EasyChair Preprint

№ 1446

Driverless Car : Software Modelling and Design Using Python and Tensorflow

Poondru Prithvinath Reddy

September 1, 2019

DRIVERLESS CAR : SOFTWARE MODELLING AND DESIGN USING PYTHON AND TENSORFLOW

POONDRU PRITHVINATH REDDY

ABSTRACT

For autonomous vehicles, several real – time systems must work tightly together. These real – time systems include environmental mapping and understanding, localization, route planning and movement control. For these real – time systems to have a platform to work on, the self-driving car itself needs to be equipped with the appropriate software infrastructure. For this, the software programming architecture using ROS is presented. One of the key functions i.e. Object Detection is in self-driving cars. Also implementation of different object detection methods for detecting objects in images like Deep Learning and Deep Reinforcement Learning are presented. The Deep Reinforcement Learning uses a CNN and DQN where an agent extracts features and localize the object. The results of different techniques are comparable with high prediction accuracies.

INTRODUCTION

A **self-driving car**, also known as a **robot car**, a**utonomous car**, or **driverless car**, is a vehicle that is capable of sensing its environment and moving with little or no human input. Autonomous cars combine a variety of sensors to perceive their surroundings, such as radar, Lidar, sonar, GPS, odometry and inertial measurement units. Advanced control systems interpret sensory information to identify appropriate navigation paths, as well as obstacles .

CONCEPT OF AUTONOMOUS DRIVING

A car capable of autonomous driving should be able to drive itself without any human input. To achieve this, the autonomous car needs to sense its environment, navigate and react without human interaction. A wide range of sensors, such as LIDAR, RADAR, GPS, wheel odometry sensors and cameras are used by self-driving cars to perceive their surroundings. In addition, the autonomous car must have a control system that is able to understand the data received from the sensors and make a difference between

traffic signs, obstacles, pedestrian and other expected and unexpected things on the road .

For a machine to be called a robot, it should satisfy at least three important capabilities: to be able to sense, plan, and act . For a car to be called an autonomous car, it should satisfy the same requirements . Self-driving cars are essentially robot cars that can make decisions about how to get from point A to point B. The passenger only needs to specify the destination, and the autonomous car should be able to take him or her there safely.

SELF – DRIVING VEHICLES

The following sensors should be present in all self-driving cars:

Global positioning system (GPS). Global positioning system is used to determine the position of a self-driving car by triangulating signals received from GPS satellites . It is often used in combination with data gathered from an IMU and wheel odometry encoder for more accurate vehicle positioning and state using sensor fusion algorithms.

Light detection and ranging (LIDAR). A core sensor of a self-driving car, this measures the distance to an object by sending a laser signal and receiving its reflection . It can provide accurate 3D data of the environment, computed from each received laser signal. Self-driving vehicles use LIDAR to map the environment and detect and avoid obstacles .

Camera. Camera on board of a self-driving car is used to detect traffic signs, traffic lights, pedestrians, etc. by using image processing algorithms .

RADAR. RADAR is used for the same purposes as LIDAR. The advantages of RADAR over LIDAR are that it is lighter and has the capability to operate in different conditions .

Ultrasound sensors. Ultrasound sensors play an important role in the parking of self-driving vehicles and avoiding and detecting obstacles in blind spots, as their range is usually up to 10 meters .

Wheel odometry encoder. Wheel encoders provide data about the rotation of car’s wheels per second. Odometry makes use of this data, calculates the speed, and estimates the car’s position and velocity based on it. Odometry is often used with other sensor’s data to determine a car’s position more accurately.

Inertial measurement unit (IMU). An IMU consists of gyroscopes and accelerometers. These sensors provide data on the rotational and linear motion of the car, which is then used to calculate the motion and position of the vehicle regardless of speed .

On-board computer. This is the core part of any self-driving car. As any computer, it can be of varying power, depending on how much sensor data it has to process and how efficient it needs to be. All sensors connect to this computer, which has to make use of sensor’s data by understanding it, planning the route and controlling the car’s actuators.

The control is performed by sending the control commands such as steering angle, throttle and braking to the wheels, motors and servo of the autonomous car .

