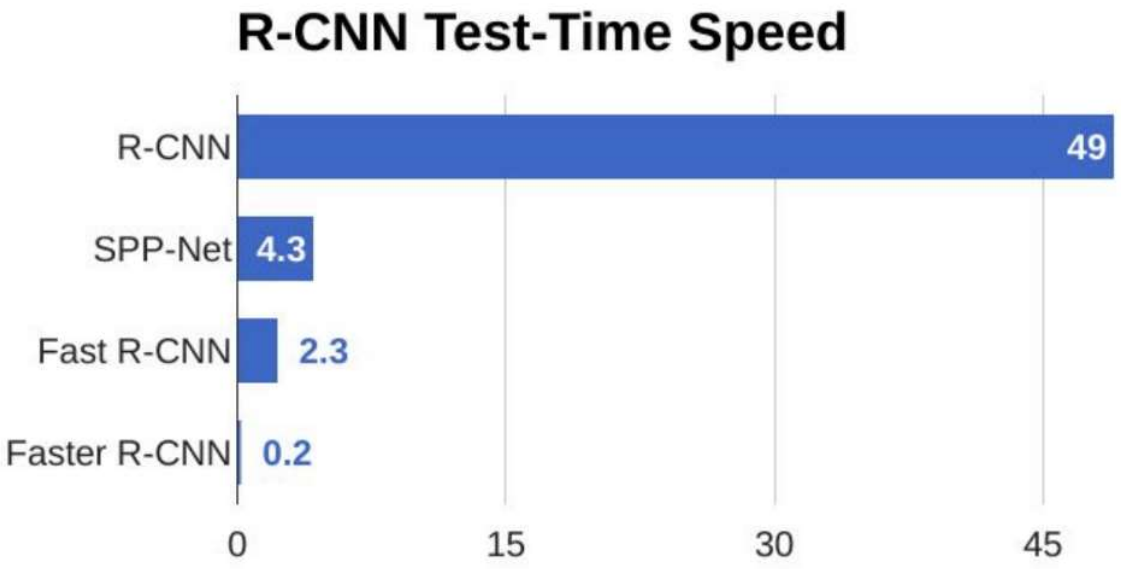
From the above graphs, you can infer that Fast R-CNN is significantly faster in training and testing sessions over R-CNN. When you look at the performance of Fast R-CNN during testing time, including region proposals slows down the algorithm significantly when compared to not using region proposals. Therefore, region proposals become bottlenecks in Fast R-CNN algorithm affecting its performance.

3. Faster R-CNN



Both of the above algorithms (R-CNN & Fast R-CNN) uses selective search to find out the region proposals. Selective search is a slow and time-consuming process affecting the performance of the network. Therefore, Shaoqing Ren et al. came up with an object detection algorithm that eliminates the selective search algorithm and lets the network learn the region proposals.

Similar to Fast R-CNN, the image is provided as an input to a convolutional network which provides a convolutional feature map. Instead of using selective search algorithm on the feature map to identify the region proposals, a separate network is used to predict the region proposals. The predicted region proposals are then reshaped using a RoI pooling layer which is then used to classify the image within the proposed region and predict the offset values for the bounding boxes.



From the above graph, you can see that **Faster R-CNN** is much faster than it's predecessors. Therefore, it can even be used for real-time object detection.

3.Implementation

3.1. Major Software and Libraries used:

3.1.1. Anaconda

Anaconda isa free and open-source distribution of the Python and R programming languages for data science and machine learning applications (large-scale data processing, predictive analytics, scientific computing), that aims to simplify package management and deployment. Package versions are managed by the package management system conda. The Anaconda distribution is used by over 6 million users and includes more than 250 popular data-science packages suitable for Windows, Linux, and MacOS.

3.1.2. Python

The programming style of Python is simple, clear and it also contains powerful different kinds of classes. Moreover, Python can easily combine other programming languages,

such as C or C++.

As a successful programming language, it has its own advantages:

- Simple and easy to learn
- Open source
- Scalability

3.1.3. Scikit-learn

Scikit–learn is an open source machine learning library for the Python programming language. It features various classification, regression, and clustering algorithms and is designed to interoperate with the Python numerical libraries NumPy and SciPy. SciKit-learn contains the K-means algorithm based on Python and it helps to figure out how to implement this algorithm in programming.

3.1.4 Numpy, Scipy and Matplotlib

In Python, there is no data type called array. In order to implement the data type of array with python, numpy and scipy are the essential libraries for analysing and calculating data. They are all open source libraries. Numpy is mainly used

for the matrix calculation. scipy is developed based on numpy and it is mainly used for scientific research.

3.1.5 Keras

Keras is an open source neural network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, or Theano Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. It was developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System), and its primary author and maintainer is François Chollet, a Google engineer.

3.2 System requirements as used in this project

- Intel i5 processor
- 8 GB RAM
- 1TB hard disk
- NVIDIA geforce 920Mx

But for better results CPU and GPU of high specifications must be used.

4.OpenCV

OpenCV (Open source computer vision) is a library of programming functions mainly aimed at real-time computer vision. Originally developed by Intel, it was later supported by Willow Garage then Itseez (which was later acquired by Intel). The library is cross-platform and free for use under the open-source BSD license.

OpenCV supports the deep leaning framework TensorFlow, Torch/PyTorch and caffe.

OpenCV is written in C++ and its primary interface is in C++, but it still retains a less comprehensive though extensive older C interface. There are bindings in Python, Java and MATLAB/OCTAVE. The API for these interfaces can be found in the online documentation. Wrappers in other languages such as C#, Perl, Ch, Haskell, and Ruby have been developed to encourage adoption by a wider audience.

A CUDA-based GPU interface has been in progress since September 2010.

An OpenCL-based GPU interface has been in progress since October 2012, documentation for version 2.4.13.3 can be found at docs.opencv.org.

4.1 OpenCV installation

On Windows Operating System:

```
C:\Users\Pankaj Negi>pip install opencv-python
```

5. CUDA

CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing — an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

The CUDA platform is designed to work with programming languages such as C, C++, and Fortran. This accessibility makes it easier for specialists in parallel programming to use GPU resources, in contrast to prior APIs like Direct3D and OpenGL, which required advanced skills in graphics programming. Also, CUDA supports programming frameworks such as OpenACC and OpenCL. When it was first introduced by Nvidia, the name CUDA was an acronym for **Compute Unified Device Architecture**, but Nvidia subsequently dropped the use of the acronym.

5.1 Advantages of CUDA

- Scattered reads – code can read from arbitrary addresses in memory
- Unified virtual memory (CUDA 4.0 and above)
- Unified memory (CUDA 6.0 and above)
- Shared memory – CUDA exposes a fast shared memory region that can be shared among threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- Faster downloads and readbacks to and from the GPU
- Full support for integer and bitwise operations, including integer texture lookups

6.Tensorflow

TensorFlow, a open-source software library for dataflow programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. It is used for both research and production at Google.

TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache 2.0 open-source license on November 9, 2015.

TensorFlow is Google Brain's second-generation system. Version 1.0.0 was released on February 11, 2017. While the reference implementation runs on single devices, TensorFlow can run on multiple CPUs and GPUs (with optional CUDA and SYCL extensions for general-purpose computing on graphics processing units). TensorFlow is available on 64-bit Linux, macOS, Windows, and mobile computing platforms including Android and iOS.

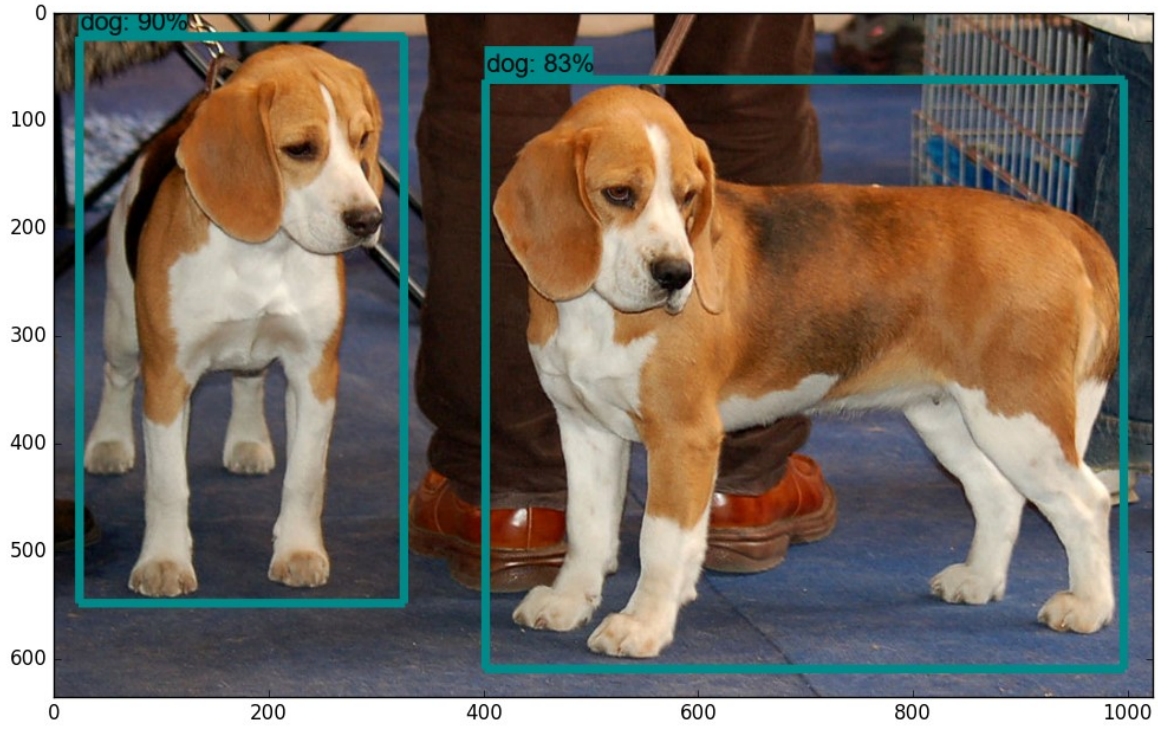
Its flexible architecture allows for the easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices.

TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays. These arrays are referred to as "tensors". In June 2016, Dean stated that 1,500 repositories on GitHub mentioned TensorFlow, of which only 5 were from Google.

6.1 Why Tensorflow for Object Detection

- It allows Deep Learning.
- Known as the second-generation machine learning system, it performs numerical computations through data flow graphs.
- It is open source and free.
- It is reliable (and without major bugs).

- It is backed by Google and a good community.
- It is a skill recognized by many employers.
- It is easy to implement.
- With capability of running on CPUs and GPUs, it can be deployed in broad range of products of Google such as Speech Recognition, Google Photos, Gmail and even Search.



TensorFlow Object Detection sample

6.2 TensorFlow Installation

Dependencies

Tensorflow Object Detection API depends on the following libraries:

- Protobuf 3.0.0
- Python-tk
- Pillow 1.0
- lxml
- tf Slim (which is included in the "tensorflow/models/research/" checkout)
- Jupyter notebook
- Matplotlib
- Tensorflow (>=1.9.0)

- Cython
- contextlib2
- cocoapi

For detailed steps to install Tensorflow, follow the Tensorflow installation instructions. A typical user can install Tensorflow using one of the following commands:

```
# For CPU
pip install tensorflow
# For GPU
pip install tensorflow-gpu
```

The remaining libraries can be installed on Ubuntu 16.04 using via apt-get:

```
sudo apt-get install protobuf-compiler python-pil python-lxml python-tk
pip install --user Cython
pip install --user contextlib2
pip install --user jupyter
pip install --user matplotlib
```

Alternatively, users can install dependencies using pip:

```
pip install --user Cython
pip install --user contextlib2
pip install --user pillow
pip install --user lxml
pip install --user jupyter
pip install --user matplotlib
```

Note: sometimes "sudo apt-get install protobuf-compiler" will install Protobuf 3+ versions for you and some users have issues when using 3.5. If that is your case, try the manual installation.

COCO API installation

We can download the cocoapi and copy the pycocotools subfolder to the tensorflow/models/research directory if you are interested in using COCO evaluation metrics. 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.

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

Protobuf 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 compiled. This should be done by running the following command from the tensorflow/models/research/ directory:

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

Manual protobuf-compiler installation and usage

For linux:

We have to download and install the 3.0 release of protoc, then unzip the file.

```
# From tensorflow/models/research/
wget -O protobuf.zip
https://github.com/google/protobuf/releases/download/v3.0.0/protoc-3.0.0-
linux-x86_64.zip
unzip protobuf.zip
```

Run the compilation process again, but use the downloaded version of protoc

```
# From tensorflow/models/research/
./bin/protoc object_detection/protos/*.proto --python_out=.Add Libraries to
PYTHONPATH
```

When running locally, the tensorflow/models/research/ and slim directories should be appended to PYTHONPATH. This can be done by running the following from tensorflow/models/research/:

```
# From tensorflow/models/research/
export PYTHONPATH=$PYTHONPATH:`pwd`:`pwd`/slim
```

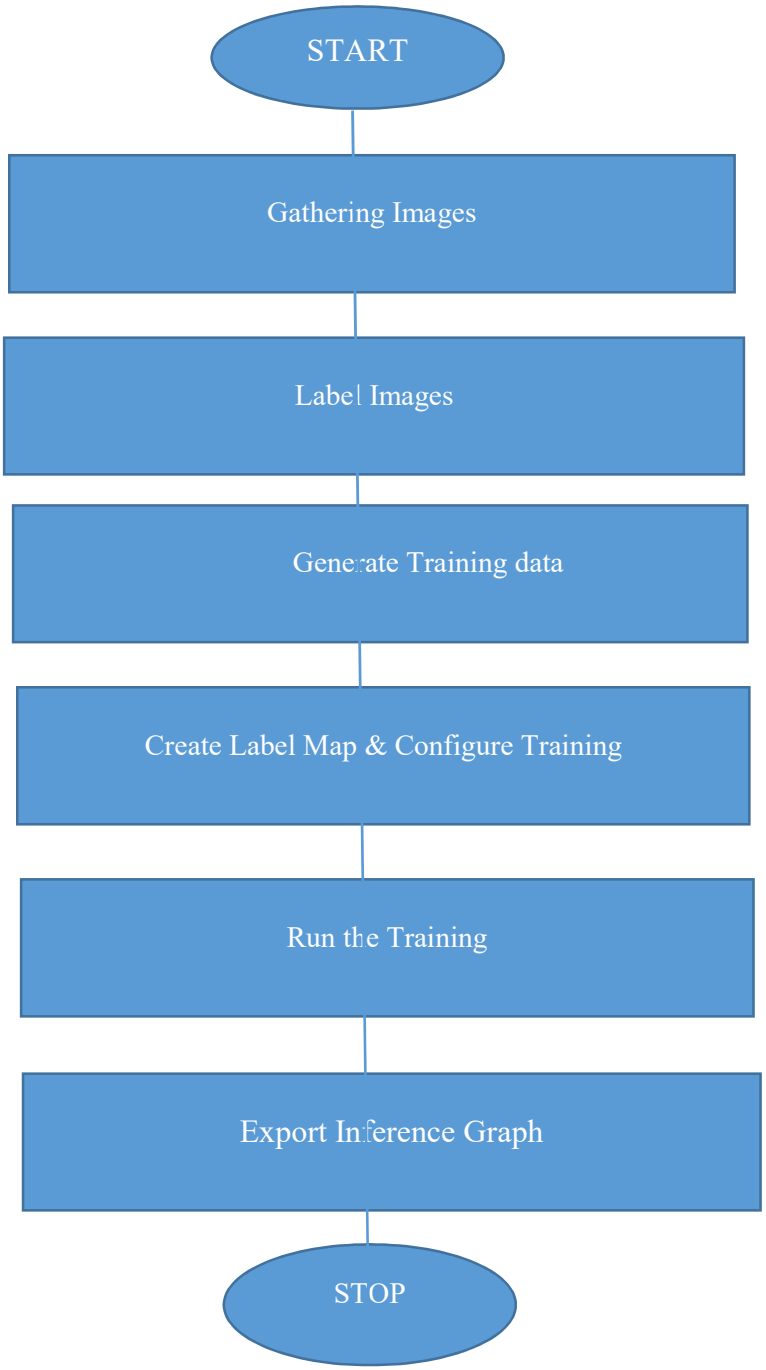
Testing the Installation

You can test that you have correctly installed the Tensorflow Object Detection API by running the following command:

```
python object_detection/builders/model_builder_test.py
```

7.Methodology

To identify and detect a two-wheeler (bike Rider) detection, the **workflow of process**



In this approach we need a object detection model, which can detect a bike rider.

TensorFlow provides several object detection models (pre-trained classifiers with specific neural network architectures) in its model zoo. Some models (such as the SSD-MobileNet model) have an architecture that allows for faster detection but with less accuracy, while some models (such as the Faster-RCNN model) give slower detection but with more accuracy. I initially started with the SSD-MobileNet-V1 model, but it didn't do a very good job identifying the cards in my images. I re-trained my detector on the Faster-RCNN-Inception-V2 model, and the detection worked considerably better, but with a noticeably slower speed.

Procedure for training a Faster-RCNN-Inception-V2 model

7.1. Gathering images (Creating data set):

To detect a bike rider with helmet or without helmet and license plate. We need bunch of images of bike-riders with helmet, bike-rider without helmet and bike license plate.

In this project, I'm using 627 images. Put the 20% of images in \object_detection\images\test directory and remaining 80% in \object_detection\images\train directory.

