Lane Keeping Assistance of Autonomous Vehicles using Reinforcement Learning and Deep Learning on ROS

Authors

Karim Mohamed Ibrahim	201301883
Mohamed Ahmed Mohamed Ali	201304242
Khaled Hefnawy Mohamed	201300812
Rohanda Hamed El Sayed	201304126
Mohamed Alaa El-din Farghaly	201305056

Under Supervision of

Dr. Hazem M.Abbas

Eng. Mohammed Abdou

Communication and Information Engineering Department

University of Science and Technology at Zewail City

2018

Abstract

In this project we use several algorithms to tackle the problem of controlling an autonomous vehicle in the tasks of lane keeping and crash avoidance. The main simulation environment is ROS where we implemented a vehicle model equipped with a realistic laser sensor and a camera. Two separate simulation scenarios are designed for each task, one scenario is for training the algorithms and the other is to test the performance of the trained models. Three main learning algorithms are used. A discrete reinforcement learning algorithm called Q-learning, A continuous reinforcement learning algorithm called DDPG. The previous algorithms used laser sensor readings as input. The final algorithm, which is a supervised deep learning algorithm that uses an architecture of a Convolutional Neural Network, relied on images fed from the camera as input. A comparative study of the results is done based on specific evaluation criteria.

Acknowledgement

We have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals and organizations. We would like to extend our sincere thanks to all of them.

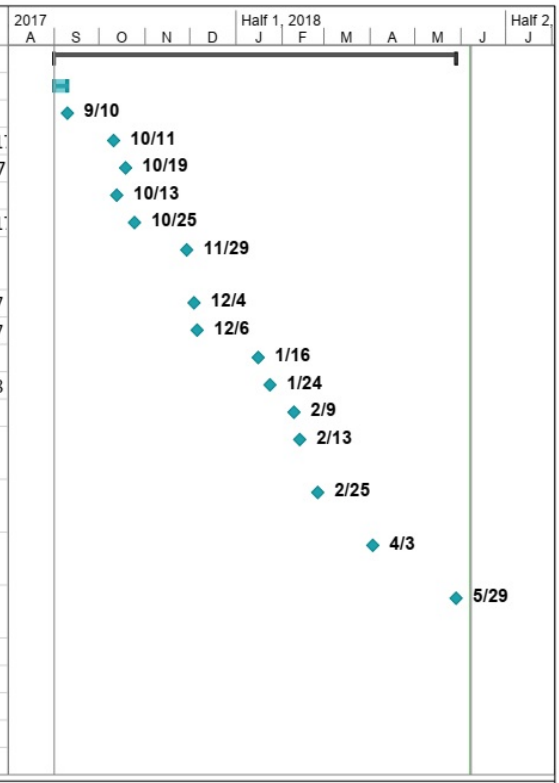
We are highly indebted to Dr. Hazem M.Abbas and Eng. Mohammed Abdou for their guidance and constant supervision as well as for providing necessary information regarding the project and also for their support in completing the project.

We would like to express our gratitude towards members of Valeo Egypt for their kind co-operation and encouragement which helped us in completion of this project.

Finally, this project would have never seen the light without the help of the global community of open source education, especially mentioning Prof. Andrew Ng, Prof. Andrej Karpathy, Prof. David Silver and Prof. Richard Sutton.

Gantt Chart

ID	Task Mode	Task Name	Duration	Start
1		**All meetings**	**192 days**	**Fri 9/1/17**
2		Problem study	7 days	Fri 9/1/17
3		proposal sent to work with Valeo	0 days	Sun 9/10/17
4		online meeting (chat report)	0 days	Wed 10/11/1
5		Eng Abdo Master master presentation	0 days	Thu 10/19/17
6		proposal to work with Valeo is acceptec	0 days	Fri 10/13/17
7		Meeting at Valeo	0 days	Wed 10/25/1
8		Graduation project proposal to the supervisor professor	0 days	Wed 11/29/17
9		Cancelled meeting at Valeo	0 days	Mon 12/4/17
10		Meeting at valeo	0 days	Wed 12/6/17
11		Team members meeting	0 days	Tue 1/16/18
12		Meeting at valeo	0 days	Wed 1/24/18
13		progress report meeting at uni	0 days	Fri 2/9/18
14		Progress meeting with supervisor professor	0 days	Tue 2/13/18
15		Progress meeting with supervisor professor	0 days	Sun 2/25/18
16		Progress meeting with supervisor professor	0 days	Tue 4/3/18
17		Progress meeting with supervisor professor	0 days	Tue 5/29/18
18				
19				
20				
21				
22				

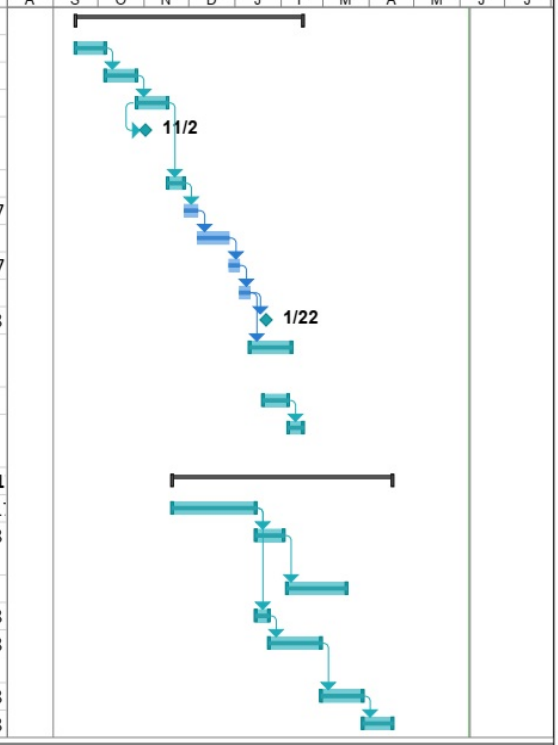


Project: Project1.mpp
Date: Thu 6/7/18

Task		Inactive Summary		External Tasks	
Split		Manual Task		External Milestone	
Milestone		Duration-only		Deadline	
Summary		Manual Summary Rollup		Progress	
Project Summary		Manual Summary		Manual Progress	
Inactive Task		Start-only			
Inactive Milestone		Finish-only			

Page 1

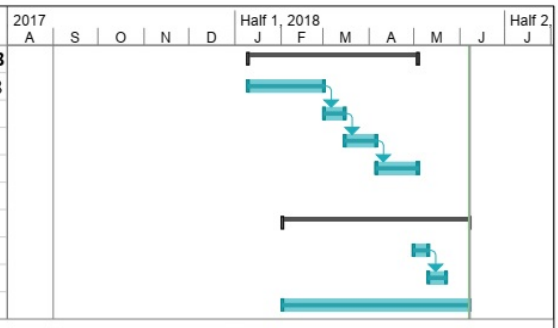
ID	Task Mode	Task Name	Duration	Start
23		**Building scenarios(simulation)**	**109 days**	**Sat 9/16/17**
24		prepare car model on ROS	15 days	Sat 9/16/17
25		build ackerman model for steering	15 days	Fri 10/6/17
26		Design training track on blender	15 days	Fri 10/27/17
27		first simulation of track designed and car model	0 days	Thu 11/2/17
28		Insert car and track on ROS	7 days	Fri 11/17/17
29		Add Informed action to control the car	7 days	Tue 11/28/17
30		Modify car model	15 days	Thu 12/7/17
31		Design testing track on blender	5 days	Thu 12/28/17
32		Insert testing track on ROS	5 days	Thu 1/4/18
33		Training & testing tracks final output	0 days	Mon 1/22/18
34		prepare simulation system for RL control	20 days	Thu 1/11/18
35		Add camera to simulation	12 days	Sat 1/20/18
36		prepare simualtion system for CNN algorithms	8 days	Tue 2/6/18
37		**RL Algorithms**	**106 days**	**Mon 11/20/1**
38		Reinforcement course lectures	41 days	Mon 11/20/1
39		Q-learning testing algorithm on maze problem	15 days	Mon 1/15/18
40		develop Q-learning on ROS sim	30 days	Mon 2/5/18
41		Run DDPG Code on TORCS	7 days	Mon 1/15/18
42		Understand DDPG optimized code on TORCS	25 days	Wed 1/24/18
43		DDPG code(steering) on ROS sim	20 days	Wed 2/28/18
44		DDPG code (steering and acceleration)	14 days	Wed 3/28/18



Project: Project1.mpp
Date: Thu 6/7/18

Task
Split
Milestone
Summary
Inactive Summary
Manual Task
Duration-only
Manual Summary Rollup
External Tasks
External Milestone
Deadline
Progress

ID	Task Mode	Task Name	Duration	Start
45		**CNN**	**82 days**	**Wed 1/10/18**
46		Deep Learning courses studying	37 days	Wed 1/10/18
47		Prepare CNN algorihms	10 days	Fri 3/2/18
48		Implment CNN algorithm	15 days	Fri 3/16/18
49		Tune CNN and debug	20 days	Fri 4/6/18
50				
51		**Other**	**90 days**	**Fri 2/2/18**
52		Choose Comparison & Evaluation Criter	8 days?	Tue 5/1/18
53		Evaluate the four algorithms	8 days	Fri 5/11/18
54		Report	90 days	Fri 2/2/18



Project: Project1.mpp
Date: Thu 6/7/18

Task
Split
Milestone
Summary
Project Summary
Inactive Task
Inactive Milestone
Inactive Summary
Manual Task
Duration-only
Manual Summary Rollup
Manual Summary
Start-only
Finish-only
External Tasks
External Milestone
Deadline
Progress
Manual Progress

List of Tables

List of Figures

Nomenclature

Ackermann Model

ϕ_{inner} the angle that the steering of the inner front wheel will turn with measured from the vertical axis

ϕ_{outer} the angle that the steering of the outer front wheel will turn with measured from the vertical axis

θ the angle the whole car should turn with

a the angle that the steering of the outer front wheel will turn with measured from the horizontal axis

b the angle that the steering of the inner front wheel will turn with measured from the horizontal axis

H the distance between the back wheels and the front one

$speed_{inner}$ the speed of the inner back wheel

$speed_{outer}$ the speed of the outer back wheel

W the distance between the back wheels

Reinforcement learning

α learning-rate parameter

ϵ probability of taking a random action in an ϵ greedy policy

γ discount-rate parameter

π policy, decision making rule

a action

$A(s)$ set of all actions possible in state s

Q, Q_t array estimate of action-value function $q_\pi(s, a)$

$q_\pi(s, a)$ action-value function of taking action a in state s under policy π

$q_\star(s, a)$ action-value function of taking action a in state s under optimal policy

R_t Reward at time t

s state

V, V_t array estimate of state-value function $v_\pi(s)$

$v_\pi(s)$ state-value function of state s under policy π

$v_\star(s)$ state-value function of state s under optimal policy

Contents

1 Introduction

Lane detection systems have been investigated for more than 20 years (Gurghian et al., 2016). Research in this field has been conducted in the aim of reducing the number of accidents that are caused by unintended lane departure due to driver inattention to the surrounding cars. Early researches aimed at creating a warning system that alerts the driver for lane deviation and may in some emergency cases interfere to take a control action (Narote et al., 2018).

This is called lane departure warning system where the position of the vehicle relative to the lane markings is being monitored from both sides and if the vehicle approaches the markers, the driver is being warned by the system to take the appropriate corrective action.The building blocks for the LDW system are presented in figure 1 .Many lane detection methods are based on computer vision techniques incorporating hand-crafted features to output a mathematical model for the lane markers. This was found to have many disadvantages (Narote et al., 2018). First, the created mathematical models are not scalable (i.e. could not be generalized over a big number of lane markers variations). Second, in order to determine the vehicle position, accurate segmentation is needed which is computationally expensive.

As the industry is moving forward from assisted driving to shorty autonomous driving, there is an increasing need for the implementation of new algorithms that are robust,scalable and timely optimized.Here comes the role of deep learning where no need for computationally expensive segmentation and where the generated model could be easily generalized over wide variety of lane markers and environmental conditions; this is done through implicitly learning the efficient features of the application at hand.