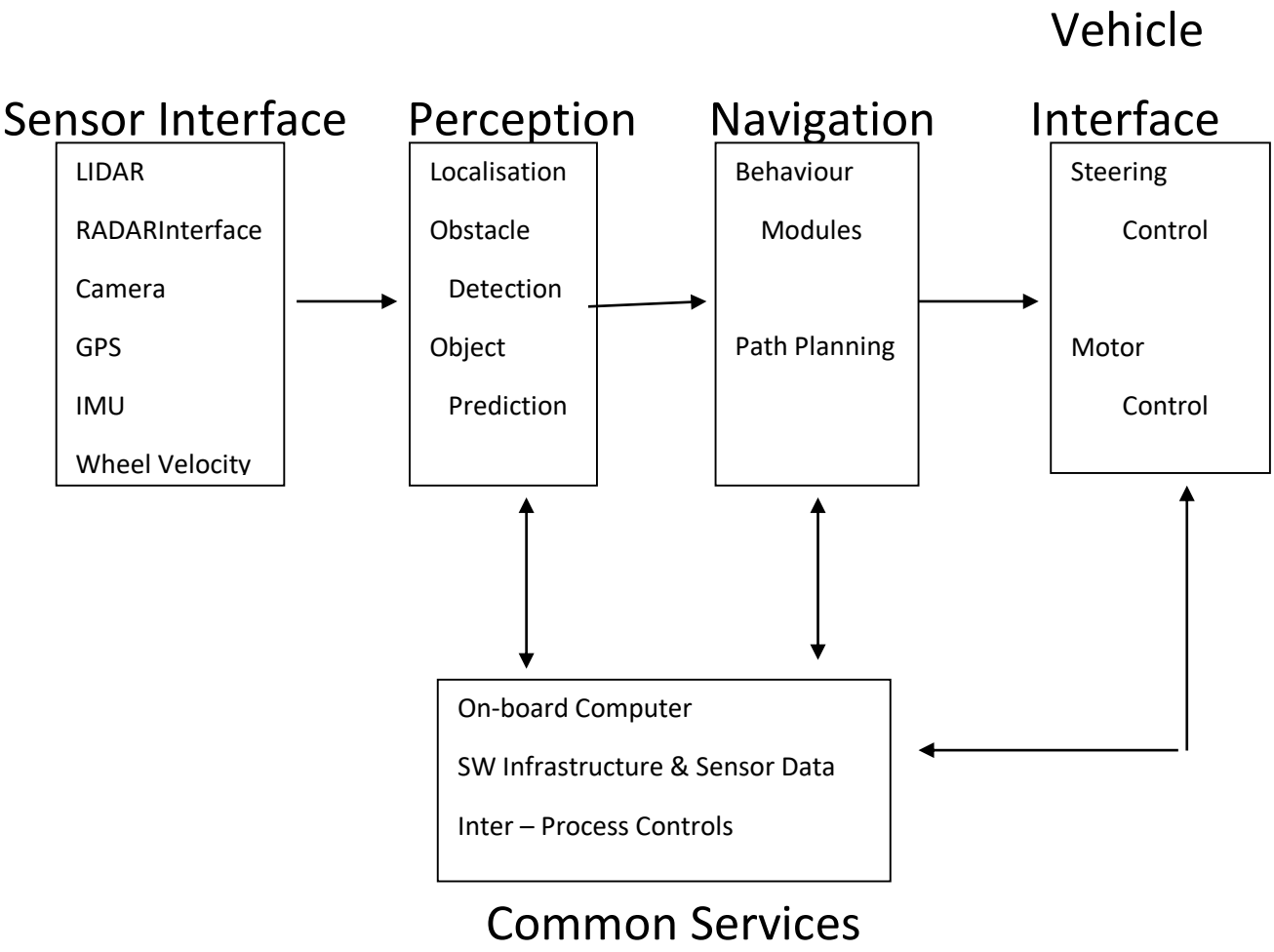

Figure 1 illustrates the SW block diagram of the standard self-driving car.

Each block seen in Figure 1 can interact with other blocks using inter-process communication (IPC) and identified the following blocks for the SW block diagram of a typical self-driving car:

Sensor interface modules. All communication between sensors and the car is performed in this block, as it enables data acquired from sensors to be shared with other blocks.

Perception modules. These modules process perception data from sensors such as LIDAR, RADAR and cameras, then segment the processed data to locate different objects that are staying still or moving.

Navigation modules. Navigation modules determine the behaviour of the self-driving car, as they have route and motion planners, as well as a state machine of car's behaviour .

Vehicle interface. This interface's goal is to send control commands such as steering, throttle and braking to the car after the path has been plotted in the navigation module.

Common services. Common services module controls the car's SW reliability by allowing logging and time-stamping of car's sensor data.

KEY TECHNOLOGY OF SELF-DRIVING CAR

The key technology of self-driving car into four parts according to the function of a self-driving car: Environment perception, Car navigation, Path planning, and Car control.

Car navigation system

During self-driving, two issues, which are the current location of the car and how to go from the location to the destination, must be resolved. However, in self-driving, the car must be able to automatically and intelligently locate its position and perform the path planning to destination. For this objective, the on-board car navigation system is deployed on the self-driving car.