Sample images bike-rider without helmet

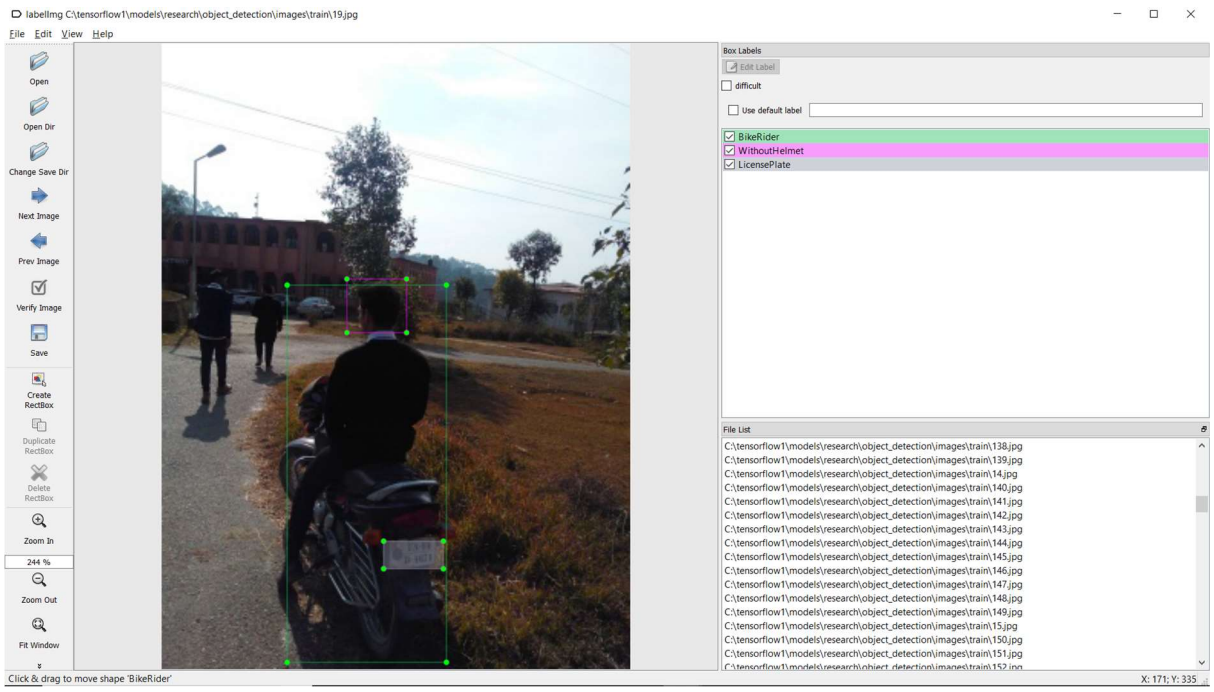
Sample images bike-rider with helmet

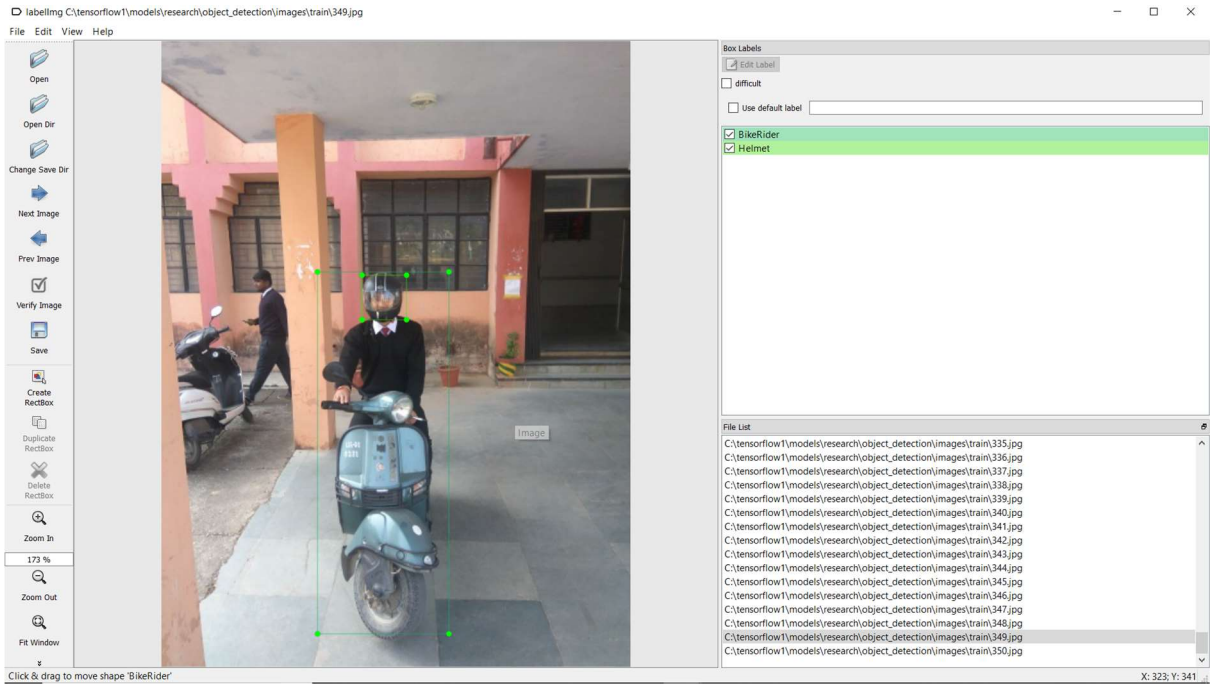
Make sure the images aren't too large. They should be less than 200KB each, and their resolution shouldn't be more than 720x1280. The larger the images are, the longer it will take to train the classifier. You can use the resizer.py script in this repository to reduce the size of the images.

7.2. Label Images:

Label the all images inside the \object_detection\images\test directory and \object_detection\images\train directory with the help of **LableImg** tool.

In this project, BikeRider, Helmet, WithoutHelmet and LicensePlate four classes were created with the help of LabelImg tool. Create .xml file corresponding to each image with the above following categories of classes.





Creating labels on images

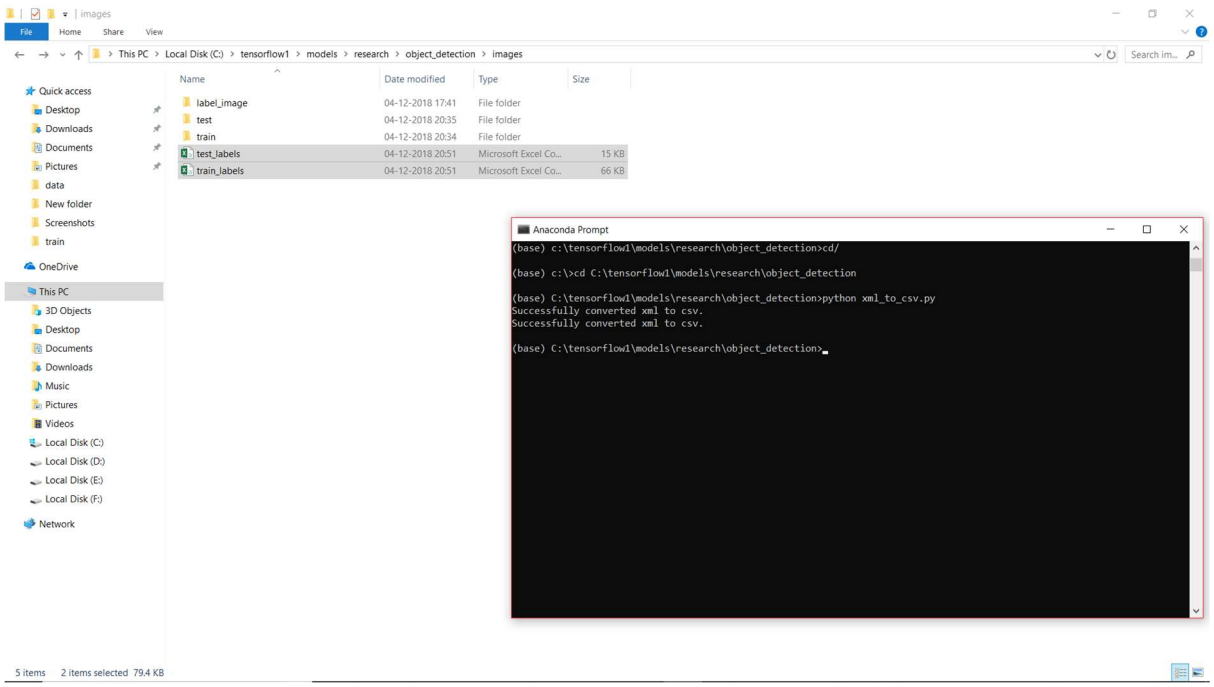
7.3. Generate Training Data:

With the images labeled, it's time to generate the TFRecords that serve as input data to the TensorFlow training model. This tutorial uses the xml_to_csv.py and generate_tfrecord.py scripts from Dat Tran's Raccoon Detector dataset, with some slight modifications to work with our directory structure.

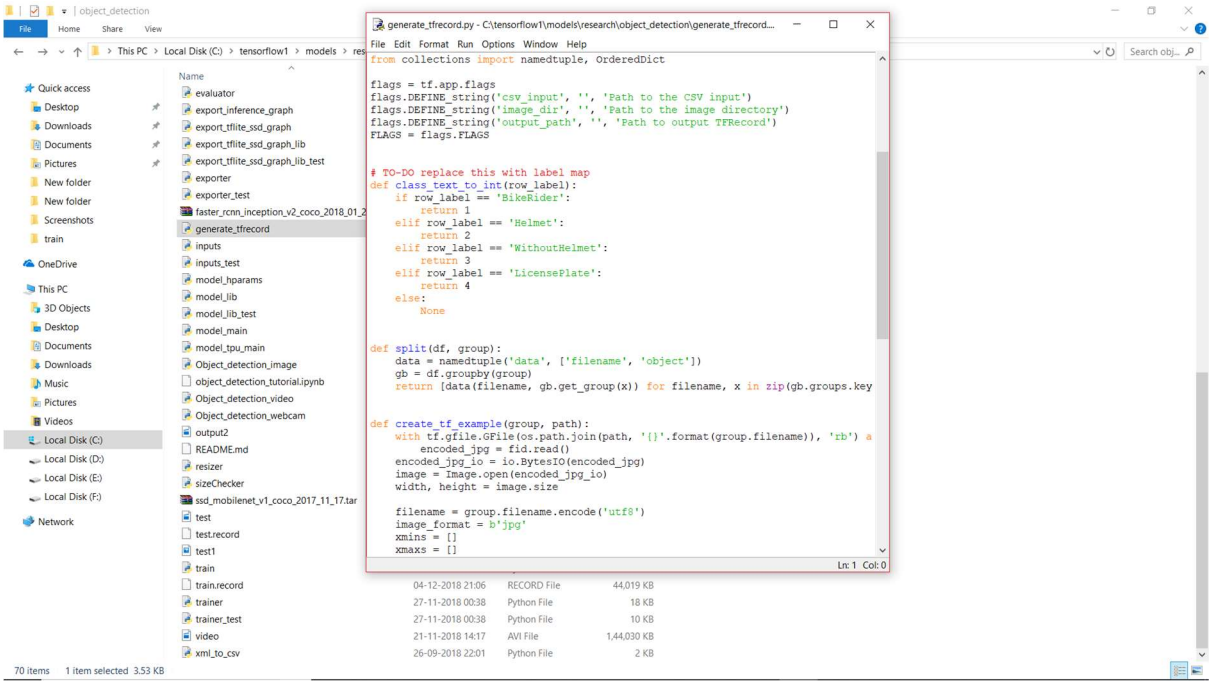
First, the image .xml data will be used to create .csv files containing all the data for the train and test images. From the \object_detection folder, issue the following command in the Anaconda command prompt

```
(tensorflow1) C:\tensorflow1\models\research\object_detection> python xml_to_csv.py
```