Motivation and Related Work The study of effectiveness of deep learning in autonomous driving has started as early as 1989 when (Pomerleau, 1989) built the Autonomous Land Vehicle in a Neural Network (ALVINN) system and demonstrated that an end-to-end trained neural network can indeed steer a car on public roads. At that early time, there was a lack of labeled data to train the vehicle and a shortage in computational power.Then starting from 2005,with the advancement of parallel graphics processing units, many efforts have come to light such as Dave (LeCun et al., 2005) , an RC vehicle which used a complicated neural network for object avoidance,DAVE II, created by NVIDIA (Bojarski et al., 2016) , an end-to-end deep learning system which managed to drive autonomously 98% of the time in their trials on highways and recently in (Innocenti et al., 2017) ,inspired by the work of NVIDIA , used deep neural network for Imitation Learning which succeeded in positioning the car within the lane 99% of the time.

Possible limitations of the previous work are that they used multiple sensors(Camera ,LIDAR, stereo cameras) to acquire the data and to increase model robustness which is very costly ,also they kept a constant speed in training the car outputting only the steering angle. The first limitation was circumvented by (Innocenti et al., 2017)

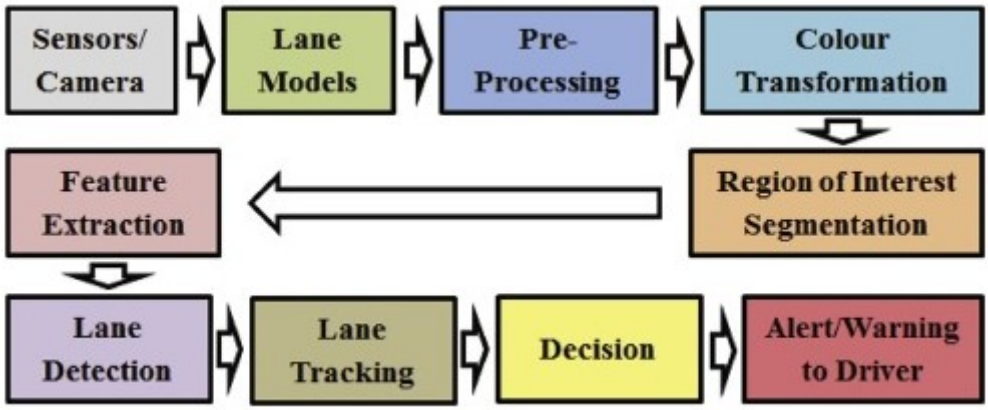


Figure 1: Block diagram for lane detection and departure system (Narote et al., 2018)

who managed to achieve high positioning accuracy using a single front Camera.

Our breakthrough in this project is the integration of two parts, the simulation and the algorithms. For the simulation, ROS is used which is a realistic simulator for robots and autonomous vehicles. The power of ROS lies in its capability of simulating the movement of a real car and the sensors connected to it. For the algorithms, three algorithms are used to approach the lane keeping task, CNN, Q-learning and DDPG. It is meant to output a model that is capable of behaving autonomously without any human interference. The output model can make the car keep its lane and avoid crashing into the roadsides. A comparative study is made between the results of all used algorithms based on specific evaluation criteria. The uniqueness of our work lies in the integration between a realistic simulator and state-of-art control algorithms to produce fully functional autonomous control models in both steering and acceleration control.

2 Literature Review

Previous work on Autonomous control Several implementations of autonomous vehicles control using AI driven algorithms are available in literature in both simulation and real environments. The DAVE project was an attempt to develop an end-to-end control solution for a RC-vehicle designed for outdoor missions (DARPA-IPTO, 2004) . A small network was trained using data acquired from human interactions. The RC-car managed to make a crash-free 20 meters drive. The project went on to continue with the aim of learning long-range vision for autonomous driving using more advanced hardware (Hadsell et al., 2009). As for DAVE2 developed by NVIDIA (Bojarski et al., 2016), it resulted in an autonomous control agent capable of driving a real-sized car in a real urban environment using only camera images and current steering angle as inputs for the neural network. Similarly, (Huval et al., 2015), focused on developing a real-time system to measure and evaluate the performance of supervised learning neural networks in detecting lanes and surrounding vehicles in a highway environment. The neural networks were also trained using data recorded from the behaviour of human drivers. It became clear that using real human behaviour as training data was a tedious process that consumes a lot of time and resources. Alternatively, simulation environments could save resources by generating artificial data instantly. A project named (CAD)2RL managed to train a quadrotor to avoid collisions using a discrete RL-algorithm called Q-learning. Surprisingly, the model trained on a totally simulated environment was successfully deployed on a real quadcopter and achieved good results without any fine tuning of the model parameters (Sadeghi & Levine, 2016).

A current reinforcement learning research is hugely inspired by DeepMind's (Mnih et al., 2013) achievements in the ATARI environment using the Deep Q Network (DQN) algorithm which outperformed both humans and handcrafted controllers using only the game images as an input to a system of convolutional neural networks alongside a simple reward function. Using RL-based agents to control autonomous vehicles in a simulated environment is an active area of research. One of the most popular simulation environments known for such purpose is TORCS (Loiacono et al., 2016), which is a fairly popular racing simulation platform for deploying AI driven autonomous vehicles control agents. An early example for such an agent is Monte Carlo tree search (Fischer et al., 2015), which used forward motion model and attempted maximizing the designed objective function by performing random exploration of the action space.

Another deployed agent was based on evolutionary algorithms (Koutnik et al., 2013).This agent added images to the state-space after refining them using Fourier transform. In a different approach, (Loiacono et al., 2010), used a discrete Q-learning agent based on high-level navigation inputs to control the vehicle. A more recent approach using continuous state-space algorithms was implemented by (Kardell & Kuoscu, 2017). It successfully developed a steering control agent using two different deep reinforcement learning continuous algorithms, the first one is Deep Deterministic Policy Gradient (DDPG) and the second one is Actor Critic

with Experience Replay (ACER). Training data consisted of image data concatenated with internal states of the vehicle, i.e., current velocity, acceleration, and jerk.

Previous work done on ROS In recent years, ROS has been used extensively in creating, training and testing autonomous vehicles. Academic Research and industrial sector are porting their applications on ROS for the all advantages and flexibility it offers. In the coming, we present few examples of autonomous cars on ROS. (ros.org, 2018b)

Marvin is an autonomous car from the Department of Computer Science at The University of Texas at Austin. It has competed in DARPA urban challenge in 2007.It is equipped with Velodyne HDL LIDAR and a Position and Orientation System for Land Vehicles. Drivers for these are available as open source packages on ROS.Also Marvin team have created a library for navigation system on ROS .Currently Marvin is used in a multiagent research where a framework for handling intersection between autonomous vehicles safely and efficiently is being investigated.

Stanford Racing team has built an autonomous car called Junior to participate in DARPA challenge, they first used Player as framework but they started using ROS-based perception libraries in Junior's obstacle classification system. They made use of ROS point Cloud library for classification.

Some developers are using ROS framework to build racing cars and invite other players to make their car and participate in the competition .For example Formula PI (Selby, 2018) is a lane tracking car built on ROS and uses Gazebo as simulation environment. The car model is made through modifying an existing open source model called "Husky" (ros.org, 2018a) which is an outdoor ready unmanned ground vehicle used for research and rapid prototyping. It comes with packages for path planning and is equipped with a small camera and LIDAR. The track is designed on Sketch up and spawned in the simulation environment as .dae format. Formula Pi uses Open CV for lane detection.

BMW migrated to ROS when they realized that it is extensively used in research and when they discovered that the framework they were using- EB Assist ADT - had reached its limit.(Paepcke, 2016)

In his talk in Roscon 2015, a representative from BMW justified their choice for ROS:

Autonomous driving benefits from robotics research and ROS have been become very popular in the robotics community, Stability and reliability from a very large user base, Quick tests and integration of already available algorithms and software packages saves development time, Open source ,Easier cooperation with universities and other research institutes.

At BMW they depend on ROS for recording laser scanner readings, object and lane markings detection, localization, trajectory planning and automated driving function.

Related work in Lane detection Lane detection systems had been investigated for 20 years (Gurghian et al., 2016). The most common approach was to use computer vision techniques to predict the model of the lane. Popular models are splines (Aly, 2008), clothoids (Gackstatter et al., 2010) or cubic polynomials (Loose et al., 2009) . Researchers used Particle or Kalman filters to track the lanes fusing temporal information from the image and the visual odometry (Teng et al., 2010).

These models have several disadvantages: First they can't be easily scalable because of road scene variation, Second, Position estimation of the lanes depend on accurate segmentation which is computationally expensive. In recent years, deep learning methods have been used for lane detection and lane keeping assistance. The usage of deep neural network eliminates the need for segmentation and hand crafted features as it implicitly learns the relevant features of the task at hand . In the coming we give a brief view about convolutional neural network and show related work done with CNN.

CNN was commercially used 20 years ago, but it is not until recently that it was used extensively in research field and in industry .This has two main reasons. First, large, labeled data are now widely available for training and validation. Second, CNN learning algorithms are now implemented on massively parallel graphics processing units (GPUs), tremendously accelerating learning and inference ability. (Bojarski et al., 2016)

CNNs are used in the field of self-driving cars in two ways The first way is followed by the majority of researchers; they investigate the performance of a deep neural network in the recognition ,detection or classification of specific visual features of the road such as Lane marking, road signs ,pedestrians, cars,.. etc. (Redmon et al., 2015) **The second way** is to use an end to end deep neural network that is trained to autonomously drive a car without dividing the task into features detection, depending in this case on the network to learn the relevant features without interference. The later approach is found to be more optimum and less coasty; however it remains complicated and difficult to understand its learning process.

In the following we review some work done both in **lane detection** and **in end to end driving using Neural Networks**.

(Gurghian et al., 2016) implemented deep neural network to estimate the lane position and then calculate its orientation. Images are generated from two laterally mounted down facing cameras with an update rate of 100 frames/s. This camera configuration helps providing clear and uncluttered view of the lane markings.

The paper formulates the task of estimating the lane position as a classification problem using the Network architecture shown in figure 2. For a given image Xi, the deep neural network computes a softmax probability output vector $Y_i = (y_0, ..., y_3 16)$. Entry y_k in Y_i corresponds to the probability that row k in image Xi contains the position of the lane marking. For inferring the position of the lane marking, the input image is fed through the network and the estimated position of the lane marking e_i for image X_i is assumed to be in the image row corresponding to the entry in Y_i with the highest probability:

$$e_i = \underset{0<i<316}{\text{argmax}}\ y_i$$

The training dataset consisted of 80000 real world images plus additional synthesized 40000 images. The network was trained using the open source deep learning framework Caffe on a Nvidia Digits DevBox.

The evaluation shows that E2 = 97.85% of the estimated lane positions deviate 2 or less rows from the ground truth and E5 = 99.04% deviate 5 or less image rows

The softmax loss function reaches a value of 0.016 after 60 epochs.

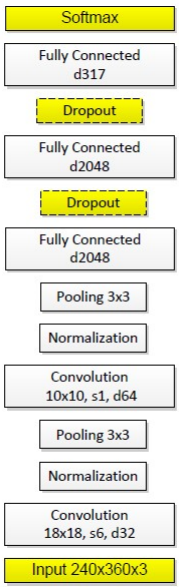


Figure 2: Network architecture for lane detection

Other researchers went beyond lane detection and used CNN for an end to end learning for self-driving cars. This work dates back to 1989 when Pomerleau first built the Autonomous Land Vehicle in a Neural Network (ALVINN) system. In his paper, Pomerleau provided the initial proof of a concept that an end to end neural network can be trained to provide the steering angle for an autonomous car. Pomerleau used artificial images for training, managed to drive a 400-meter path through a wooded area under sunny conditions with a speed of 0.5m/s indicating that neural

network could be a potential solution to autonomous driving.(Pomerleau, 1989)

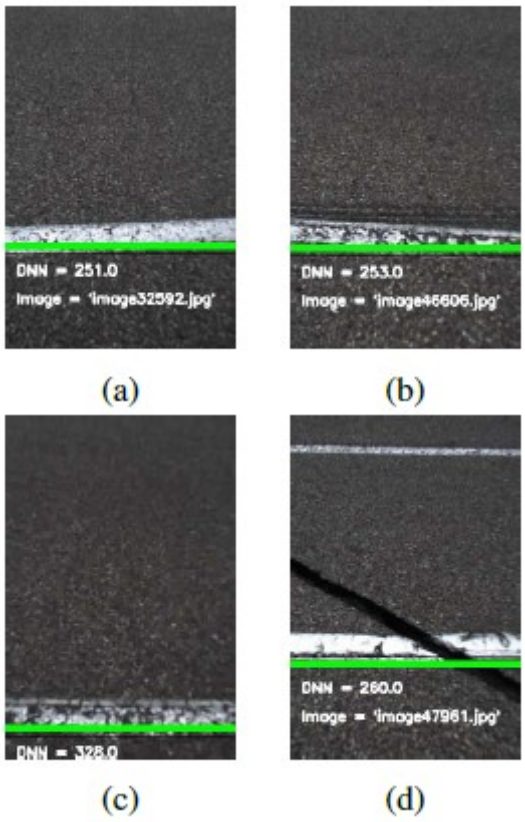


Figure 3: This figure shows examples images with a deviation from the ground truth larger than 5

Then in 2005,the work of Pomerleau has inspired DARPA to start its autonomous vehicle project known as DAVE (LeCun et al., 2005) in which a sub-scale radio control (RC) car drove through a junk-filled alley way. The training data included video from two cameras and the steering commands sent by a human operator. But DAVE's performance was not sufficiently reliable to provide a full alternative to the more modular approaches to off-road driving. For example the mean distance between crashes was about 20 meters in complex environments.