Path planning

Map matching, which is the foundation of the path planning, calculates out the car's location by using the geographical information from GPS.

Environment perception

Environment perception is the key module of a self-driving car. To provide necessary information for a car's control decision, the car is required to independently perceive surrounding environment. The major methods of environment perception include laser navigation, visual navigation and radar navigation.

During environment perception, multi-sensors are deployed to sense the comprehensive information from the environment, which are then fused to perceive the environment. Among the sensors, the laser sensor is utilized for bridging between the real world and data world, radar sensor is used for distance perception and visual sensor is for traffic sign recognition. The self-driving car fuses data from laser sensors, radar sensors and visual sensors, and generates the surrounding environment perception, such as road edge stone, obstacles, road marking and so on.

By measuring the reflection time, reflection signal strength and the data of target point can be generated, then the testing object information, such as location (distance and angle), shape (size) and state (velocity and attitude) can be calculated out.

Car control system

Finally, the vehicle control system executes those instructions to control the vehicle's direction, speed, light, horn and so on.

The control platform controls the various systems of the vehicle, which includes the car anti-lock braking system, the car drive anti-slip system, the car electronic stability program, etc

FEATURES OF SELF-DRIVING CARS

Synergistic Combining of Sensors

All the data gathered by these sensors is collated and interpreted together by the car's CPU or in built software system to create a safe driving experience.

Programmed to Interpret Common Road Signs

The software has been programmed to rightly interpret common road behaviour and motorist signs. For example, if a cyclist gestures that he intends to make a manoeuvre, the driverless car interprets it correctly and slows down to allow the motorist to turn. Predetermined shape and motion descriptors are programmed into the system to help the car make intelligent decisions. For instance, if the car detects a 2 wheel object and determines the speed of the object as 10mph rather than 50 mph, the car instantly interprets that this vehicle is a bicycle and not a motorbike and behaves accordingly. Several such programs fed into the car's central processing unit will work simultaneously, helping the car make safe and intelligent decisions on busy roads.

Mapping in Advance

At the moment, before a self-driven car is tested, a regular car is driven along the route and maps out the route and it's road conditions including poles, road markers, road signs and more. This map is fed into the car's software helping the car identify what is a regular part of the road. As the car moves, its Velodyne laser range finder kicks in and generates a detailed 3D map of the environment at that moment. The car compares this map with the pre-existing map to figure out the non-standard aspects in the road, rightly identifying them as pedestrians and/or other motorists, thus avoiding them.

Programming Real Life Road Behaviour

While the vehicle does slow down to allow other motorists to go ahead, especially in 4 way intersections, the car has also been programmed to advance ahead if it detects that the other vehicle is not moving.

SOFTWARE PROGRAMMING

C++: General-Purpose Object-Oriented Programming Language

C++ is a programming language that is commonly used to program the onboard computer of autonomous vehicles. But it is one of the highest performance for programming on a Linux .

Linux: Open Source Operating Systems

Linux comes in many flavors, but is typically the operating system of choice for autonomous vehicle engineers. This is because it has a large, open-source community of people and tools.

The two versions of Linux we see most often are Ubuntu, since it works so well with the tools for autonomous vehicle development, and embedded Linux, which is a deterministic, Real-Time Operating System (RTOS) version of Linux.

Python: High-Level Programming Language

Python became so popular since it's easy to pick up and has a huge open-source community constantly developing tools for it. It is typically very easy to look at well written Python code and understand what's going on. Python is very popular with autonomous vehicle engineers because there are comprehensive libraries for math, science, data visualization, machine learning, AI, deep learning, etc. The disadvantage with Python is that it's a large, compiled language. This makes it unsuitable for very high performance applications.

Robot Operating System (ROS): Robots Middleware

ROS is an ecosystem of software libraries for robot development. Since autonomous vehicles are just large, wheeled robots, this tool makes developing autonomous vehicles significantly easier than it would otherwise be.

While ROS supports a few operating systems, it is commonly run on top of Ubuntu. People typically program ROS is C++ and/or Python. ROS is great for all the tools it includes for autonomous vehicles.

ROS is not an actual operating system, but rather a meta-operating system. Simply put, ROS works on top of other operating system and allows different processes to

communicate with each other during runtime. Therefore, it is necessary to know ROS to develop autonomous vehicles.

ROS is highly interesting for development of autonomous driving. Because autonomous cars are robot cars, the same type of programs used to control robots can be used to control autonomous cars.