This creates a train_labels.csv and test_labels.csv file in the \object_detection\images folder.

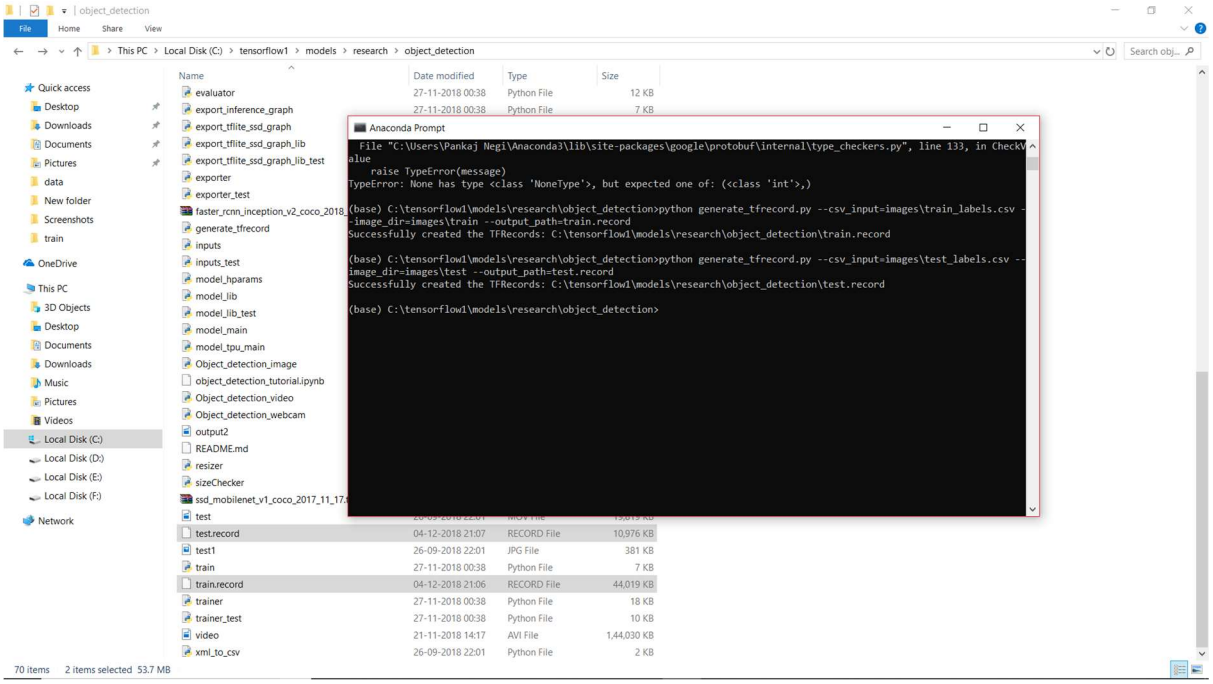


Next, open the generate_tfrecord.py file in a text editor. Replace the label map starting at line 31 with your own label map(BikeRider, Helmet, WithoutHelmet, LicensePlate), where each object is assigned an ID number. This same number assignment will be used when configuring the labelmap.pbtxt file.



Then, generate the TFRecord files by issuing these commands from the \object_detection folder:

```
python generate_tfrecord.py --csv_input=images\train_labels.csv --
image_dir=images\train --output_path=train.record
python generate_tfrecord.py --csv_input=images\test_labels.csv --
image_dir=images\test --output_path=test.record
```

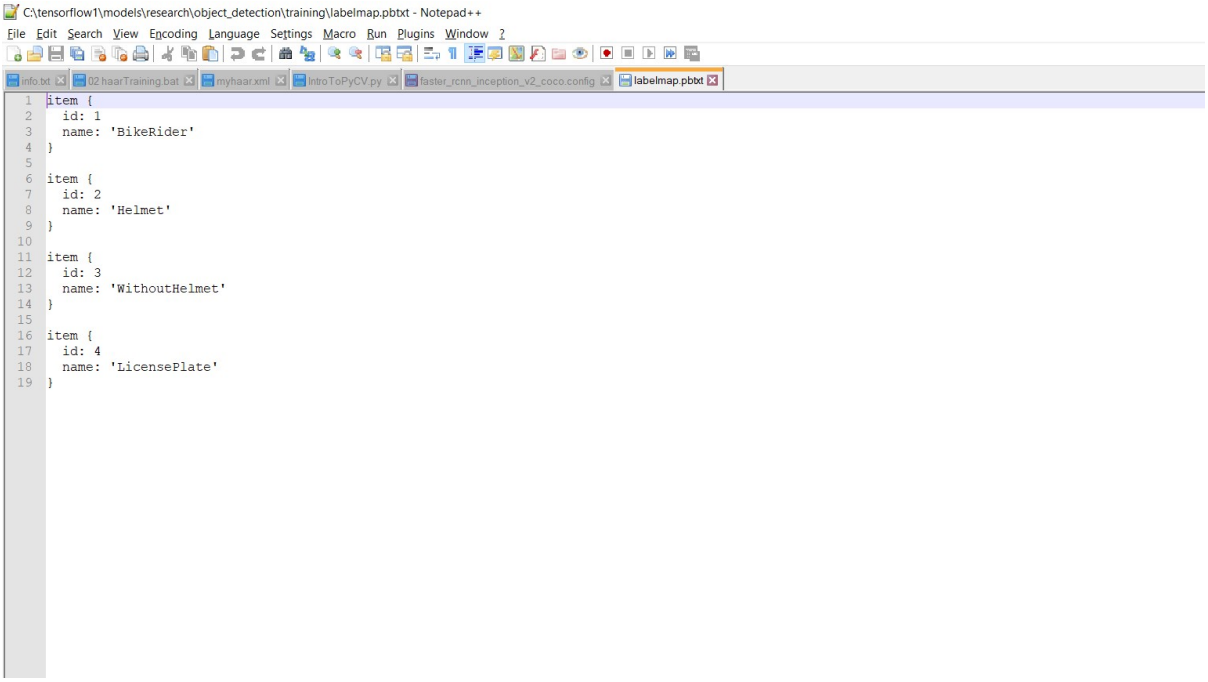


These generate a train.record and a test.record file in \object_detection. These will be used to train the new object detection classifier.

7.4. Create Label Map and Configure Training:

7.4.1 Label Map

The label map tells the trainer what each object is by defining a mapping of class names to class ID numbers. Use a text editor to create a new file and save it as labelmap.pbtxt in the C:\tensorflow1\models\research\object_detection\training folder.



The label map ID numbers should be the same as what is defined in the generate_tfrecord.py file.

7.4.2 Configure training

Finally, the object detection training pipeline must be configured. It defines which model and what parameters will be used for training. This is the last step before running training!