Further in 2016, researchers at Nvidia started a new effort to improve on the original DAVE, and create a robust system for driving on public roads DAVE–2 (Bojarski et al., 2016) . The primary motivation for this work is to avoid the need to recognize specific human-designated features, such as lane markings, guard rails, or other cars, and to avoid having to create a collection of "if, then, else" rules, based on observation of these features.

Dave-2 collects time stamped videos as training data from 3 cameras mounted on the data acquisition car along with the steering command from the driver through the vehicle's Controller Area Network (CAN) bus, steering command is presented as 1/r to avoid singularities. For lane keeping the video is sampled at 10 FPS.Data are

then augmented to represent the recovery from mistakes .The network is trained to reduce the mean square error between the output and the driver steering command. Figure 3 shows the network architecture, which consists of 9 layers, including a normalization layer, 5 convolutional layers, and 3 fully connected layers. The input image is split into YUV planes and passed to the network.

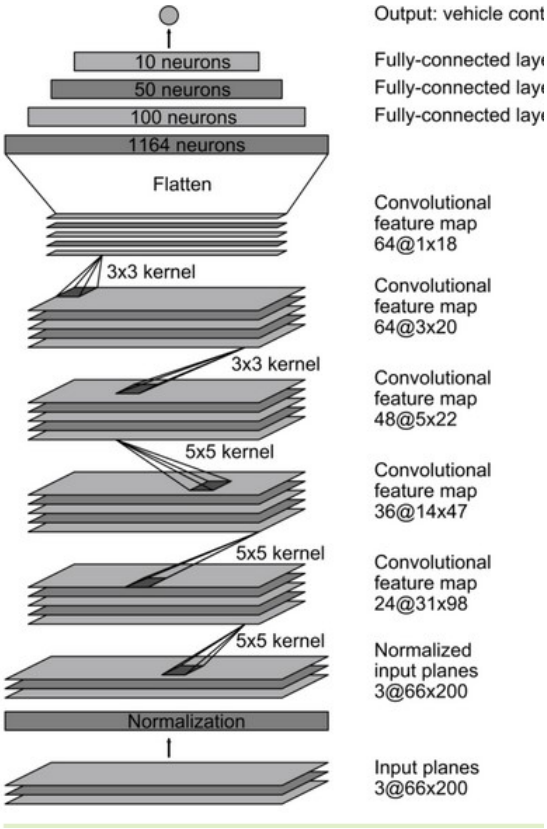


Figure 4: The network has about 27 million connections and 250 thousand

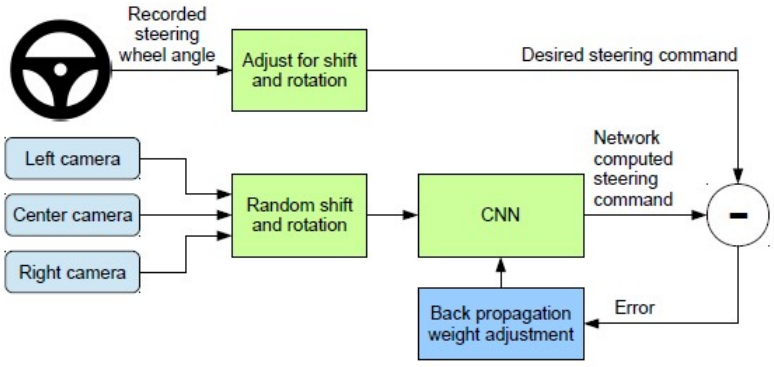


Figure 5: Training the neural network.

The network performance is first evaluated on a simulator and then on-road. A performance metric used for the evaluation which is the autonomy: the percentage

of time the car drove itself without the intervention of a driver to recover from a mistake, it's assumed that in real life an actual intervention would require a total of six seconds, the autonomy is

$$autonomy = (1 - \frac{\#\,of\,interventions * 6[seconds]}{elapsed\,time[seconds]}$$

Result	Simulation	On Road
autonomy	90%	98%

The paper has shown that learning the entire task of self-driving without decomposing the task road or lane marking detection, semantic abstraction, path planning, and controls, is possible through CNN. Still more work is needed to improve the robust-ness of the network and to verify it. One of the drawbacks of the previous researches the multitude of sensors which may be coasty.

Recently in 2017, Innocenti investigated whether robust driving behavior could be learned by means of imitation learning using only a single front view camera. He used and end-to end CNN with similar architecture to the one employed by NVIDIA. The evaluation was based on how well the vehicle is positioned in a lane and the smoothness of the driven trajectory. The learned policy succeeded in positioning the car approximately 99% of the time. While it lacks smoothness in actions as it does not consider previous states and take instantaneous decisions.(Innocenti et al., 2017)

3 Statement of the problems

3.1 Problem Formulation

3.1.1 Driver Control problem definition

Making simulation that is near to the reality as much as possible so that we could train the autonomous car on a simulated track then we could take the values of our model to apply on a real car. We will begin by making lane keeping then object avoidance and parking. We will try several algorithms and compare their results to each other and with the required output safe driving and reasonable speed.

We start by considering the most recent way of solving the problem. Instead of using supervised learning and unsupervised learning, we considered using Reinforcement learning which is a more natural way of learning in human life. In real life when an infant start learning a new skill he just tries a lot of things sensing the rewards that he receives. According to his environment he starts by preferring actions over other actions that will maximize his rewarding. Cause and effect provides a great knowledge of the environment that we are partially aware of. We seek to control it through our actions. RL is a goal-oriented methodology from interaction with the environment. RL is not kind of supervised learning which gets a categorized data from an external supervisor that gives examples of several states and the correct action to take in every state. The learning in this process is about generalizing from the agent from the examples it sees to those that are unknown. Also, RL is not kind of unsupervised learning which try to find hidden structure in training data and then classifying the coming data to those structures. RL is a new paradigm of machine learning that intersects with supervised learning and unsupervised as shown in figure 6 . RL is concerned with mapping situations into actions. RL agent isn't taught what to do as other machine learning algorithms do. Instead it explores different actions and calculates the cumulative reward of its action. It is greedy about taking actions. Here comes the problem of exploration and exploitation. There are different algorithms that try to solve this problem. When to exploit the best actions and doing them always and when to explore unknown actions that may give it more reward. RL is a closed loop process meaning that actions taken in the past time affect the future. A complete description of the RL process is through MDP Markov decision process.

Markov decision process MDP captures the most important features of the environment through sensation. Forming some states. Then actions are taken by the agent to reach a goal which is maximizing the reward on traversing those states. in MDP, environment's dynamics could be described by specifying only the probability of transitioning to S_{t+1} giving current s and a

$$P(s', r|s, a) = P(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$$

The state could be the location of the agent, it's velocity or the curvature of the road.

Solving Reinforcement learning problem There are **four** main elements in RL besides the agent and the environment they are policy, value function, reward signal, and optionally a model of the environment.

1. policy

   Policy π is the agent's behavior that is learnt from interaction with the environment. It is the mapping from the observed states to actions required to be done. It is the same as stimulus and response in psychology. Policy could be stochastic or deterministic. In some games such as back-gammon the optimal behavior is to be stochastic and not playing the same way every time. In back gammon game if the RL agent used deterministic policy the opponent will win every time as he will play the counter plan. However, in stochastic policy actions are given some probability to be played. Policy is the main core of RL. If the agent has the optimal policy π^*. It could behave in the best way according to the coming states. policy is the probability of acting in a certain state.

   $$\pi(a|s) = P(A_t = a|S_t = s)$$

   Policy could be simple as searching in look-up tables as in discrete case with small number of states. It can also be complex and requires computation in case of large number of states that cannot be stored in a look-up table such as continuous actions and states in autonomous car.

2. Reward Signal

   The second component is the reward signal. Reward signal is what defines the goal of RL. The agent main objective is to maximize the cumulative discounted reward following some policy π

   $$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

   γ is the discount factor to make the future rewards less important and make the return finite value in case of continuous non-episodic states that doesn't have a terminal state. Thus, the reward signal defines good and bad actions. Rewards comes immediately after taking an action and they define the problem faced by the agent. Agent can change the coming reward in two ways. Firstly, directly by changing the action it takes. Secondly, indirectly by changing the environment's state by transitioning to another state. However, the function that generates the reward function cannot be changed i.e. it cannot change the problem that it is facing. Rewards modify the policy on the long term. If the followed policy gives low cumulative reward, it will be updated in the agent.

3. Value function

The third component is the value function of the environments' states. Whereas reward specifies what is good immediately, value function indicates the good actions on the long run. Value function gives an indication of how desirable a state is. For example, a state may always give low reward, but it still has a high value function as it followed by states that give a high reward. So, the value functions makes the agent far-sighted rather than being myopic from the immediate rewards. Value function is important ,however, they are secondary and cannot be formed without the rewards. The only need for calculating value function is to achieve more rewards in the future. So that actions are taken to seek the highest value states not the highest rewards. Because these actions are the ones that obtain the highest rewards in the long run. These values are formed after experimenting different actions and rewards. The value function is defined using the bellman expectation equation as the following

$$\begin{aligned} v_\pi(s) &= E_\pi[G_t|S_t = s] \\ &= E_\pi[R(s,a) + \gamma V_\pi(s_{t+1})|s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned}$$

There are different algorithms that tackles how to compute the value function. Estimating states' Value is more complex than rewards as they require many observations to be made for estimating them and re- estimating them again until the optimal values are found. Another needed definition is value of acting. It is also defined recursively using the next state action pair.

$$\begin{aligned} Q_\pi(s,a) &= E_\pi[G_t|S_t = s, A_t = a] \\ &= E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \end{aligned}$$

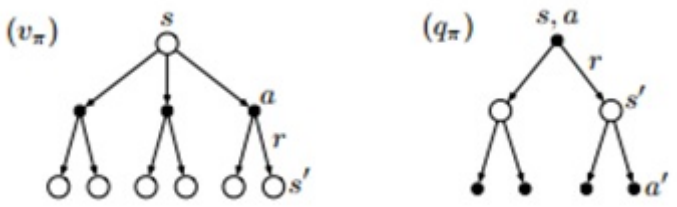


Figure 6: Backup diagrams for v_{pi} and q_π

The optimal bellman equation is by taking the max operator over the next

state action pair in Q function and max next state in v_{pi}

$$v^*(s) = max_a\, q^*(s, a)$$

Optimal policy is selecting the actions that maximizes $q^*(s, a)$

$$\pi^*(a|s) = a = argmax\, q^*(s, a)$$

4. Model

   The fourth element of RL is the model. Models allows the agent to make inferences about the environment. Given a state and an action the model could expect the resultant next states rewards. Learning by models are kind of planning. Models decide which actions should be taken before they are experienced. So, methods using models are called model-based methods in contrast with other methods that are trial-and-error learners. There also methods that make the use of both planning and trial-and-error. So, there is a whole spectrum between the two methods.