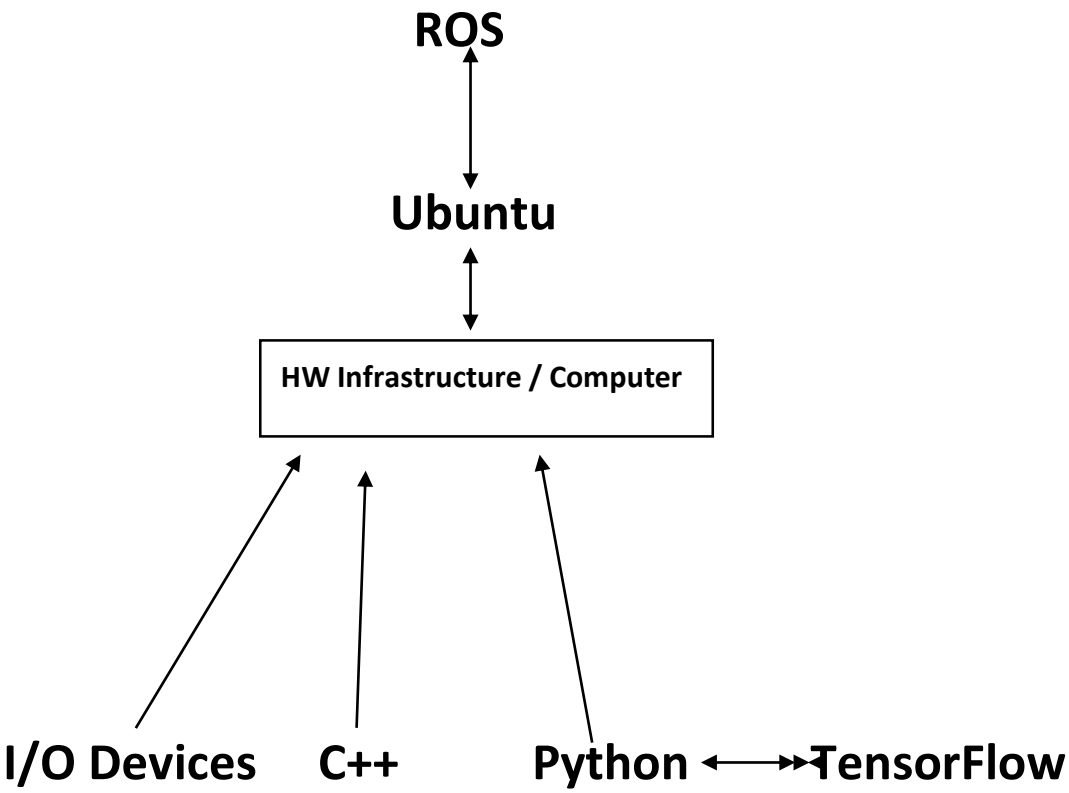

Figure 2: Software System Running ROS

TENSORFLOW

TensorFlow is a software framework for building and deploying machine learning models. It provides the basic building blocks to design, train, and deploy machine

learning models. Mostly it is famous for building deep neural networks such as image recognition, speech recognition and translation, image style transfer…etc.

In machine learning problems, we deal with a large amount of multi-dimensional data. This multi-dimensional data is to be stored in a multi-dimensional array called tensors. TensorFlow is designed to work on tensors with different individual attributes. In TensorFlow, we build a computational graph similar to a flowchart that determines how data or tensors will flow through the system.

DRIVERLESS CAR SOFTWARE DEVELOPMENT

Fully automatic mode of operation is too challenging, with which software can deal only in particular circumstances. In the urban environment saturated with traffic, far from being ideal roads and unpredictable drivers, it's too early to rely on the autopilot. Here, the main task faced by driverless cars software developers is to make the product that will adapt to external environmental factors as quickly as possible.

Another global trend of the self driving cars software development is the integration of navigation systems into a common information field. The vehicles will be able to obtain information not only from satellites, but also from other cars and even city infrastructure.

OBJECT DETECTION

Object Detection is the process of finding real-world object instances like car, bike, and humans in still images or Videos. It allows for the recognition, localization, and detection of multiple objects within an image which provides us with a much better understanding of an image as a whole.

Methods for object detection generally fall into three categories. For Machine Learning approaches, it becomes necessary to first define features, 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 object detection without specifically defining features, and are based on convolutional neural networks (CNN). Also the reinforcement learning methods are based on CNN and Q-learning features.

- Machine Learning approaches:
    - Viola–Jones object detection framework
    - Scale-invariant feature transform (SIFT)
    - Histogram of oriented gradients (HOG)
- Deep Learning approaches:
    - Region Proposals (R-CNN, Fast R-CNN, Faster R-CNN)
    - Single Shot MultiBox Detector (SSD)

    - You Only Look Once (YOLO)
- Reinforcement Learning approaches:
    - Q-Learning
    - Deep Q-Learning

We present two different methods of Object Detection implementation. One uses deep learning approach and the other based on deep reinforcement learning technique.