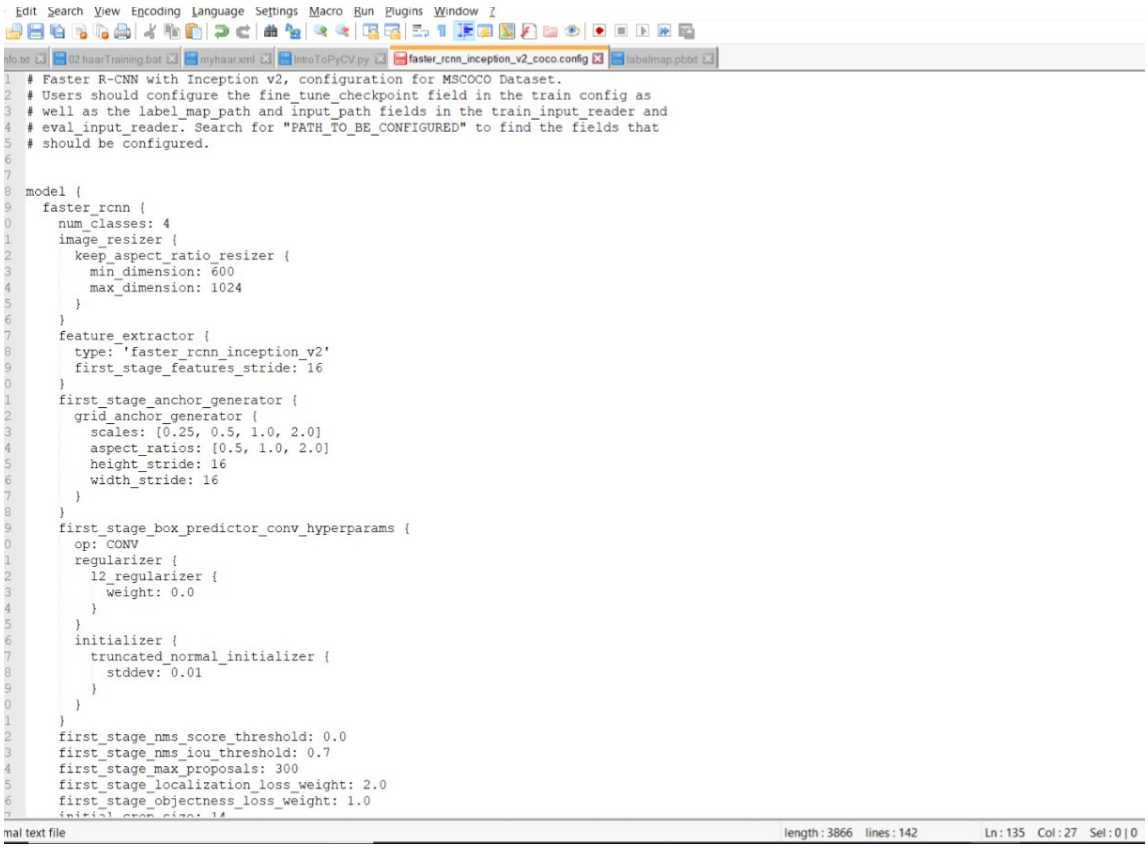
Navigate to C:\tensorflow1\models\research\object_detection\samples\configs and copy the faster_rcnn_inception_v2_coco.config file into the \object_detection\training directory. Then, open the file with a text editor. There

are several changes to make to the .config file, mainly changing the number of classes and examples, and adding the file paths to the training data.

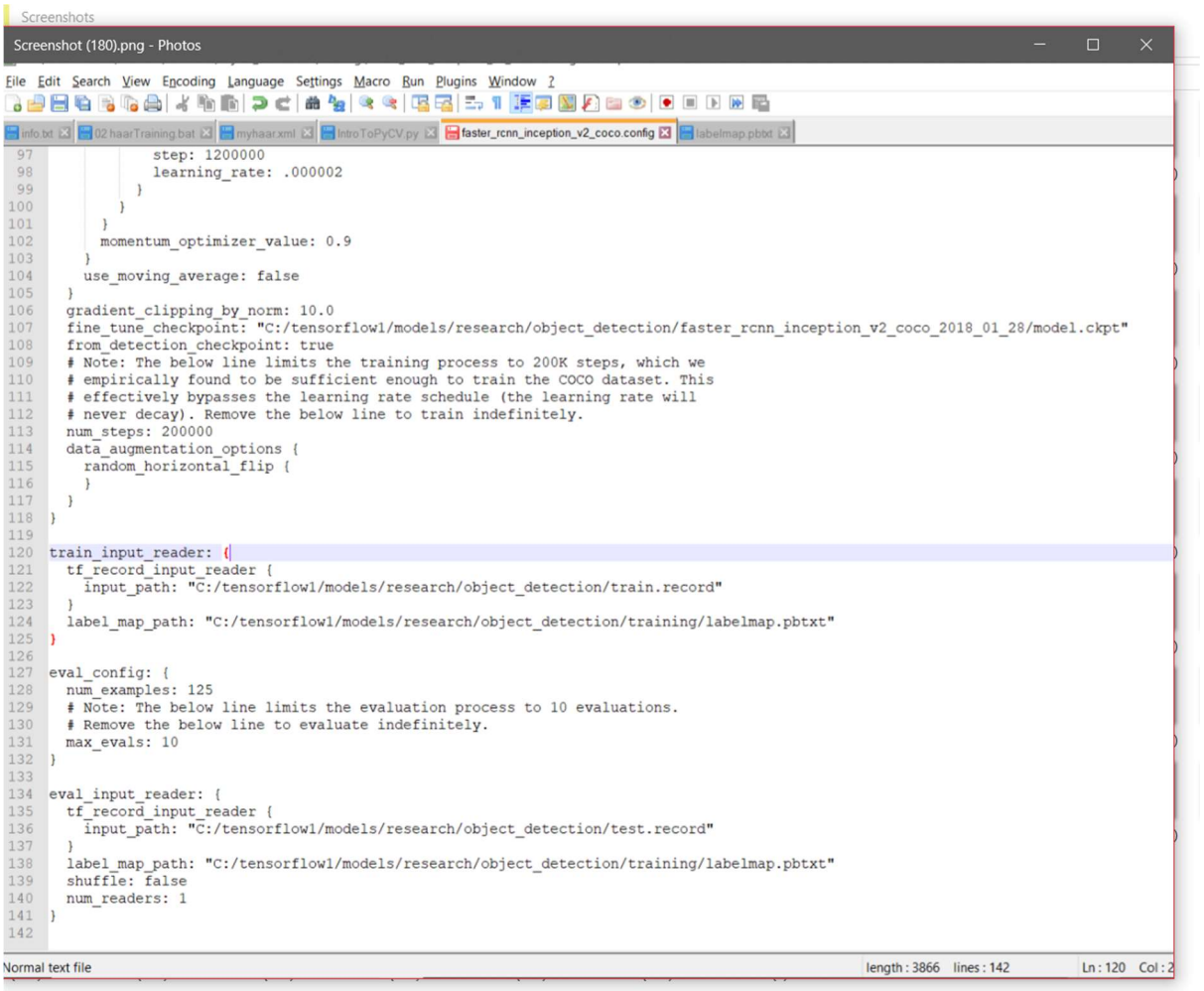
Make the following changes to the faster_rcnn_inception_v2_coco.config file. Note: The paths must be entered with single forward slashes (NOT backslashes), or TensorFlow will give a file path error when trying to train the model! Also, the paths must be in double quotation marks ("), not single quotation marks (').

- Line 9. Change num_classes to the number of different objects you want the classifier to detect. For the BikeRider, Helmet, WithoutHelmet and LicensePlate, it would be num_classes : 4.
- Line 110. Change fine_tune_checkpoint to:
    - fine_tune_checkpoint : "C:/tensorflow1/models/research/object_detection/faster_rcnn_inception_v2_coco_2018_01_28/model.ckpt"
- Lines 126 and 128. In the train_input_reader section, change input_path and label_map_path to:
    - input_path : "C:/tensorflow1/models/research/object_detection/train.record"
    - label_map_path: "C:/tensorflow1/models/research/object_detection/training/labelmap.pbtxt"
- Line 132. Change num_examples to the number of images you have in the \images\test directory.
- Lines 140 and 142. In the eval_input_reader section, change input_path and label_map_path to:
    - input_path : "C:/tensorflow1/models/research/object_detection/test.record"
    - label_map_path: "C:/tensorflow1/models/research/object_detection/training/labelmap.pbtxt"

Save the file after the changes have been made. That’s it! The training job is all configured and ready to go!



```
# Faster R-CNN 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 {
  faster_rcnn {
    num_classes: 4
    image_resizer {
      keep_aspect_ratio_resizer {
        min_dimension: 600
        max_dimension: 1024
      }
    }
    feature_extractor {
      type: 'faster_rcnn_inception_v2'
      first_stage_features_stride: 16
    }
    first_stage_anchor_generator {
      grid_anchor_generator {
        scales: [0.25, 0.5, 1.0, 2.0]
        aspect_ratios: [0.5, 1.0, 2.0]
        height_stride: 16
        width_stride: 16
      }
    }
    first_stage_box_predictor_conv_hyperparams {
      op: CONV
      regularizer {
        l2_regularizer {
          weight: 0.0
        }
      }
      initializer {
        truncated_normal_initializer {
          stddev: 0.01
        }
      }
    }
    first_stage_nms_score_threshold: 0.0
    first_stage_nms_iou_threshold: 0.7
    first_stage_max_proposals: 300
    first_stage_localization_loss_weight: 2.0
    first_stage_objectness_loss_weight: 1.0
```