3.1.2 Vehicle simulation problem definition

We cannot depend on a physical car to train the RL algorithm as it will end up with destroying the car. As a result, we need a simulation environment to work on it. To get a complete simulation environment, 4 things are needed:

1. The car must be simulated with every detail such as the mechanical structure of the car, the joints of the wheels, the weight distributions, the inertia calculations, the position of the motor and how to translate the force to the wheels and the steering mechanism.

2. Sensors and cameras must be simulated in order to get data from the surrounding simulated world and train our algorithms based on that data. These sensors could be laser sensor, camera or ultrasonic sensor. Besides, the specifications of the sensors and the camera should be configurable to change it if the algorithm need to.

3. A real environment must be simulated. This environment will be the track that the car will move through with a capability to complicate it to look like the physical world that the physical car experiences.

4. The simulation should be able to detect collisions between objects in case the car takes a bad decision. In the physical world, wrong decision in driving could result in not only collisions but also flipping the car over. If the simulated car flipped over, the simulator should be able to define that the simulation stuck and need to be restarted without removing or resetting the parameters that the machine learning algorithm has learnt.

3.2 Possibilities and Limitations

3.2.1 RL Algorithms

The main limitation in a complex control problem is usually constructing a fairly accurate model of the environment the agent is interacting with. If there exists such a model, it is practically feasible to solve the problem using Dynamic Programming methods. Unfortunately, it is currently impossible to generate such an accurate model for all the possible variables and factors that are affecting the control decision of an autonomous vehicle through affecting the surrounding environment whether this factor was the density of the traffic or a rushing car coming through an intersection. That is where reinforcement learning is proven to be useful as it has the capability of model-free learning, which means it can learn from scratch and without knowing anything about the environment initially, through interaction with the real or simulated environment then generalizing to other unseen comparable scenarios.

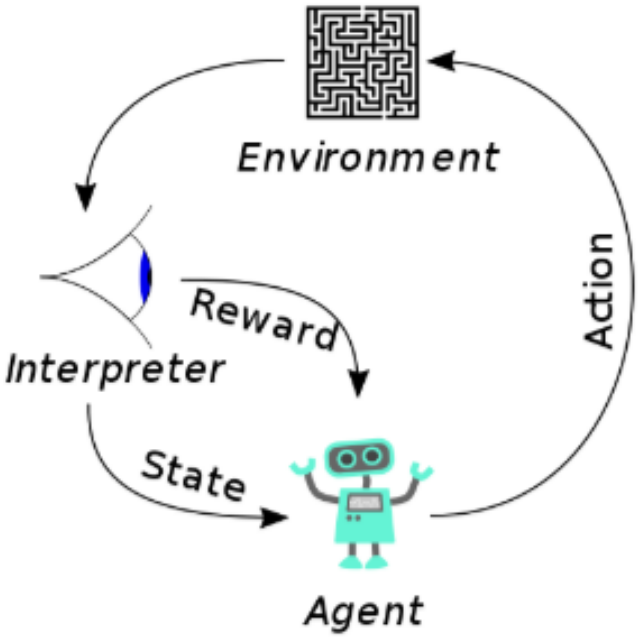


Figure 7: RL Scenario

Getting into the inner workings of reinforcement learning, it is clear that there is a lot of aspects to be considered and a lot of crossroad decisions to be taken. Comparisons between different methods of handling each crossroad is to be made with each method having its own set of advantages and limitations. The main ones are discussed as follows:

Discrete vs Continuous A discrete RL agent is used to control models with discrete state-action pairs. The agent mainly creates a table of all combinations of states and actions with every cell of the table contains a value representative of how good to be in this state-action pair. The most used metric is the Action-value function Q(s,a) of this specific state-action pair. However, the autonomous vehicle environment is continuous by nature as there is an infinite set of states and actions, so to use a discrete RL algorithm such as Q-learning or SARSA, we have to carefully find a suitable discretization for states and actions. Discrete RL algorithms are relatively simple compared to continuous ones yet they are still functional. However, Discretization is known to cause a varying degree of insensitivity and lack of robustness as it limits the scope of agent's ability to acquire data from the environment. In such cases, a continuous RL agent is preferable to control models with infinite state-action pairs. Unfortunately for the continuous algorithm's policy to be able to process infinite states and actions to produce continuous action output, it has to use a function approximator. A function approximator is a function model that has the ability to generalize from the finite set of observed states/actions during the inevitably finite time of learning to be able to predict for the infinite possible unseen states/actions using approximation. Function approximators can be as simple as just using a linear combination of features. Nevertheless, Relatively complex function approximators such as decision trees and neural networks are generally used in literature (Kardell & Kuoscu, 2017).

Exploration vs Exploitation Another crucial concept to be considered during agent design is the dilemma of exploitation vs exploration, where exploitation means to make the agent always execute the best-known action with the highest action-value function according to the given information. On the other hand, exploration allows drifting away from the optimal action in order to investigate the output of different actions with unknown or partially-determined action-value function. A fully exploitative policy is prone to getting stuck at a local optimum and is highly unlikely to converge to the optimal behavior, whereas a fully explorative one will hardly witness any improvement as it has no specific tendency to repeat the highly profitable action and can spend significant amount of time exploring unnecessary action-state pairs. A successful agent should combine both attributes in a certain formula to get as nearly as possible to the optimal behavior.

On-Policy vs Off policy Two learning strategies can be followed during learning phase. On-policy learning means that the agent is trained by directly adjusting its policy using experience sampled by following the same policy. In other words, the same policy is being followed and updated concurrently. Alternatively, Off-policy learning means that the agent's policy is updated using experience sampled by following a different policy. Off-policy learning enables the agent to learn from previously saved behavior such as old policies or even human-driven behavior. It can also be used to learn about optimal policy while following exploratory policy, which can be used to solve the exploration vs exploitation dilemma.

Value-based vs Policy-based Learning phase of RL agents requires iterating over a set of functions while updating a set of variables until convergence. In continuous RL algorithms, the update is mostly done using gradient descent/ascent. The function being updated iteratively can be value-based or policy-based. In a value-based algorithm, a parameterized value -or "action-value"- function is updated iteratively to minimize the temporal-difference errors. The next action to be taken by the agent is chosen based on an implicit policy that is dependent on the current estimated value function (e.g. greedy or ϵ-greedy). In other words, the policy cannot exist without a value function. Value-based algorithms are simpler to construct and easier to debug as they are more intuitive and quantitative. However, they are less effective in high dimensional action spaces. Alternatively, Policy-based algorithms learn a parametrized policy directly without the need for a value function to form the policy and take the next action. Updates and learning are applied directly to the policy using the gradient of the expected reward w.r.t. the parameters of the policy. They have better convergence properties, but they often get stuck at a local optimum. Some other advantages include the ability to learn stochastic policies and their effectiveness in continuous action spaces.

3.2.2 TORCS simulation

One of the famous simulation environment which is associated with artificial intelligence researches is The Open Racing Car Simulators software, TORCS. TORCS started as an open source race simulation game, on Linux and other platforms, which includes 3D environment and race or road tracks in addition to controllable simulated car (Anderson, 2000). TORCS becomes known as research simulation environment for artificial intelligence and autonomous racing car development, however, there are online competition to create AI bots to win specific race on the game as racing championship 2017 (Bernhard, 2016).

Daniele Loiacono, Luigi Cardamone and Pier Luca Lanzi have published a brief explanation to use TORCS in AI research and application in their manual for Simulated Car Racing Championship Competition software, international competition in 2013. This manual is still the main source of information to handle any following modified edition of TORCS or to understand the main architecture and possibilities of the software. Furthermore, the manual provides how to install the software and a full description of the sensors and actuators in the software environment. (Loiacono et al., 2016)

The competition software of TORCS is used on AI algorithms comparison instead of original TORCS due to the drawbacks of the original TORCS architecture: the bots are compiled and loaded in the main memory. As a result, bots can block race real-time execution if it takes longtime to take an action. On the other hand, the bots and simulation engine are attached, and the bot can use a lot of information to drive the simulated car. This is not providing a reasonable judgment on the algorithms of the controller because they use different inputs or actions and have

Figure 8: TORCS Simulation

unrealistic information as a driver. In addition, the original TORCS architecture provides writing the bots in C/C++ only. (Loiacono et al., 2016)

The new Competition Software architecture, which is widely used now, converts TORCS from standalone application into client-server one. The bots, which takes the action on AI algorithms, are another process off the race process connected as client to server respectively through UDP connection. The architecture solves the real-time problem by the following: the serve sends sensors readings and current states to the bot client and wait for action from the clients for 10ms in real time otherwise it performs the last action until it receives the new one. Adding a separation between control code and race simulation provides availability of using any programming language to code and choosing of simulation information to provide the AI code. It also provides the client with a restart request to be send to server at any time. (Loiacono et al., 2016)

Creating a bot is not requiring any awareness with TORCS internal structure. The inputs expected by the AI driver are categorized into car status, race status, the car surrounding and realistic actions. All these are derived from sensors and actuators information. ***Sensors*** support the bot with the race and car information in addition to environment related as follows:

1. Angle of the car with the tack axis
2. Current lap time
3. Damage of the car
4. Distance from the beginning of the race

5. Distance from the start line
6. 5 Range finder sensors reading: the distance between the track edge and car and return only -1 outside the track
7. Fuel level
8. Gear number
9. Time last lap completed
10. 36 opponent sensors: distance between the car and the closest opponent car
11. Race position: position of the car in Race
12. Rpm of the car engine
13. Speed in (x, y, z) directions.
14. 19 range finder sensors: the distance between the track edge and car and return only -1 outside the track
15. 4 sensors of the rotation speed of the wheel
16. The distance of car mass center from the track surface

Actuators value will be got from the bot client to the race server to simulate the driver actions on the race. *Actions* as follows:

1. Binary decision for virtual gas pedal
2. Binary decision for virtual brake pedal
3. Binary decision for virtual clutch pedal
4. Gear value
5. Steering angle

There is also a meta action for restarting the race.

Selecting sensors to get information from depends on the driver control design. TORCS makes it also easy to disable a lot of feature in race environment such as fuel consumption, damage, lap time and timeout of the server to act can be specified in nanoseconds. TORCS provides the range finder sensors as not noisy (default) or noisy if specified. Noisy sensors are preferred in real simulation because sensors are not accurate in real life and get affected by a lot of noise in their system which affect its accuracy.(Loiacono et al., 2016)

3.2.3 ROS simulation

What is ROS? ROS is an open source operating system for robots .It offers many tools and libraries to help developers implement their algorithms and deploy their work to a wide variety of robot platform. ROS allows running processes to be loosely coupled with each other through its communication infrastructure. Communication between processes could be synchronous over services, asynchronous streaming of data over topics, or storage of data on a Parameter Server. (wiki.ros.org, 2018)

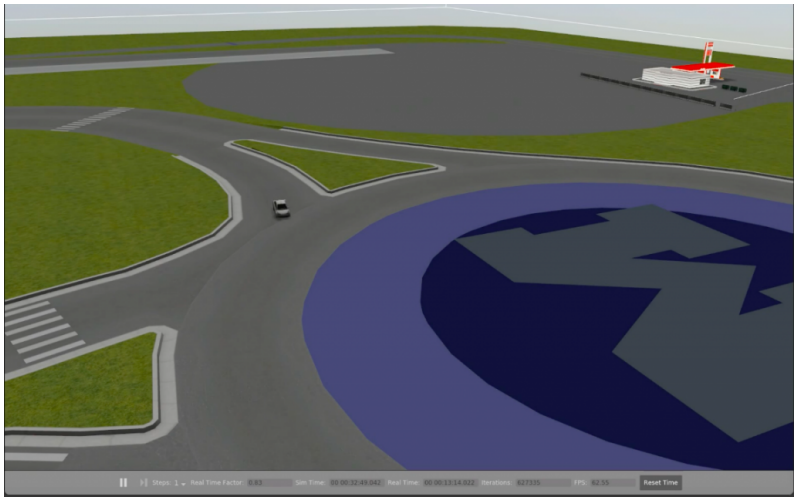


Figure 9: Example of Simulation of Autonomous vehicle on ROS by Open Robotics

Alternatives for ROS Other software frameworks are Player, YARP, Orocos, CARMEN, Orca, MOOS, and Microsoft Robotics Studio.