One of the applications Of Object Detection is in Self Driving Cars.

OBJECT DETECTION USING TENSORFLOW

Deep learning is a subfield of machine learning that is a set of algorithms that is inspired by the structure and function of the brain. We can use the TensorFlow library do to numerical computations, but these computations are done with data flow graphs. In these graphs, nodes represent mathematical operations, while the edges represent the data, which usually are multidimensional data arrays or tensors, that are communicated between these edges.

In this Object Detection application, we'll focus on Deep Learning Object Detection as Tensorflow uses Deep Learning for computation.

Every Object Detection Algorithm has a different way of working, but they all work on the same principle. Feature Extraction: They extract features from the input images at hands and use these features to determine the class of the image. Be it through MatLab, Open CV, Viola Jones or Deep Learning.

OBJECT DETECTION IMPLEMENTATION

To begin, we are going to want to make sure we have TensorFlow and all of the dependencies. For CPU TensorFlow, we can just do pip install tensorflow,

For all the other libraries we can use pip or conda to install them. The code is provided below

```
pip install  pillow

pip install  lxml

pip install  jupyter

pip install  matplotlib
```

Next, we have Protobuf: Protocol Buffers (Protobuf) and need to Download Protobuf version 3.4 or above and extract it.

Head to the protoc releases page and download the protoc-3.4.0-win32.zip, extract it, and will find protoc.exe in the bin directory.

Next click the green "clone or download" button on the https://github.com/tensorflow/models page, download the .zip, and extract it.Next, we need to clone the github. We can do this with git, or we can just download the repository to .zip: Once we have the models directory (or models-master if we downloaded and extracted the .zip), navigate to that directory in our terminal/cmd.exe.

After the environment is set up, we need to go to the “object_detection” directory and then create a new python file. We can use Spyder or Jupyter to write the code.

First of all, we need to import all libraries.

```
import numpy as np

import os

import six.moves.urllib as urllib

import sys

import tarfile

import tensorflow as tf

import zipfile

from collections import defaultdict

from io import StringIO

from matplotlib import pyplot as plt

from PIL import Image


sys.path.append("..")

from object_detection.utils import ops as utils_ops

from utils import label_map_util

from utils import visualization_utils as vis_util
```

Next, we will download the model which is trained on the COCO dataset.

Next, we provide the required model and the frozen inference graph generated by Tensorflow to use.

```
MODEL_NAME = 'ssd_mobilenet_v1_coco_2018_01_28'

MODEL_FILE = MODEL_NAME + '.tar.gz'

DOWNLOAD_BASE = 'http://download.tensorflow.org/models/object_detection/'

 PATH_TO_FROZEN_GRAPH = MODEL_NAME + '/frozen_inference_graph.pb'
```

```
PATH_TO_LABELS = os.path.join('data', 'mscoco_label_map.pbtx')
```

This code will download that model from the internet and extract the frozen inference graph of that 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.GraphDef()
  with tf.gfile.GFile(PATH_TO_FROZEN_GRAPH, 'rb') as fid:
    serialized_graph = fid.read()
    od_graph_def.ParseFromString(serialized_graph)
    tf.import_graph_def(od_graph_def, name='')
```

Next, we are going to load all the labels

```
category_index = label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS,
use_display_name=True)
```

Now we will convert the images data into a numPy array for processing.

```
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)
```

The path to the images for the testing purpose is defined here.

```
PATH_TO_TEST_IMAGES_DIR = 'test_images'
TEST_IMAGE_PATHS = [ os.path.join(PATH_TO_TEST_IMAGES_DIR, 'image{}.jpg'.format(i)) for i in
range(1, 8) ]
IMAGE_SIZE=(12,8)
```

This code runs the inference for a single image, where it detects the objects, make boxes and provide the class and the class score of that particular object.