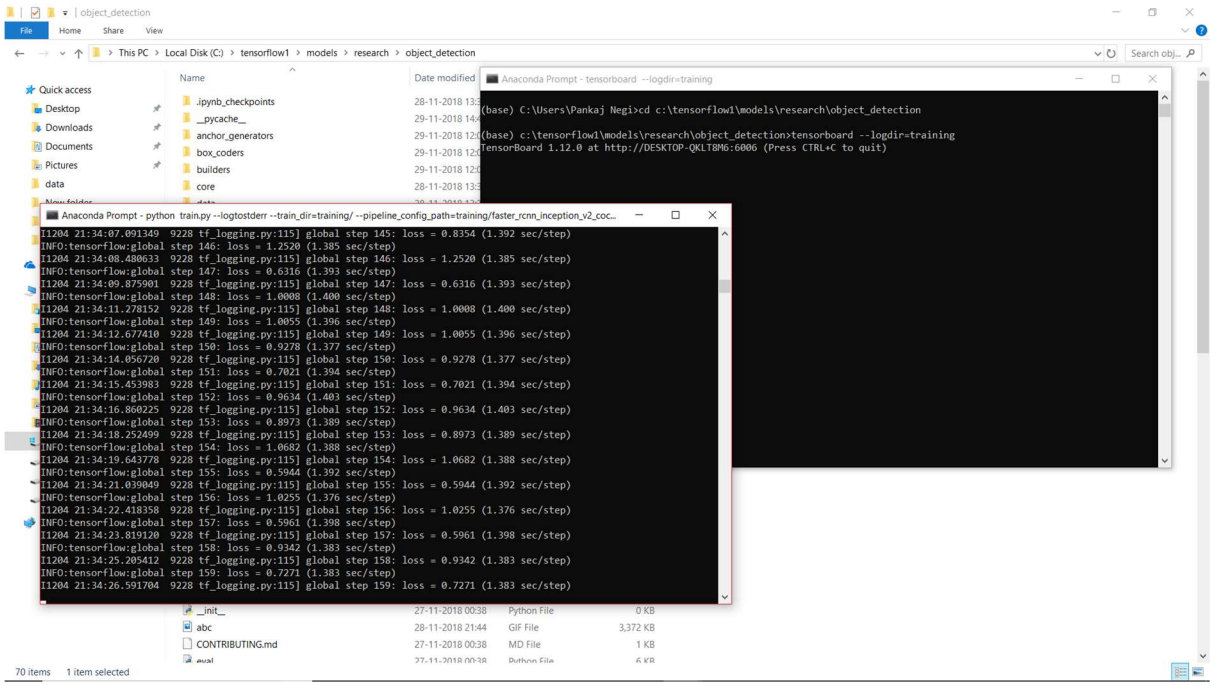
```
            step: 1200000
            learning_rate: .000002
          }
        }
      }
      momentum_optimizer_value: 0.9
    }
    use_moving_average: false
  }
  gradient_clipping_by_norm: 10.0
  fine_tune_checkpoint: "C:/tensorflow1/models/research/object_detection/faster_rcnn_inception_v2_coco_2018_01_28/model.ckpt"
  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 COCO 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 {
    }
  }
}

train_input_reader: {
  tf_record_input_reader {
    input_path: "C:/tensorflow1/models/research/object_detection/train.record"
  }
  label_map_path: "C:/tensorflow1/models/research/object_detection/training/labelmap.pbtxt"
}

eval_config: {
  num_examples: 125
  # 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: "C:/tensorflow1/models/research/object_detection/test.record"
  }
  label_map_path: "C:/tensorflow1/models/research/object_detection/training/labelmap.pbtxt"
  shuffle: false
  num_readers: 1
}
```

7.5. Run the Training:

From the \object_detection directory, issue the following command to begin training:

```
python train.py --logtostderr --train_dir=training/ --
pipeline_config_path=training/faster_rcnn_inception_v2_coco.config
```

If everything has been set up correctly, TensorFlow will initialize the training. The initialization can take up to 30 seconds before the actual training begins. When training begins, it will look like this:

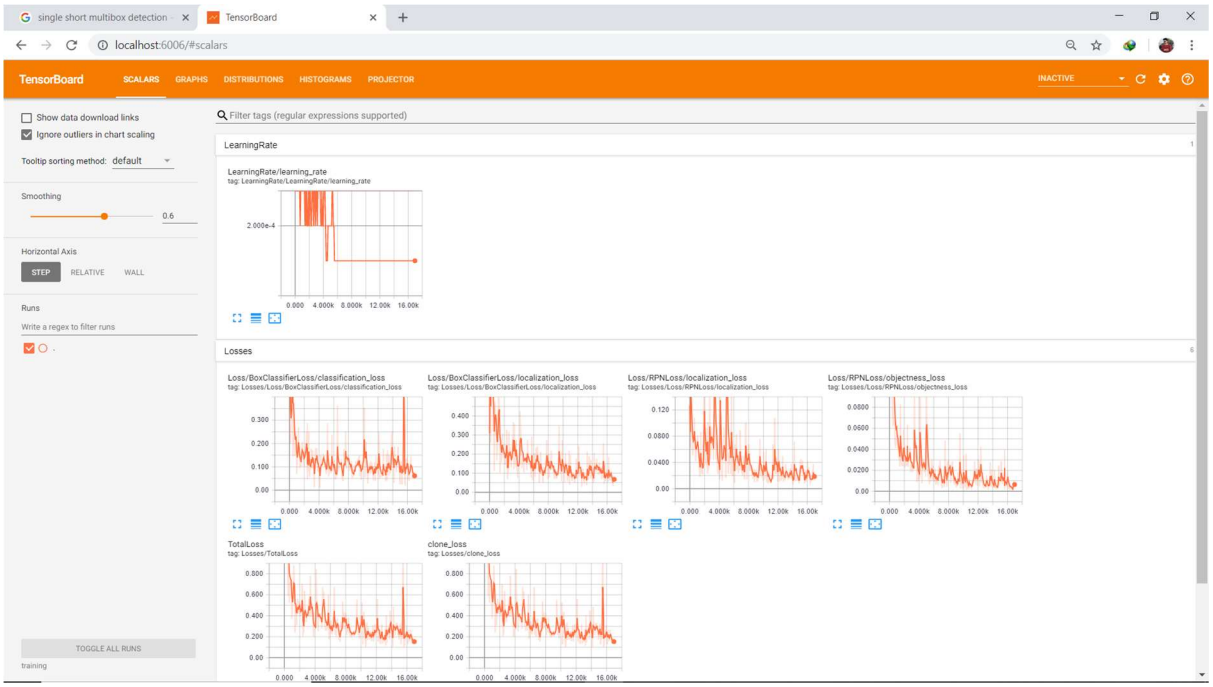


Each step of training reports the loss. It will start high and get lower and lower as training progresses. For my training on the Faster-RCNN-Inception-V2 model, it started at about 3.0 and quickly dropped below 0.8. I recommend allowing your model to train until the loss consistently drops below 0.05, which will take about 40,000 steps, or about 2 hours (depending on how powerful your CPU and GPU are). Note: The loss numbers will be different if a different model is used. MobileNet-SSD starts with a loss of about 20, and should be trained until the loss is consistently under 2.

You can view the progress of the training job by using TensorBoard. To do this, open a new instance of Anaconda Prompt, activate the tensorflow1 virtual environment, change to the C:\tensorflow1\models\research\object_detection directory, and issue the following command:

```
(tensorflow1) C:\tensorflow1\models\research\object_detection>tensorboard --logdir=training
```

This will create a webpage on your local machine at YourPCName:6006, which can be viewed through a web browser. The TensorBoard page provides information and graphs that show how the training is progressing. One important graph is the Loss graph, which shows the overall loss of the classifier over time.



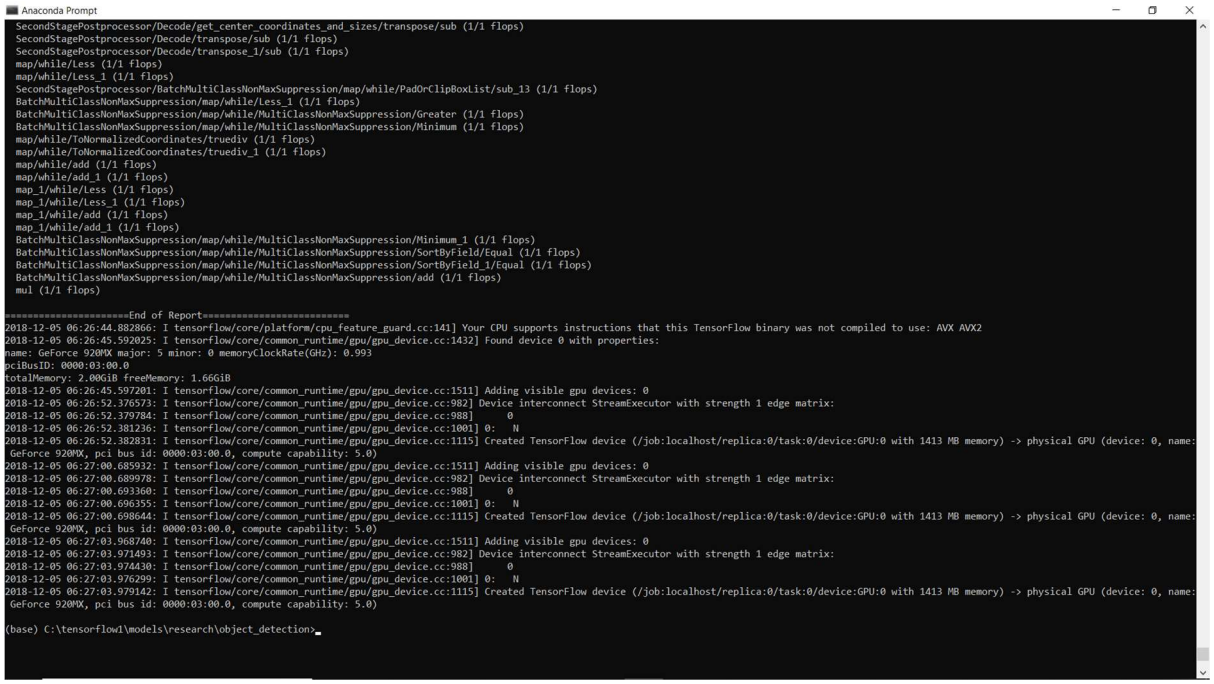
The training routine periodically saves checkpoints about every five minutes. You can terminate the training by pressing Ctrl+C while in the command prompt window. I typically wait until just after a checkpoint has been saved to terminate the training. You can terminate training and start it later, and it will restart from the last saved checkpoint. The checkpoint at the highest number of steps will be used to generate the frozen inference graph.