Why ROS as framework

- ROS is a flexible and powerful tool that supports modular design of the project. Modularity allows a distributed computation based on message delivery.
- ROS is an open source framework with good and clear documentation available, big community and large amount of project.
- Language independence: the ROS framework is easy to implement in any modern programming language. Coding is written in C++ or python.
- Over 3,000 packages are available in the ROS ecosystem, most of which are user contributed packages.(ros.org, 2018c)
- ROS gives flexibility to create your own hardware implementation by either modifying existing URDF files or creating your own file description.

ROS has many disadvantages that affect our process in building a simulated car such as:

1. ROS is very primitive in drawing objects which means we will not be able to draw the car as it is and we will try to come with a model that looks like the real one to simulate the weight distribution of it.
2. ROS is very sensitive to the weight values, inertia values and the joint position. Any change in this values may cause defections in the simulation. That's why these parameters should be from an actual car.
3. ROS has no model for the steering mechanism and requires it to be coded in order to result in a simulation of the movement of the physical car.
4. ROS and Gazebo are not able to detect that the simulation should be restarted if the car is flipped or crashed. That's why a plugin should be written to work aside with the machine learning algorithm.

3.3 Suggested Options for Solution

3.3.1 TORCS as a simulation option

The main problem which is formulated and discussed here is driving a car on a road, keeping its lane and avoiding obstacles or other cars using a car simulator to train an AI algorithm such as Reinforcement Learning Algorithms. To work on TORCS with RL algorithms, it should be formulated on Marcov Decision Process (MDP) and define its components (states, actions and rewards in TORCS's environment and agent.

In the Reinforcement Learning solution, model-free algorithms are used in the agent instead of using TORCS's transition model which is complex and include many transitions resulting in infeasibility in implementation (Karavolos, 2013). In the model-free algorithms, the agent cannot predict the next state and reward before taking the action after training. However, there are RL algorithms which are model-free such as Q-learning which estimates the optimal Q-values of each action so the policy will be derived from the action with highest Q-values in the current state. Mainly, we should determine the Marcov Decision Process (MDP) components in TORCS which RL agent will work on. The main MDP components in a model-free MDP algorithm are states, actions, Reward. The states are defined from the sensor reading and observation mentioned in the previous section. In spite, the whole 16 observations or sensors are not needed in the lane keeping problem or object avoidance because they contain non-important information. In addition, some sensors information is normalized such as angles from $[-\pi, \pi]$ to $[-1, 1]$. The sensors will be used in noisy mode to estimate reality and exploration feature. The fuel usage, car damage and lap time termination features in TORCS race is

turned off. However, we ca use damage observation to detect collisions. This leaves the following observations to be used in Q-learning or Deep Deterministic Policy Gradient (DDPG) algorithms, etc.

1. Distance from track sides (19 sensors), its dimension equal 19
2. Speed in (x, y, z) directions, Dim=3
3. Angle of the car, Dim=1
4. Rpm of the engine, Dim=1
5. Closest opponent distance, Dim=1
6. The 4 wheels spin Velocity, Dim=4

Theses dimensions are continuous. In Q-learning, discretization is a must but in policy gradient algorithms such as DDPG, function estimator is used for dealing with infinite states (continuous dimensions).

The actions in TORCS are Gas pedal, Brake pedal, Gear value and steering angle. Gas pedal represents acceleration, brake represents negative acceleration, gear values will not be added to automatically estimated action by RL policy, but it will be automatically changed from speed (x-direction) observation, while steering is the angle of the car with the longitude axis of the track. This abstracts the actions of the RL agent into 3 actions as MDP actions. In other words, the dimension of the actions is 3 and the dimensions of the states is 29.

Reward is the third component of MDP problem. In this problem, the reward depends on the speed of the car. Leaving the track and collisions are bad behavior, ending the episode by termination conditions is also bad behavior. Thus, the optimal policy should never reach bad events. The termination conditions include termination of the episode if the car progress or velocity is very small which is a waste of training time, processing power and memory on useless information. Another termination is if agent (car) runs backward or if the car gets stuck after many actions in the same large action. Although taking an action every 20ms in TORCS seems too rapid to realize the effect of different action in certain states, increasing the time to 200ms proved that it is bad at high speed for the agent to respond to new states so 20ms is fine as period between successive actions.

3.3.2 ROS as a simulation option

We started to search for another simulation programs. There are some programs that are dedicated to self-driving car projects. Some programs that are dedicated to car simulation are not real time simulators and their codes are not flexible to be changed. Instead of looking for a complete car simulator, we search for a simulation

environment that can serve in our target such as ROS. ROS (robot operating system) is a simulation environment that is dedicated for robot simulation and came up with a real time solution for the simulation.

Car Simulation

First, we need a complete simulation of a car that includes the weight distribution, the steering mechanism and the joints of the wheels. To come up with such a simulation, we need the complete information of a working car and import them into ROS which is hard process as we need the calculations of the inertia of every part of the car. Besides, ROS is very primitive in drawing objects which means we will not be able to draw the car as it is and we will try to come with a model that looks like the real one to simulate the weight distribution of it. A good advantage of ROS is that it is an open source program and feasible to import others works. After searching, many companies and projects adopted the project of simulating a physical car on ROS.

The first solution to the car simulation problem was the car that is offered by Cyber-Physical Systems Virtual Organization. It has a project of simulating a physical car on ROS for modern control purposes. It simulated a full sized Ford Escape that is available to be controlled using ROS messages. The team documented how to to install the simulation and interact with it in ROS. The simulated car can move using two parameters, the speed that it should move with and the angle that the car should turn around with. (Sprinkle, 2017)

Unfortunately, the design has deficiencies. When the speed of the car goes above a specific speed, the car starts to accelerate and decelerate. After debugging the problem, the way the steering mechanism was simulated in ROS is not right that makes the front wheels of the car do not move with the same speed of the back wheels of it. As a result, if the back wheels tried to move the car with a specific speed, the front wheel do not move with that speed due to problems with the joints of the steering and makes it decelerate while the back wheels try to accelerate. To solve this problem, we should correct the positions of the joints.

Instead, an organization called Open Source Robotics Foundation has released a simulation of the Prius car on ROS. Their team has simulated the vehicle's throttle, brake, steering and transmission and made them controllable by ROS. Besides, the car speed can be increased without any problems unlike the previous one. The only problem with that simulation is that it was done on the latest version of ROS (which called kinetic) and the latest version of Gazebo (which is 8) and cannot be run on earlier versions. ROS has a useful feature. Everything in ROS project is separable independent and could be moved from one project to another if you know what to move and where to move it to. For the car simulation, the car files are divided into URDF file, meshes folder, gazebo files, CMake file and the plugins that work on the simulation. (Tully Foote, 2017)

1. URDF file is the description of the weight distribution, the position of every

part in the car, the position of the joints including the steering joints and the motors joints and the inertia calculations of every part in the car.

2. Meshes folder is for inserting complicated shapes into ROS.
3. Gazebo files is responsible for defining which part of the car is the motor, laser sensor, camera or any sensor. Besides, it gives the ability to define the aspects of any sensor or motor.
4. CMake file is the coordinator between all the sources of ROS and the project.
5. Plugins are the codes that run when the simulation start to facilitate the simulation and make it much closer to reality.

So we could separate the description of the car from the Prius project by getting the car information from the URDF file. Then, by inserting the new Prius model as a replacement for the the jeep car in the first project, we have a well functioning car on ROS that works on the earlier versions of ROS and Gazebo.

Sensors

After simulation the car in ROS, the car needs sensors to get information from the surrounding simulated environment and then take a decision based on it.The sensors chosen to get data from the surrounding environment are the laser sensor and the camera. For the Reinforcement Learning algorithms, laser sensor is used. To simulate a laser sensor in ROS and Gazebo, we need to define the geometric dimensions of the sensor and where it will be jointed. After that, we define the aspects of the sensor that we need in our project that includes :

1. The number of rays the laser sensor will send and receive in the horizontal plane.
2. The number of rays the laser sensor will send and receive in the vertical plane.
3. The minimum and maximum angle the sensor observe in the horizontal plane.
4. The minimum and maximum angle the sensor observe in the vertical plane.
5. The minimum and maximum range for the sensor to observe any object.
6. The resolution of the sensor readings.

For the Convolutional Neural Network algorithms, camera is used. To simulate the camera in Gazebo, the same procedure is followed as the laser sensor. The geometric dimensions are defined and where it will be jointed. Then, we define the aspects of the camera that includes :

1. The frame size of the image captured by the camera.

2. The format of image.
3. If there is a noise or not and what the type of the noise is

After simulating the sensors, they could be accessed by any code that works in ROS and these codes can be for learning phase or testing one.

Scenarios Simulation

After having a simulation of a car with sensors, surrounding environments need to be simulated. We could divide surrounding environments into scenarios depending on the problem the machine learning algorithm is solving such as lane keeping scenario and object avoidance scenario. Each scenario consist of two designs, the training design and the testing.

Unfortunately, ROS is very primitive in designing any environment. It is either to write a HTML description of the scenario or try to draw it using the designing tools of gazebo which is very primitive. On the other hand, ROS and Gazebo have the option of importing meshes that are designed using any external programs if these meshes have a certain extension (.dae files). The most common program for designing meshes is blender. Blender facilities drawing complicated meshes and save them as dae files for ROS and Gazebo.

Plugins

After having a simulation of a car with sensors, a question must be asked is how will the car move. To answer this question we need first to know how the motors are inserted into our simulated car. The simulation as described before is simplified so that it simulates the physical car mechanism without simulating each part in the physical car itself. For example, the axis of the wheels is too complicated in the physical car. We don't need to draw each part of it. On the other hand, we could simulate the mechanism itself. If we take this example and applied it on the steering mechanism. To simulate the steering mechanism, we don't have to simulate each part of the axis or the hinges and then put a single motor to work on a particular hinge and a single steering wheel to another particular one. We could come with a simplified simulation that has the steering properties and work as it.

In order to simplify the simulation, it is suggested that each wheel will be jointed to the chassis with a rotating joint. This joint is offered by ROS and the main property of it is the ability to rotate freely in one direction. After that, for each of the back wheels, a motor will be added. This mainly because motor simulation in ROS could be work on only one joint. Then, a steering joint must be added to each one of the front wheels. This simplifies the simulation of the physical car.

Unfortunately, this simplification causes a problem. We can classify the movement of the simulated car into two categories, moving in straight line and turning around. For the first one, the car takes an order to move with a specific speed and a zero angle of steering. Then, each motor of the back wheels moves with that speed. For

the steering joint of the front wheels, they will take a zero angle. The car moves properly and no problem happen. On the other hand, if the car takes an order to move with a specific speed and a specific angle of steering otherwise a zero angle. Here comes the problem. First, for the motors of the back wheels, they cannot move with the same speed that is equal to the speed that the car take an order to move with. This will make the car to flip. The wheel that in the direction of rotation must have a speed less than the other wheel. Second, for the steering joints of the front wheel, they cannot have the same angle that the car takes an order to move with. This doesn't happen in the physical car. But, each steering joint will take different angle to move with. The steering joint on the wheel that is in the direction of rotation must take an angle larger than the angle of the other joint.

One suggested solution to this problem is the Ackermann model. This model is a mechanical description of the car movement mechanism if there is a steering angle. Suppose we have a scenario like the described one in the following figure.