```
def run_inference_for_single_image(image, graph):
  with graph.as_default():
    with tf.Session() as sess:
      Get handles to input and output tensors
      ops = tf.get_default_graph().get_operations()
      all_tensor_names = {output.name for op in ops for output in op.outputs}
      tensor_dict = {}
      for key in [
          'num_detections', 'detection_boxes', 'detection_scores',
          'detection_classes', 'detection_masks'
      ]:
        tensor_name = key + ':0'
        if tensor_name in all_tensor_names:
          tensor_dict[key] = tf.get_default_graph().get_tensor_by_name(
            tensor_name)
      if 'detection_masks' in tensor_dict:
        The following processing is only for single image
        detection_boxes = tf.squeeze(tensor_dict['detection_boxes'], [0])
        detection_masks = tf.squeeze(tensor_dict['detection_masks'], [0])
        Reframe is required to translate mask from box coordinates to image coordinates and fit the image size.
        real_num_detection = tf.cast(tensor_dict['num_detections'][0], tf.int32)
        detection_boxes = tf.slice(detection_boxes, [0, 0], [real_num_detection, -1])
        detection_masks = tf.slice(detection_masks, [0, 0, 0], [real_num_detection, -1, -1])
        detection_masks_reframed = utils_ops.reframe_box_masks_to_image_masks(
            detection_masks, detection_boxes, image.shape[0], image.shape[1])
        detection_masks_reframed = tf.cast(
            tf.greater(detection_masks_reframed, 0.5), tf.uint8)
        Need to follow the convention by adding back the batch dimension
        tensor_dict['detection_masks'] = tf.expand_dims(
```