7.6.Export Inference Graph:

Now that training is complete, the last step is to generate the frozen inference graph (.pb file). From the \object_detection folder, issue the following command, where "XXXX" in "model.ckpt-XXXX" should be replaced with the highest-numbered .ckpt file in the training folder:

In my case xxxx is 21094.

```
python export_inference_graph.py --input_type image_tensor --
pipeline_config_path training/faster_rcnn_inception_v2_coco.config --
trained_checkpoint_prefix training/model.ckpt-21094 --output_directory
inference_graph
```



This creates a frozen_inference_graph.pb file in the \object_detection\inference_graph folder. The .pb file contains the object detection classifier.

7.7.Use Your Newly Trained Object Detection Classifier!

Before running the Python scripts, you need to modify the NUM_CLASSES variable in the script to equal the number of classes you want to detect. (For my BikeRider Detector, there are four classes I want to detect, so NUM_CLASSES = 4).

To run any of the scripts, type “idle” in the Anaconda Command Prompt (with the “tensorflow1” virtual environment activated) and press ENTER. This will open IDLE, and from there, you can open any of the scripts and run them.

If everything is working properly, the object detector will initialize for about 10 seconds and then display a window showing any objects it’s detected in the image!

7.7.1 Bike-Rider detection image

```
# Import packages
import os
import cv2
import numpy as np
import tensorflow as tf
import sys

# This is needed since the notebook is stored in the object_detection folder.
sys.path.append("..")

# Import utilites
from utils import label_map_util
from utils import visualization_utils as vis_util

# Name of the directory containing the object detection module we're using
MODEL_NAME = 'inference_graph'
IMAGE_NAME = 'test1.jpg'

# Grab path to current working directory
CWD_PATH = os.getcwd()

# Path to frozen detection graph .pb file, which contains the model that is used
# for object detection.
PATH_TO_CKPT =
os.path.join(CWD_PATH,MODEL_NAME,'frozen_inference_graph.pb')

# Path to label map file
PATH_TO_LABELS = os.path.join(CWD_PATH,'training','labelmap.pbtxt')

# Path to image
PATH_TO_IMAGE = os.path.join(CWD_PATH,IMAGE_NAME)

# Number of classes the object detector can identify
NUM_CLASSES = 4

# Load the label map.
# Label maps map indices to category names, so that when our convolution
# network predicts `5`, we know that this corresponds to `king`.
# 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)

# Load the Tensorflow model into memory.
detection_graph = tf.Graph()
with detection_graph.as_default():
    od_graph_def = tf.GraphDef()
    with tf.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='')

    sess = tf.Session(graph=detection_graph)

# Define input and output tensors (i.e. data) for the object detection classifier

# Input tensor is the image
image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')

# Output tensors are the detection boxes, scores, and classes
# Each box represents a part of the image where a particular object was detected
detection_boxes = detection_graph.get_tensor_by_name('detection_boxes:0')

# Each score represents level of confidence for each of the objects.
# The score is shown on the result image, together with the class label.
detection_scores = detection_graph.get_tensor_by_name('detection_scores:0')
detection_classes = detection_graph.get_tensor_by_name('detection_classes:0')

# Number of objects detected
num_detections = detection_graph.get_tensor_by_name('num_detections:0')

# Load image using OpenCV and
# expand image dimensions to have shape: [1, None, None, 3]
# i.e. a single-column array, where each item in the column has the pixel RGB
value
image = cv2.imread(PATH_TO_IMAGE)
image_expanded = np.expand_dims(image, axis=0)

# Perform the actual detection by running the model with the image as input
(boxes, scores, classes, num) = sess.run(
    [detection_boxes, detection_scores, detection_classes, num_detections],
    feed_dict={image_tensor: image_expanded})
```

```
# Draw the results of the detection (aka 'visulaize the results')

vis_util.visualize_boxes_and_labels_on_image_array(
    image,
    np.squeeze(boxes),
    np.squeeze(classes).astype(np.int32),
    np.squeeze(scores),
    category_index,
    use_normalized_coordinates=True,
    line_thickness=8,
    min_score_thresh=0.80)

# All the results have been drawn on image. Now display the image.
cv2.imshow('Object detector', image)

# Press any key to close the image
cv2.waitKey(0)

# Clean up
cv2.destroyAllWindows()
```

Input Image Output Image

7.7.2 Bike-Rider Detection Video

```
import os
import cv2
import numpy as np
import tensorflow as tf
import sys

# This is needed since the notebook is stored in the object_detection folder.
sys.path.append("..")

# Import utilites
from utils import label_map_util
from utils import visualization_utils as vis_util

# Name of the directory containing the object detection module we're using
MODEL_NAME = 'inference_graph'
VIDEO_NAME = 'test1.mkv'

# Grab path to current working directory
CWD_PATH = os.getcwd()

# Path to frozen detection graph .pb file, which contains the model that is used
# for object detection.
PATH_TO_CKPT =
os.path.join(CWD_PATH,MODEL_NAME,'frozen_inference_graph.pb')

# Path to label map file
PATH_TO_LABELS = os.path.join(CWD_PATH,'training','labelmap.pbtxt')

# Path to video
PATH_TO_VIDEO = os.path.join(CWD_PATH,VIDEO_NAME)

# Number of classes the object detector can identify
NUM_CLASSES = 4

# Load the label map.
# Label maps map indices to category names, so that when our convolution
# network predicts `5`, we know that this corresponds to `king`.
# 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)

# Load the Tensorflow model into memory.
detection_graph = tf.Graph()
with detection_graph.as_default():
    od_graph_def = tf.GraphDef()
    with tf.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='')

    sess = tf.Session(graph=detection_graph)

# Input tensor is the image
image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')

# Output tensors are the detection boxes, scores, and classes
# Each box represents a part of the image where a particular object was detected
detection_boxes = detection_graph.get_tensor_by_name('detection_boxes:0')

# Each score represents level of confidence for each of the objects.
# The score is shown on the result image, together with the class label.
detection_scores = detection_graph.get_tensor_by_name('detection_scores:0')
detection_classes = detection_graph.get_tensor_by_name('detection_classes:0')

# Number of objects detected
num_detections = detection_graph.get_tensor_by_name('num_detections:0')

# Open video file
video = cv2.VideoCapture(PATH_TO_VIDEO)

while(video.isOpened()):

    # Acquire frame and expand frame dimensions to have shape: [1, None,
None, 3]
    # i.e. a single-column array, where each item in the column has the pixel
RGB value
    ret, frame = video.read()
    frame_expanded = np.expand_dims(frame, axis=0)

    # Perform the actual detection by running the model with the image as input
```

```
    (boxes, scores, classes, num) = sess.run(
        [detection_boxes, detection_scores, detection_classes, num_detections],
        feed_dict={image_tensor: frame_expanded})

    # Draw the results of the detection (aka 'visulaize the results')
    vis_util.visualize_boxes_and_labels_on_image_array(
        frame,
        np.squeeze(boxes),
        np.squeeze(classes).astype(np.int32),
        np.squeeze(scores),
        category_index,
        use_normalized_coordinates=True,
        line_thickness=8,
        min_score_thresh=0.80)

    # All the results have been drawn on the frame, so it's time to display it.
    cv2.imshow('Object detector', frame)

    # Press 'q' to quit
    if cv2.waitKey(1) == ord('q'):
        break

# Clean up
video.release()
cv2.destroyAllWindows()
```