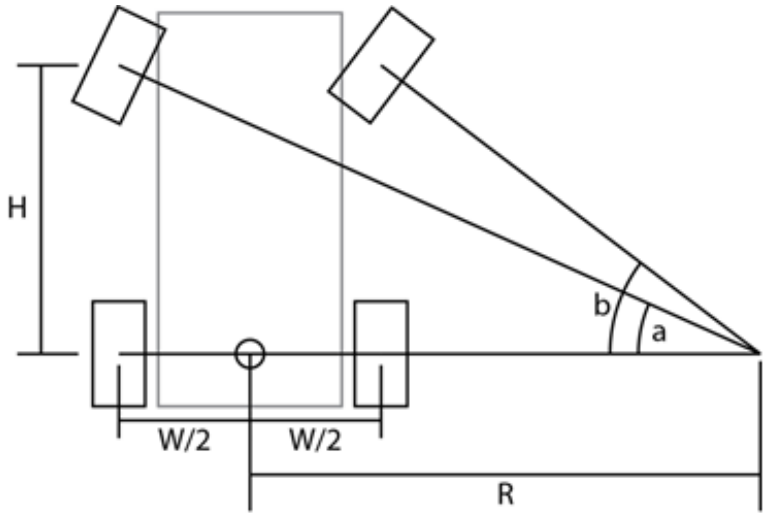


Figure 10: Suggested scenario for 4-wheels vehicle steering

θ $\rightarrow$ the angle the whole car should turn with
W $\rightarrow$ the distance between the back wheels
H $\rightarrow$ the distance between the back wheels and the front one
R $\rightarrow$ the distance between the center of the car and the point the car should moves around
b $\rightarrow$ the angle that the steering of the inner front wheel will turn with measured from the horizontal axis
a $\rightarrow$ the angle that the steering of the outer front wheel will turn with measured from the horizontal axis
ϕ_{inner} $\rightarrow$ the angle that the steering of the inner front wheel will turn with measured from the vertical axis
ϕ_{outer} $\rightarrow$ the angle that the steering of the outer front wheel will turn with measured from the vertical axis

Before starting to get a formula for ϕ_{inner} & ϕ_{outer}, we need to calculate R. R is calculated from the center of the vehicle that should turn with angle θ .

$$R = \frac{H}{\tan\theta}$$

Form trigonometry, we can calculate a and b.

$$a = tan^{-1}(\frac{H}{R + \frac{W}{2}})$$

$$a = tan^{-1}(\frac{H}{R - \frac{W}{2}})$$

To get the angle with respect to the vertical axis:

$$\phi_{outter} = \frac{\pi}{2} - \tan^{-1}(\frac{H}{R + \frac{W}{2}})$$

$$\phi_{inner} = \frac{\pi}{2} - \tan^{-1}(\frac{H}{R - \frac{W}{2}})$$

This solves the problem of the steering angle for each wheel. For the problem of the speed of each wheel of the back one, a ration has been defined.

$$ratio_{outer} = R + \frac{W}{2}$$

$$ratio_{inner} = R - \frac{W}{2}$$

Then,

$$speed_{outer} = required_{speed} * ratio_{outer}/R$$

$$speed_{inner} = required_{speed} * ratio_{inner}/R$$

These ratios increase the speed of the outer wheel and decrease the speed of the inner one which makes the simulated car turn around smoothly.

Simulation Enhancement

The machine learning process is an iterative process that need train on the data until reaching a satisfying results. The reinforcement learning algorithm will run on the simulation and will take time till it reaches a satisfying result which means that the car may be crashes or flipped over and the simulation need to be restarted again.

There is no implemented code in ROS that detect if the simulation need to be restarted. On the other hand, there is a sensor called contact sensor. This kind of sensors is responsible for advertising all the forces applied to the object that is connected to. If each wheel on the simulated car has this sensor and the car is moving without crashing then the contact sensor will have the forces coming from touching the ground. When the car flips over, the contact sensor will advertise no force on the wheels. That is the moment the simulation should restart to continue learning.

3.3.3 RL as an algorithm option

Solving MDP So far, the important elements of RL have been defined in 3.2.1. There are two parts of solving the MDP. Firstly, the prediction problem which is calculating the value function. Secondly the control problem which is finding the optimal policy π^*. There are two methods for solving the MDP using the model of the known the MDP. Those methods are based on Dynamic programming which are policy iteration and value iteration. Dynamic programming methods are not used as they have high computation cost and require full knowledge of the environment, however, they have a theoretical importance to build other methods which have less computation cost and are based on experience. Dynamic programming keeps making full back up of the states in the MDP several times till the optimal value is found.

Policy iteration to solve MDP this method consists of two parts policy evaluation during which the value of each state is calculated according to bellman equation and then policy improvement which update the policy. If the policy doesn't improve the algorithm stops

Pseudocode
initialize $V(s) \in R$ and $\pi(s) \in -A(s)$ arbitrarily for all $s \in S$
policy evaluation
  Repeat
    $\delta \leftarrow 0$
    For each $s \in S$:
      $V \leftarrow V(s)$
      $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma v_\pi(s')]$
      $\delta \leftarrow max(\delta, |V - V(s)|)$
  Until $\delta < \theta$ (small positive number)
policy improvement
  Policy-stable $\leftarrow true$
  For each $s \in S$:
    Old-action $\leftarrow \pi(s)$
    $\pi(s) \leftarrow argmax_a \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma v_\pi(s')]$
    If old-action $\neq \pi(s)$, then Policy-stable $\leftarrow false$
    If policy-stable, then stop and return $V = v^* \, and \, \pi = \pi^*$

Value iteration to solve MDP Value iteration converges faster than policy iteration. It makes in each sweep one policy evaluation by computing the value of $p(s', r|s, \pi(s))[r+\gamma v\pi(s')]$ of every new state from state s and then chooses the max of them to be the value function of the current states. figure 11 represent the back up for value iteration while figure 6(a) represents the backup for policy evaluation

Value iteration pseudo-code
initialize $V(s) \in R$ and $\pi(s) \in A(s)$ arbitrarily for all $sinS$
Repeat
  $\delta \leftarrow 0$
  For each $s \in S$:
  $V \leftarrow V(s)$
  $V(s) \leftarrow max_a \sum_(s', r)p(s', r|s, \pi(s))[r + \gamma v\pi(s')]$
  $\delta \leftarrow max(\delta, |V - V(s)|)$
Output is policy $\pi \approx \pi^*$ such that $\pi(s) = argmax_a \sum_(s', r)p(s', r|s, \pi(s))[r + \gamma v\pi(s')]$

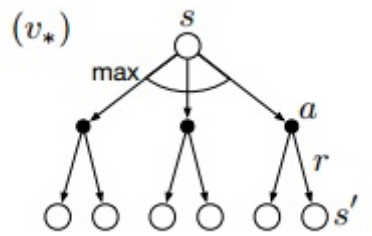


Figure 11: backup for value iteration

Model free Learning methods Model free methods don't require complete knowledge of the environment. They require only sampled experience of states, actions and rewards from actual experience or simulated experience. Monte Carlo learn from complete episode they don't bootstrap, i.e. they don't assume initial values for the states. The update step for all the states occurs only after the terminal state. MC uses the simplest idea which the value function is the mean return.

Therefore, to evaluate state s Increment total return $V(s) \leftarrow V(s) + G_t$

Value function is estimated by mean return $V(s) = V(s)/N$ Where N is the number of returns.

Whereas MC methods update the value of the states after the terminal state, TD(n) methods update each state after a fixed number n. So, for instance, TD(0) update each state after knowing the reward from the next state. the second state value is bootstrapped (given some initial value). so, the $V(s)$ is updated as

$$V(s) \leftarrow V(s) + \alpha[R + \gamma V(s^{'} - V(s)]$$

In both MC and TD(n) methods only the states that are explored have known values. Those methods need to be extended in continuous action and state function where there are many states and actions. Artificial neural network is the solution for this problem.

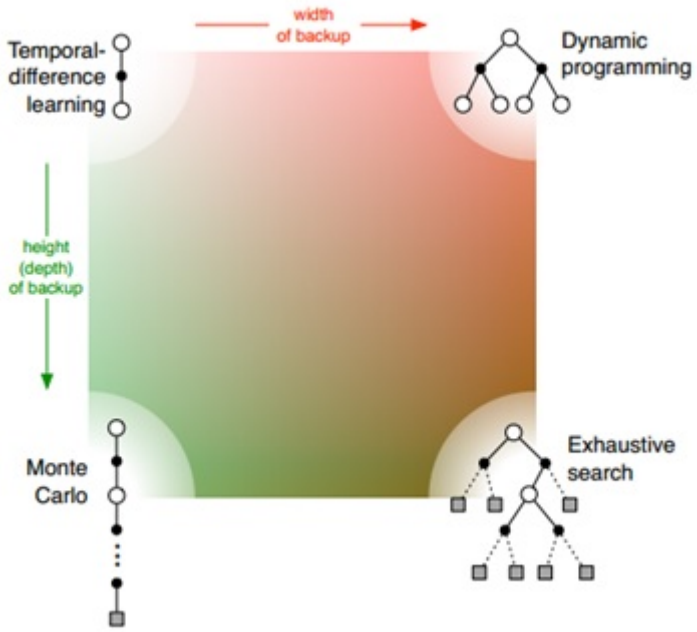


Figure 12: The space of reinforcement learning methods

Sarsa: On-Policy TD Control Now we use $TD(0)$ method for the control problem. This method learns the action-value function rather than the state-value

function.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

The update is made after every transition to a state that is non-terminal state. If next state is terminal then $Q(S, A)$ is defined as zero the update rule make use of five elements $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$ which are current state-action pair, the immediate reward and the next state-action pair. So this methods is named as SARSA update. The control algorithm based on SARSA is simple. It follows certain policy π then predicts the Q values of the state-action pairs. Then the policy is improved greedy based on the q values.

Sarsa: An on-policy TD control algorithm
Initialize $Q(s, a), \forall s \in S, a \in A(s)$, arbitrarily, and $Q(terminal - state, \bullet) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from $Q(e.g., \epsilon - greedy)$
  Repeat (for each step of episode):
    Take action A, observe $R, S^{`}$
    Choose $A^{`}$ from $S^{`}$ using policy derived from $Q(e.g., \epsilon - greedy)$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
    $S \leftarrow S^{`}; A \leftarrow A^{`};$
until S is terminal

Q learning Off-Policy TD Control Q-Learning updates the value of the state-action pairs based on another policy different from the policy followed by the agent. This method makes the convergence of the algorithm faster and simplified the analysis. The necessary condition to converge is that all pairs has sufficient updating. The update rule is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Q-learning: An off-policy TD control algorithm
Initialize $Q(s, a), \forall s \in S, a \in A(s), arbitrarily, and Q(terminal - state, \bullet) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from $Q(e.g., \epsilon - greedy)$
    Take action A, observe $R, S^{`}$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
    $S \leftarrow S^{`}$
until S is terminal

Policy gradient Policy gradients methods are parameterized policy that select action according to some parameters without checking their action values. those

parameters are adjusted by performance objective function they may depend on the value function, however value function is not needed for action selection.

$$\pi(a|s,\theta) = Pr[A_t = a|S_t = s, \theta_t = \theta]$$

It is the probability of taking action a in state's given the policy parameters θ at time t. policy weight are learned using the gradient of the objective function $\gamma(\theta)$ with respect to the policy weights.
The objective function is defined as

$$\eta(\theta) \doteq x_{\pi_\theta}(s_0) = \Sigma\pi(a|s)q_\pi(s,a)$$

Gradient ascent is used to maximize the update of the objective function

$$\theta_{t+1} = \theta_t + \alpha\nabla\gamma(\theta)$$

$$\nabla\gamma(\theta)\text{is the derivative of the performance function} \tag{1}$$

Policy gradient could be stochastic or deterministic. The stochastic gives the different actions some probability to be taken in every state . However, the deterministic function is $\pi(a|s,\theta) \in (0,1)\forall s, a, \theta$.

The probability of the actions are determined using the softmax function

$$\frac{\pi(a|s,\theta) \doteq exp(h(s,a,\theta))}{\Sigma exp(h(s,b,\theta))}$$

Where $h(s,b,\theta)$ is the preference of taking an action.

$$h(s,b,\theta) = \theta\phi(s,a)$$

The update of the θ parameter will be

$$\theta_{t+1} = \theta_t + \alpha\gamma^t G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$$

This update favors the actions which get the highest return. This return could be Monte Carlo return or Temporal difference return. This update is proportional to the return and inversely proportional with the probability of the action.

In policy gradients methods G_t is the true return value. So in policy gradient methods return value is computed while the action probabilities are approximated. There are other methods that approximate both the value functions and the action probability. Those methods are called actor-critic. The actor is the policy that takes action according to the parameter θ and get some return G_t. while the

critic criticizes this action through the approximated value function parameter w to evaluate how good the taken action is.