```
      detection_masks_reframed, 0)
    image_tensor = tf.get_default_graph().get_tensor_by_name('image_tensor:0')
    Run inference
    output_dict = sess.run(tensor_dict,
      feed_dict={image_tensor: np.expand_dims(image, 0)})
    All outputs are float32 numpy arrays, so required to convert types as appropriate
    output_dict['num_detections'] = int(output_dict['num_detections'][0])
    output_dict['detection_classes'] = output_dict[
     'detection_classes'][0].astype(np.uint8)
    output_dict['detection_boxes'] = output_dict['detection_boxes'][0]
    output_dict['detection_scores'] = output_dict['detection_scores'][0]
    if 'detection_masks' in output_dict:
     output_dict['detection_masks'] = output_dict['detection_masks'][0]
return output_dict
```

Our Final loop, which will call all the functions defined above and will run the inference on all the input images one by one, which will provide us the output of images in which objects are detected with labels along with the percentage/score of that object.

OBJECT DETECTION WITH DEEP REINFORCEMENT LEARNING

Object Detection in an Image Applying Deep Q-Network

Object detection refers to drawing a bounding box around the most specific location of an object in an image. There is a reinforcement learning agent that autonomously performs this task after adequate training. The algorithm can be broadly classified into two steps- CNN, and DQN.

A pretrained CNN extracts feature from the proposed region.

The Deep – Q Network is built on top of the CNN that determines the optimal transformation of the bounding box so that the desired object is detected in as less number of steps as possible.

Generally speaking, DRL considers a single agent in a stationary environment, namely single agent deep reinforcement learning (SADRL). A traditional RL problem can be described as a Markov decision process (MDP), in which the agent aims to make a

sequence decisions. RL provides a coherent framework, an agent can learn from an environment a policy function that maps states to actions and take actions in order to maximize its expected cumulative rewards at each discrete time step.

Formally, the MDP has a set of actions A, a set of states S, and a reward function R. and this section presents details of these three components.

Localization Actions

The set of actions A is composed of eight transformations that can be applied to the box and one action to terminate the search process. These actions are organized in four sub-sets: actions to move the box in the horizontal and vertical axes, actions to change scale, and actions to modify aspect ratio. In this way, the agent has four degrees of freedom to transform the box during any interaction with the environment.

A box is represented by the coordinates in pixels of its two corners: b = [x1,y1,x2,y2]. Any of the transformation actions make a discrete change to the box by a factor relative to its current size in the following way: $\alpha w = \alpha * (x2 - x1)$ $\alpha h = \alpha * (y2 - y1)$ (1)

Where $\alpha \in [0,1]$. Then, the transformations are obtained by adding or removing αw or αh to the x or y coordinates, depending on the desired effect. For instance, a horizontal move to the right adds αw to x1 and xy, while decreasing aspect ratio subtracts αw from x1, and adds it to x2. Note that the origin in the image plane is located in the top-left corner. We set $\alpha = 0.2$ in all our experiments, since this value gives a good trade-off between speed and localization accuracy. In early exploration experiments we noticed that smaller values make the agent slower to localize objects, while larger values make it harder to place the box correctly.

Finally, the only action that does not transform the box is a trigger to indicate that an object is correctly localized by the current box. This action terminates the sequence of the current search, and restarts the box in an initial position to begin the search for a new object. The trigger also modifies the environment.

State

The state representation is a tuple (o,h), where o is a feature vector of the observed region, and h is a vector with the history of taken actions. The set of possible states S is very large as it includes arbitrary boxes from a large set of images, and is expanded with all the combinations of actions that lead to those boxes.

The feature vector o is extracted from the current region using a pre-trained CNN following the architecture of Zeiler and Fergus 6. Any attended region by the agent is warped to match the input of the network (224×224) regardless of its size and aspect ratio. We also expand the region to include16 pixels of context around the original box. We forward the region up to the layer 6 (fc6) and use the 4,096 dimensional feature vector to represent its content.

The history vector h is a binary vector that informs which actions have been used in the past. Each action in the history vector is represented by a 9-dimensional binary vector, where all values are zero except the one corresponding to the taken action. The history vector encodes 10 past actions, which means that $h \in R90$. Although h is very low-dimensional compared to o, it has enough energy to inform what has happened in the past.

Reward Function

The reward function R is proportional to the improvement that the agent makes to localize an object after selecting a particular action. Improvement in our setup is measured using the Intersection-over-Union (IoU) between the target object and the predicted box at any given time. More specifically, the reward function is estimated using the differential of IoU from one state to another.

Let b be the box of an observable region, and g the ground truth box for a target object. Then, IoU between b and g is defined as $IoU(b,g) = area(b \cap g)/area(b \cup g)$.

The reward function $R_a(s,s')$ is granted to the agent when it chooses the action a to move from state s to s'. Each state s has an associated box b that contains the attended region. Then, the reward is as follows : $R_a(s,s') = sign(\ IoU(b',g) - IoU(b,g)\ )$ (2)

Intuitively, equation 2 says that the reward is positive if IoU improved from state s to state s', and negative other-wise. This reward scheme is binary $r \in \{-1,+1\}$, and applies to any action that transforms the box. Without quantization, the difference in IoU is small enough to confuse the agent about which actions are good or bad choices. In this way, the agent pays a penalty for taking the box away the target, and is rewarded to keep the target object in the visible region until there is no other transformation that improves localization. In that case, the best action to choose should be the trigger.

The trigger has a different reward scheme because it leads to a terminal state that does not change the box, and thus, the differential of IoU will always be zero for this action. The reward for the trigger is a thresholding function of IoU as follows:

$R_\omega(s,s') = \{+\eta$ if $IoU(b,g) \geq \tau$. $-\eta$ otherwise (3) where ω is the trigger action, η is the trigger reward, set to 3.0 in our experiments, and τ is a threshold that indicates the minimum IoU allowed to consider the attended region as a positive detection. The standard threshold for object detection evaluation is 0.5, but we used $\tau = 0.6$ during training to encourage better localization. A larger value for τ has a negative effect in performance because the agent learns that only clearly visible objects are worth the trigger and tends to avoid truncated or naturally occluded objects.

Localization Policy

The core problem is to find a policy that guides the decision making process of the agent. A policy is a function $\pi(s)$ that specifies the action a to be chosen when the current state is s. Since we do not have the state transition probabilities and the reward

function is data-dependent, the problem is formulated as a reinforcement learning problem using Q-learning.

In this work, we follow the deep Q-learning algorithm proposed by Mnih et al. 5. This approach estimates the action-value function using a neural network, and has several advantages over previous Q-learning methods.

Q-learning

We use a Q-network that takes as input the state representation and gives as output the value of the nine actions presented. We train category specific Q-networks following the architecture illustrated in Figure 3. Notice that in our design we do not learn the full feature hierarchy of the convolutional network; instead, we rely on a pre-trained CNN.

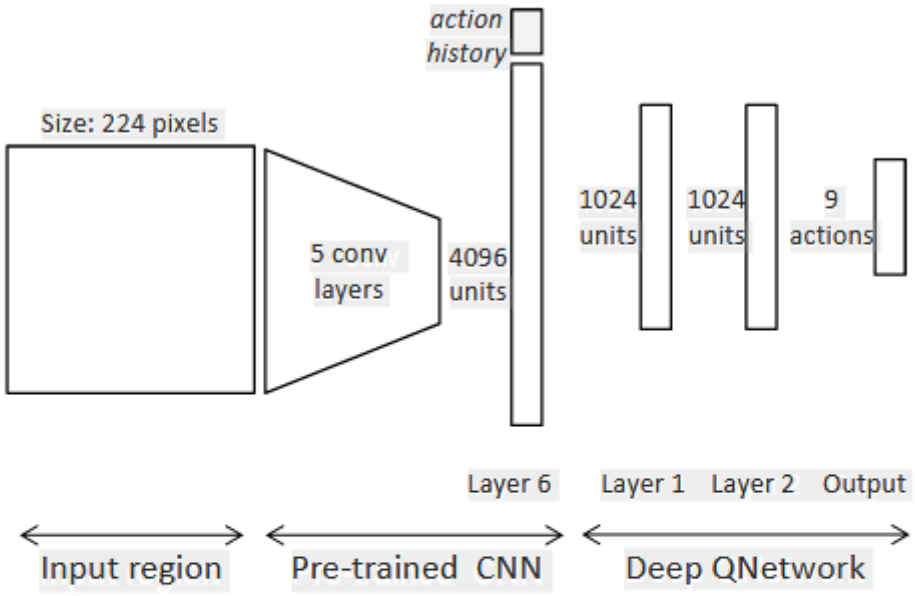

Figure 3. Architecture of the proposed QNetwork. The input region is first warped to 224 × 224 pixels and processed by a pre-trained CNN with 5 convolutional layers and 1 fully connected layer. The output of the CNN is concatenated with the action history vector to complete the state representation. It is processed by the Q-network which predicts the value of the 9 actions.

Training Localization Agents

The policy followed during training is ϵ-greedy ,which gradually shifts from exploration to exploitation according to the value of ϵ . During exploration, the agent selects actions randomly to observe different transitions and collects a varied set of experiences. During exploitation, the agent selects actions greedily according to the learned policy, and learns from its own successes and mistakes.

Then, during exploration, the agent is allowed to choose one random action from the set of positive actions. Notice that for one particular state s, there might be multiple positive actions because there is no single path to localize an object. Using this strategy, the algorithm terminates in a small number of epochs.

The ϵ-greedy training strategy is run for 15 epochs, each completed after the agent has had interaction with all training images. During the first 5 epochs,ϵ is annealed linearly from 1.0 to 0.1 to progressively let the agent use its own learned model. After the fifth epoch,ϵ is fixed to 0.1 so the agent adjusts the model parameters from experiences produced by its own decisions. The parameters are updated using stochastic gradient descent and the back-propagation algorithm, and we also use dropout regularization .

Testing a Localization Agent

Once an agent is trained, it learns to attend regions that contain objects of the target category. Since we do not know the number of objects present in a single image beforehand, we let the agent run for a maximum of 200 steps, so only 200 regions are evaluated per test image. We simplified the model with a minimum set of actions to act locally in time.

At each step, the agent makes a decision to transform the current box or selects the trigger to indicate that an object has been found. When the trigger is used, the search for other objects continues from a new box that covers a large portion of the image. The search for objects is restarted from the beginning due to two possible events: the agent used the trigger, or 40 steps passed without using the trigger. We found that 40 steps are enough to localize most objects, even the smaller ones, and when it takes longer is usually because the agent is stuck searching in an ambiguous region. Restarting the box helps to take a new perspective of the scene. The very first box covers the entire scene, and after any restarting event, the new box has a reduced size set to 75% of the image size, and is placed in one of the four corners of the image always in the same order.

Experiment and Results

The training of the backbone CNN(Zeiler & Fergus, 2014) and a simplified version of the DQN(Mnih et al., 2015) building upon the CNN was done. The training was done with Keras library, using the Tensorflow as the backend, on Python. The implemented algorithm is tested on the PASCAL VOC 2012 and PASCAL VOC 2007 dataset.

The complete object detection algorithm has two parts : CNN and DQN. We present the results of the two separate parts to validate their correctness.

By inspection of the obtained results for the CNN, we see that some images data was discarded due to having multiple labels. In particular, bottles, buses, chairs, cows, tables and plant have slightly lower prediction accuracies. Also, the percentage classification accuracy is varying for different objects.

The results for the DQN are given in terms of IoU. Having the average IoU of the predicted bounding boxes in the range 0.5 – 0.8 indicates that the algorithm is able to cover at least some portion of the bounding box most of the time.

CONCLUSION

We are focusing, primarily, on the IT solutions for driverless cars, leaving everything like vehicle motor, chassis, and car gears behind. Here, the main task faced by driverless cars software developers is to make the product that will adapt to external environmental factors as quickly as possible. An overview of autonomous vehicles components and software architecture using ROS is presented. Also different methods for detecting objects in images is presented which is a key function of self-driving cars. It is observed that the results of different techniques are comparable with good prediction rates.

REFERENCES

1. **https://github.com/**
2. *https://github.com/tensorflow/models/blob/master/.../object_detection_tutorial.ipynb*
3. Jason Marks : "What Software Do Autonomous Vehicle Engineers Use?" Part 1
   URL. https://medium.com/
4. J.C.Caicedo, S.Lazebnik : "Active Object Localization with Deep Reinforcement Learning" In The IEEE International Conference on Computer Vision (ICCV), December 2015.
5. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M.G.Bellemare,A.Graves,M.Riedmiller,A.K.Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. Nature, 518:529–533, 2015.
6. M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In ECCV. Springer, 2014.