Output frame

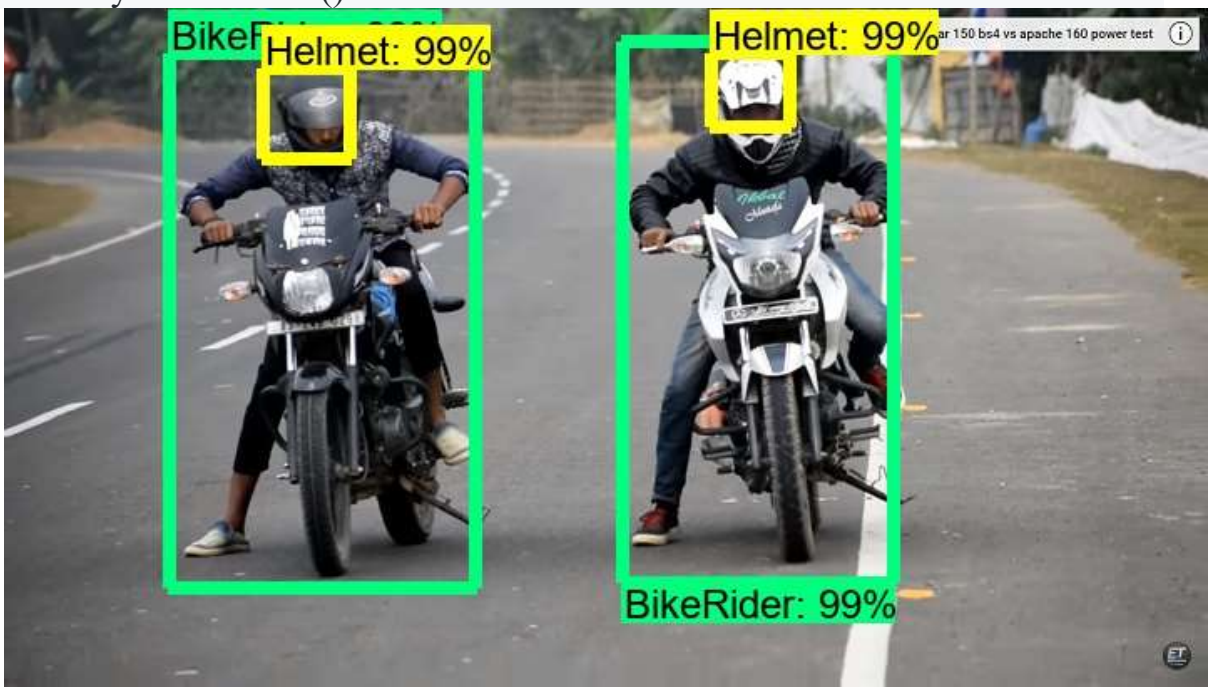
7.7.2 Bike-Rider Detection in Real Time

Using Webcam

```
import os
import cv2
import numpy as np
import tensorflow as tf
import sys

# This is needed since the notebook is stored in the object_detection folder.
sys.path.append("..")

# Import utilites
from utils import label_map_util
from utils import visualization_utils as vis_util

# Name of the directory containing the object detection module we're using
MODEL_NAME = 'inference_graph'

# Grab path to current working directory
CWD_PATH = os.getcwd()

# Path to frozen detection graph .pb file, which contains the model that is used
# for object detection.
PATH_TO_CKPT =
os.path.join(CWD_PATH,MODEL_NAME,'frozen_inference_graph.pb')

# Path to label map file
PATH_TO_LABELS = os.path.join(CWD_PATH,'training','labelmap.pbtxt')

# Number of classes the object detector can identify
NUM_CLASSES = 4

## Load the label map.
# Label maps map indices to category names, so that when our convolution
# network predicts `5`, we know that this corresponds to `king`.
# 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)
```

```
# Load the Tensorflow model into memory.
detection_graph = tf.Graph()
with detection_graph.as_default():
    od_graph_def = tf.GraphDef()
    with tf.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='')

    sess = tf.Session(graph=detection_graph)


# Define input and output tensors (i.e. data) for the object detection classifier

# Input tensor is the image
image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')

# Output tensors are the detection boxes, scores, and classes
# Each box represents a part of the image where a particular object was detected
detection_boxes = detection_graph.get_tensor_by_name('detection_boxes:0')

# Each score represents level of confidence for each of the objects.
# The score is shown on the result image, together with the class label.
detection_scores = detection_graph.get_tensor_by_name('detection_scores:0')
detection_classes = detection_graph.get_tensor_by_name('detection_classes:0')

# Number of objects detected
num_detections = detection_graph.get_tensor_by_name('num_detections:0')

# Initialize webcam feed
video = cv2.VideoCapture(0)
ret = video.set(3,1280)
ret = video.set(4,720)

while(True):

    # Acquire frame and expand frame dimensions to have shape: [1, None,
None, 3]
    # i.e. a single-column array, where each item in the column has the pixel
RGB value
    ret, frame = video.read()
    frame_expanded = np.expand_dims(frame, axis=0)
```

```
    # Perform the actual detection by running the model with the image as input
    (boxes, scores, classes, num) = sess.run(
        [detection_boxes, detection_scores, detection_classes, num_detections],
        feed_dict={image_tensor: frame_expanded})

    # Draw the results of the detection (aka 'visulaize the results')
    vis_util.visualize_boxes_and_labels_on_image_array(
        frame,
        np.squeeze(boxes),
        np.squeeze(classes).astype(np.int32),
        np.squeeze(scores),
        category_index,
        use_normalized_coordinates=True,
        line_thickness=8,
        min_score_thresh=0.60)

    # All the results have been drawn on the frame, so it's time to display it.
    cv2.imshow('Object detector', frame)

    # Press 'q' to quit
    if cv2.waitKey(1) == ord('q'):
        break

# Clean up
video.release()
cv2.destroyAllWindows()
```

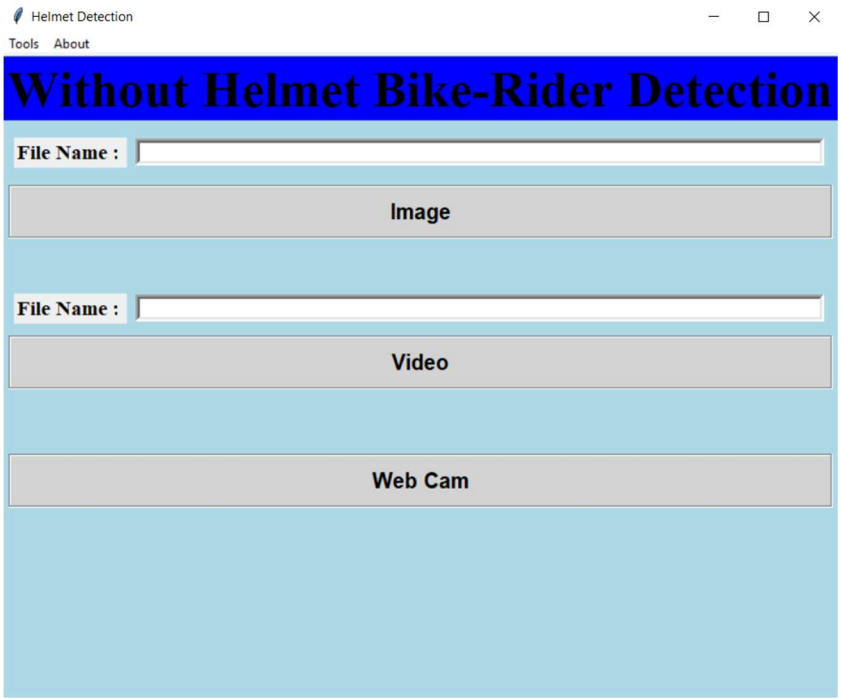


7.8 Without Helmet Bike-Rider Detection GUI

Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the tkinter package, and its extension, the **tkinter.tix** and the **tkinter.ttk** modules.

The **tkinter** package is a thin object-oriented layer on top of Tcl/Tk. To use tkinter, you don't need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. **tkinter** is a set of wrappers that implement the Tk widgets as Python classes. In addition, the internal module _tkinter provides a threadsafe mechanism which allows Python and Tcl to interact.

tkinter's chief virtues are that it is fast, and that it usually comes bundled with Python. Although its standard documentation is weak, good material is available, which includes: references, tutorials, a book and others. **tkinter** is also famous for having an outdated look and feel, which has been vastly improved in Tk 8.5. Nevertheless, there are many other GUI libraries that you could be interested in.



8. Challenges

However, in order to adopt such automatic solutions certain challenges need to be addressed:

- **Real-time Implementation*:*** Processing significant amount of information in a time constraint manner is a challenging task. As such applications involve tasks like segmentation, feature extraction, classification and tracking, in which a significant amount of information need to be processed in short duration to achieve the goal of real-time implementation.

- **Occlusion*:*** In real life scenarios, the dynamic objects usually occlude each other due to which object of interest may only be partially visible. Segmentation and classification become difficult for these partially visible objects.

- **Direction of Motion:** 3-dimensional objects in general have different appearance from different angles. It is well known that accuracy of classifiers depends on features used which in turn depends on angle to some extent. A reasonable example is to consider appearance of a bikerider from front view and side view.

- **Temporal Changes in Conditions:** Over time, there are many changes in environment conditions such as illumination, shadows, etc. There may be subtle or immediate changes which increase complexity of tasks like background modelling.

- **Quality of Video Feed:** Generally, CCTV cameras capture low resolution video. Also, conditions such as low light, bad weather complicate it further. Due to such limitations, tasks such as segmentation, classification and tracking become even more difficult. As stated in, successful framework for surveillance application should have useful properties such as *real-time performance, fine tuning, robust to sudden changes and predictive*. Keeping these challenges and desired properties in mind, we propose a method for automatic detection of bike-riders without helmet using feed from existing security cameras, which works in real time.

9.Conclusion

In this project we are trying to create a model that could detect bike rider in images or video feeds. The dataset that was used in this project was made and annotated so that the model could differentiate between image having bike rider or not.

The proposed bike rider detector has been successfully trained by using Faster R-CNN learning methods on the sample vehicle datasets and the vehicle detection process has been successfully performed by the trained vehicle detector being tested on the test data set.

In future, this model can be useful for project in which we detect a bike rider without helmet and recognize the licence plate of the bike so that e-challan could be generated.

10.References

- Romuere Silva, Kelson Aires, Thiago Santos, Kalyf Abdala, Rodrigo Veras, Andr Soares, Automatic detection of motorcyclists without helmet, Latin America Computing conference,2013.

- Prof. Chitte P.P. , Mr. Salunke Akshay S. , Mr. Thorat Aniruddha N , Mr. Bhosale Nilesh T ,Smart Helmet Intelligent Bike System, International Research Journal of Engineering and Technology (IRJET) Volume: 03 Issue: 05 - May-2016.

- Chung-Cheng Chiu, Min-Yu Ku, Hung-Tsung Chen ,Motorcycle Detection and Tracking Systern with Occlusion Segmentation, International Journal of Trend in Research andDevelopment..

- Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, " Hypertext Transfer Protocol HTTP," IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops), June 1999, pp. 45 - 46.

- Anshu Singh Gautam, Mayank Mishra, Mohita Prabhat, SMART HELMET SYSTEM, Journal of Emerging Technologies and Innovative Research (JETIR), April 2015, Volume 2, Issue 4 ,JETIR.