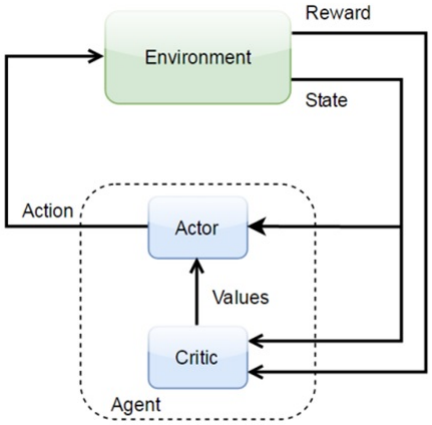


Figure 13: Actor Critic Algorithm abstract Illustration

Actor-critic algorithm pseudo-code
Input: differentiable policy approximator and state-value approximator $\pi(a|s,\theta)$ and $v(s,w)$ respectively
Parameters: step sizes $\alpha > 0$, $\beta > 0$
Repeat forever:
Initialize S (first state of episode)
$I \leftarrow 1$
While S is not terminal
A $\pi(a|s,\theta)$
Take action A, observe S' , R
$\delta \leftarrow R + \gamma v\ (S^{'}, w) – v\ (S, w)$ if($S^{'}$ is terminal , then $v^{'}(S\ , w) = 0$)
$w \leftarrow w + \beta\delta\nabla_{w} v\ (S^{'}, w)$
$\theta \leftarrow \theta + \alpha I \delta \nabla_{\theta} log \pi(A|S,\theta)$
$I \leftarrow \gamma I$
$S \leftarrow S^{'}$

Deep deterministic policy gradient (DDPG)

As the value function could be calculated by the Q value of the state-action pair as stated earlier , the objective function in DDPG is defined by the Q value of the actions and the deterministic policy $\mu_{\theta}(s)$

Therefore the gradient of the objective function will be

$$\nabla_{\theta} J(\pi_{\theta}) = \nabla_{\theta} \mu_{\theta}(s) \nabla_{a} Q^{\mu}(s,a)_{a=\mu_{\theta}(s)}$$

This gradient is using the true value for the Q value. when the Q value is approximated using some parameter w this method gets it's name deep policy gradient.

$$\nabla_{\theta} J(\pi_{\theta}) = \nabla_{\theta} \mu_{\theta}(s) \nabla_{a} Q^{w}(s,a)_{a=\mu_{\theta}(s)}$$

In DDPG the critic approximates the action value function and the actor ascends the gradient of the action value function .the actor tune the parameters θ of the policy $\mu_\theta(s)$.

Critic estimates the action – value function $Q^\mu(s,a) \rightarrow Q^w(s,a)$

The critic uses policy evaluation algorithm for example SARSA algorithm could be used to update the action value function. Similar to the previous pseudo code:

Updates and the target could be defined as

$$\delta_t = r_t + \gamma Q^w(s_{t+1}, a_{t+1}) - Q^w(s_t, a_t)$$

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w Q^w(s_t, a_t)$$

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \mu_\theta(s_t$$

$$\nabla_a Q^w(s_t, a_t)|a = \mu_\theta$$

3.3.4 CNN as an algorithm option

In order to compare between two different algorithms, convolutional neural network is chosen to compare its performance against the reinforcement algorithms. The convolutional neural network (CNN) is a non-linear algorithm that is based on image data-set and tries to relate it to the required performance. Unlike the reinforcement algorithms, CNN depend on the existence of the training data.

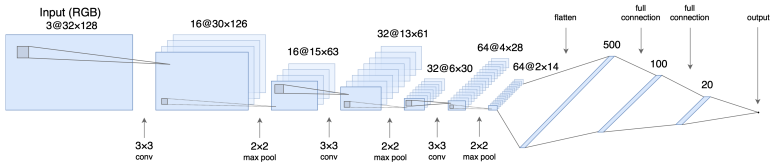


Figure 14: Example of CNN Model

1. Data-set: The data-set that the CNN algorithm works with is taken from the frontal camera that is jointed in the front of the simulated car. These data-set consists of two parameters, X parameter and Y parameter. The X parameter is the feature input to the CNN. The Y parameter is the labels for each corresponding X parameter. X parameter is the images that are taken with a fixed rate from the frontal camera. The frame size of these images is 200 in the width and 100 in the height. On the other hand, Y parameter expressed as the 2x1 dimensional array, the steering angle and the acceleration.

2. Model Architecture: The model architecture of the CNN is composed of 11 layers, 5 convolution layers, 1 flatten layer, 4 fully-connected neural network layers and the output layer. (Innocenti et al., 2017) For the convolution layers:

(a) 24 filters with 5x5 kernels applied with 2x2 strides.

(b) 36 filters with 5x5 kernels applied with 2x2 strides.

(c) 48 filters with 5x5 kernels applied with 2x2 strides.

(d) 64 filters with 3x3 kernels applied with 1x1 strides.

(e) 76 filters with 3x3 kernels applied with 1x1 strides

These five convolution layers use ELU activation function. After that, a flatten layer is used to reshape the results of the convolution layers into a vector representation of size 1216. Then, it is followed with 3 fully-connected layers.

1. Hidden layer of 100 neurons.
2. Hidden layer of 50 neurons.
3. Hidden layer of 10 neurons.

Then, the results of the hidden layers are passed to the output layer. This layer consist of two neuron and has no activation function as they act like an approximation to the required steering angle and acceleration.

4 Project Design

4.1 Tracks and Car modeling

The car design is made to be a simulation of the weight distribution of a real car. Then, simulated motors are added to the back wheels and steering joints are added to the front wheels. After that, the Ackermann model are applied to the motors and steering joints in order to give the simulated car the exact movement of the real car. As a result, the car is ready to move properly and needed to be prepared for the training phase. A simulated laser sensor is added to the front side of the car to get the data needed for reinforcement learning algorithm. Besides, a simulated camera is added on the top side of the car for the CNN algorithm. For more details, find section 3.3.2

For the track design, two tracks are needed, one for training and the other for testing. The main criteria for designing the two tracks are:

1. Changing the pattern design in test track to be different from the train track. For example, if the train track starts as a straight line, a right turn and a left turn, then the test track could start as a left turn, a straight line and right turn.
2. Making the curve design in the test track harder than the test track to guarantee that the model works properly.

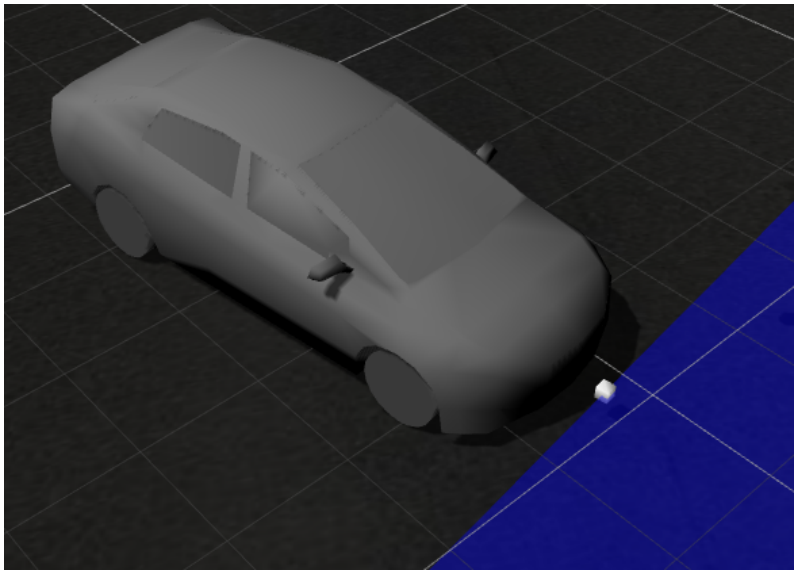


Figure 15: Simulation of Car

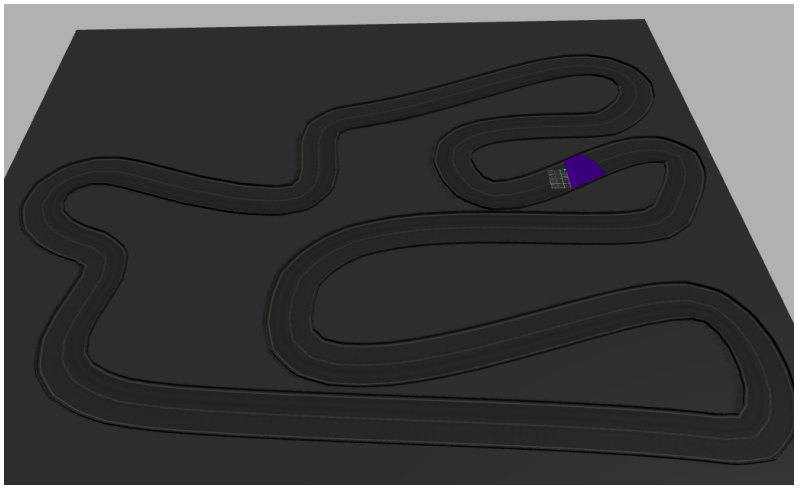


Figure 16: Simulation of Train Track

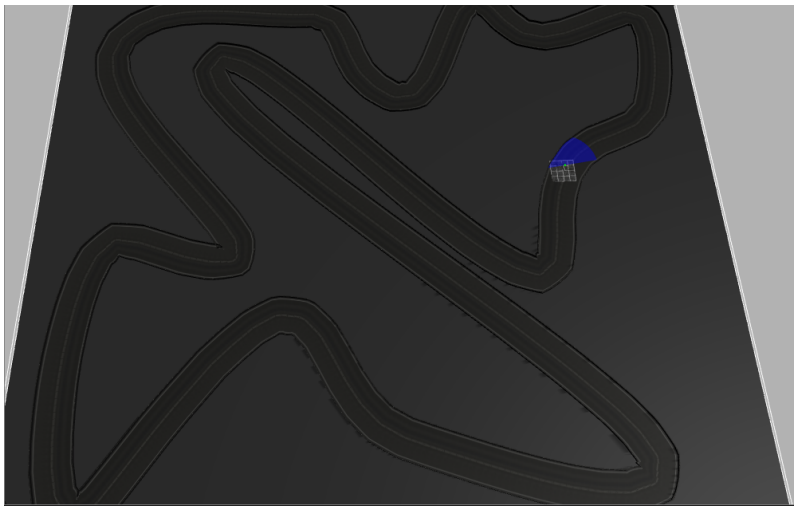


Figure 17: Simulation of Test Track

4.2 Q-learning approach

4.2.1 Tile Coding

Tile Coding is a type of linear function approximation with the ability to make linear combinations of several overlapping Q-tables with higher resolution close to the state space. State dimensions are partitioned into tiles (fields). If the input lies within the tile boundaries, the tile is marked as an active tile with a value of one. Otherwise, the tile is marked as an inactive tile with a value of zero. Tile coding can be represented as a uniform grid map with many dimensions. This technique is able to generalize and reduce the discretized states because of two reasons. Firstly, the States that results in having the same active tiles are considered as the same state. Secondly, each update affects all states that result in having active tiles. Figure 18 is an example of approximating value functions with tile coding for a

two dimensional continuous problem. Here the dot represents the actual measured continuous state, note that all dots within the intersection of the two active tiles will be considered as the same state as the dotted state.

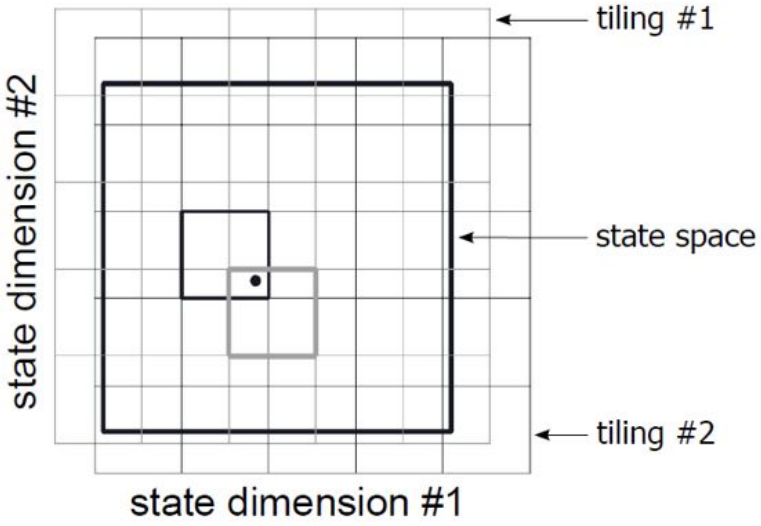


Figure 18: Tile Coding

4.2.2 Discretization

Input for the Q-learning algorithm every time step is an array of 50 laser-readings covering 180 degrees ahead of the vehicle with each laser-reading having a decimal value between 0 and 30 representing the distance between the car and the nearest object in that direction. Being a discrete algorithm , Q-learning cannot deal with continuous values so the first idea was to approximate each laser-reading to the nearest integer. However, having 50 readings with each one having 31 possible values would results in a state space with an order of magnitude of 52 which would take practically infinite amount of time to train. So Tile Coding was used to reduce the number of possible states significantly. After iterations of trial and error we settled on taking the mean of every 10 laser-readings into one value. So, we ended with only 5 representative averaged laser-readings as shown in Figure 19.

Figure 19: Laser shown in blue, each of the 5 sections was averaged into one reading

Tile coding was also used to reduce the possible values of each laser-reading from 30 into only 3 possible values (5,15,25). Finally, we ended up with a state-space of 243 possible state.

Output of the Q-learning algorithm every time step is the steering angle which is the action to be executed by the vehicle. Actions also have to be discretized in order to be inserted and extracted from the Q-table. The range of steering values varied from -0.3 to 0.3, so it was discretized with a 0.1 step size. We ended up with action-space of 7 possible values. After all the discretization and reduction the state-action space was reduce to 1701 possible state-action pair which meant a Q-table with 1701 cells. Figure 20 shows a snippet of a small part of the Q-table after few episodes of training. Each column represents a specific state and each row represents a specific action. Hence, every cell contains the action-value function Q(s,a) of a specific state-action pair. From the definition of Q(s,a) it is clear that the highest value in each column corresponds to what the agent believes to be the best action to be taken at this specific state.

-8.849945695	35.3544735888	28.4416955657	-0.3119072905	-106.406288107
-9.9448624854	25.0969718827	28.4458005409	-0.6591235998	-106.053604914
-8.0882281616	33.2532603637	28.3102367591	-0.0388766511	-104.703483436
-7.0567925207	28.2940141801	27.4444832238	-0.1113316151	-101.624424254
-8.7955192777	26.9915073111	28.8814711019	0	-90.2263570946
-7.6309557834	31.9707124555	28.486457032	0	-102.48035943
-7.7174482877	31.3260180076	28.4278222605	-0.4446189638	-100.910392526

Figure 20: Part of Q-table after few iterations

4.2.3 Reward Function

The criterion for Reward function design was to reward the agent positively as long as it is away from any obstacle including the pavement, and to reward the agent negatively whenever it approaches an obstacle. In addition, the agent was to be given a relatively high negative reward whenever the vehicle crashes, flips or loses contact with the ground. It also should be awarded negatively when its velocity is too slow. Finally, the absolute difference between the laser-readings from both sides of the vehicle is added to the Reward function with a negative reward to ensure the vehicle is sticking to the middle of the lane. The coefficients of each element of the reward function were considered hyperparameters, their initials values were chosen by intuition and after some refinement using trial and error the final Reward function is as follows.

$$R = 0.5 * (FrontLR - 15) - 0.5 * (LeftLR - RightLR) - 5 * slowMove$$

Where FrontLR is the value of the laser-reading pointing forward, LeftLR and RightLR are the values of the leftmost and rightmost laser-readings as shown in Figure 21. Whereas slowMove is a binary variable that returns 1 when the car is flipped, stuck or moving too slow and it returns zeros otherwise.

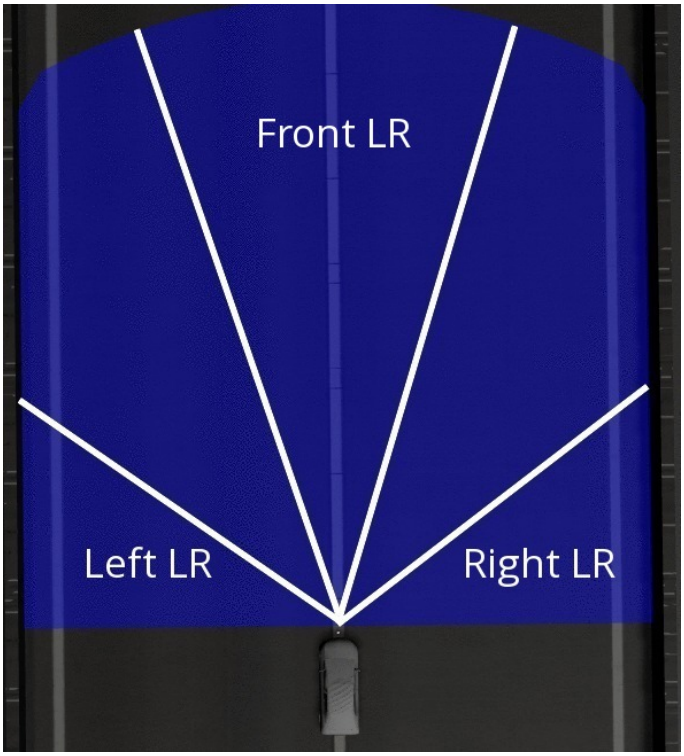


Figure 21: Laser Readings used in Reward function

4.2.4 Exploration

To achieve the balance between exploration and exploitation, we used ϵ-greedy mechanism during the learning phase. ϵ-greedy is a well known exploration technique in RL algorithms. A greedy policy will always pick the action with the highest action-value function Q(s,a) in order to maximize the estimated return, this would result in a totally exploitive agent that will hardly learn anything. Alternatively, an agent with ϵ-greedy policy ensures that a small probability (ϵ) of the chosen actions is totally random, this policy allows the agent to explore unknown areas of the state-action space which will help in the process of learning. The ϵ-greedy policy can be formulated by

$$\pi(s_t) = \begin{cases} \text{Random action a} \in A(s_t) & \text{if } (\rho < \epsilon) \\ argmax_a Q(s_t, a) & \text{otherwise} \end{cases}$$

Where (ρ) is a random number in which ($0 < (\rho) < 1$) and it is generated randomly every time we choose an action. (ϵ) is a tuning hyperparameter which is chosen to be 0.1, this means that supposedly there is one random action out of every 10 actions performed by the agent.

4.2.5 Algorithm

The algorithm followed is Q-learning algorithm discussed in section 3.3.3. The algorithm has two hyperparameters, discount factor (λ) is set to 0.9, and learning rate (α) is chosen to be a decaying function of time.

$$\alpha = t^{-0.15}$$

After the Q-table converged we were able to see the results on the training track and test the agent on the test track.

4.3 DDPG approach

We have developed this code on two stages. First stage is learning the steering of the vehicle with constant speed. Then we made the vehicle learns to accelerate and steer. Main code is the DDPG.py code. Same code starts training or testing based on the input. Firstly, the required networks of actor and critic are created with the dimensions of the states for the input and actions for the output.The Algorithm consists of actor network and critic network as mention in **??**. Actor network consists of two hidden layers consists of 300 and 600 nodes respectively as shown in figure **??**. The activation function of those layers is the relu function.

4.3.1 For training steering only with constant speed

The output of the second layer enter output layer and is activated by tanh function. it gives a range from -1 to 1. We mapped this range to (–1/2 to 1/2)radian so it will have maximum left and right steering but the vehicle will not rotate around itself. This decreases the training time. input for the neural network is 7 states. 5 of them are ranges of laser records. Each laser positioned at a specific position in the vehicle. Each laser record is the average of 10 normalized reading. Laser reading gives an output of max 30 meters. We need the input to be normalized for the neural network. Then we have speed and angle measured as the remaining states.

4.3.2 For training steering with acceleration

Steering with acceleration experiment is built on steering only experiment. we add another output layer after the second hidden layer that could output the acceleration. We used tanh as activation function that gives output in the range -1 to 1. This range enables us to make brake and acceleration by the same output. we needed to have more laser ranges than the steering. So we have made an average of

every two laser readings that result in 25 laser range. we added 4 contact sensors that could sense if the vehicle is raised above the ground in case of high speed. This will end the running episode and start a new episode. We didn't need to add contact sensors in steering as it is moving with constant speed and it will not be raised above the ground. So the number of states entering the neural network become 31 states. We needed more states as we have more actions that needed to be taken.

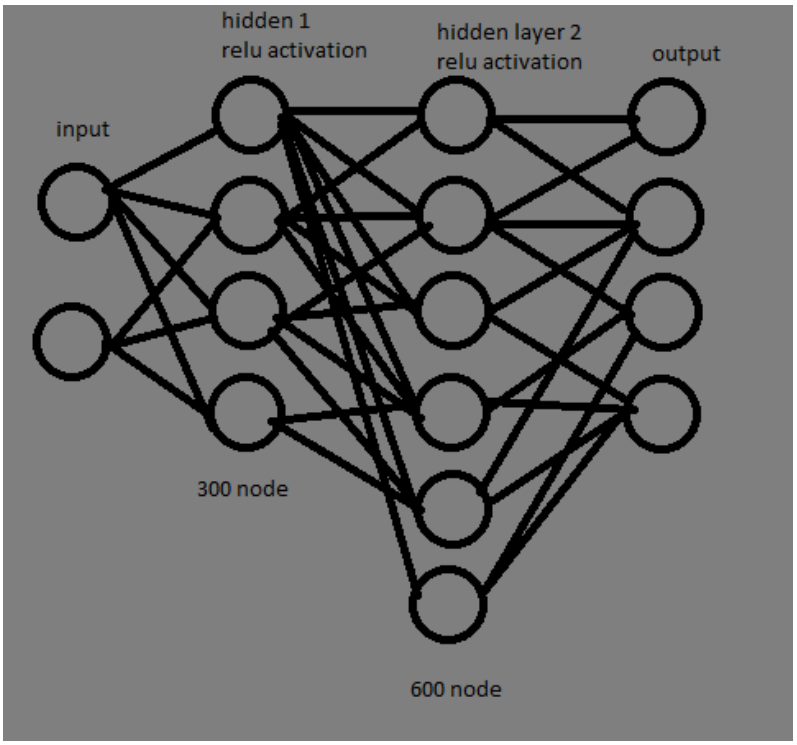


Figure 22: Actor network in DDPG algorithm

Actor Network builds the model that maps states to actions to make a generalization for states that have similar values but unvisited.

The critic network firstly passes the states through two hidden layers relu and linear activation functions respectively. The actions are also passed through linear activation functions. Then states and actions are merged by summation. This sum enters activation function relu then the output is shown in the final layer. The algorithm used in neural network for both actor and critic networks is ADAM optimizer.

This critic network sees the actions taken by the actor network and gives it feedback on how good are they to modify its weight.

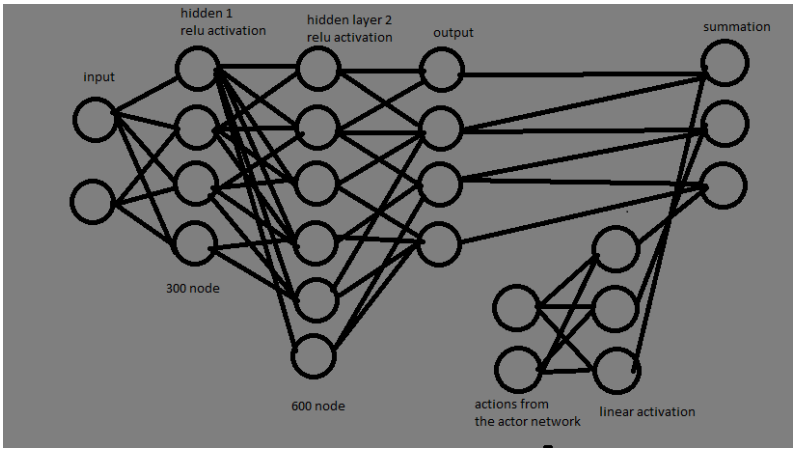


Figure 23: Critic network in DDPG algorithm

After initialization of the network Ros environment is initialized

4.3.3 Ros environment

Python codes communicate with ROS environment through publishers and subscribers. The required subscribers are declared at the beginning of the code. Then methods of dealing with them are declared. subscribers are used to communicate messages from code to ROS. In the project those are the steering, brakes and acceleration actions. While Publishers are Messages from ROS to the code. Those messages include sensors data such as Laser and speed. It also includes the reset message that makes the vehicle starts a new episode in case that it matched termination conditions. We define those termination conditions to be vehicle stuck in the wall or having very low speed.

Ros environment is then initialized. The required callbacks are defined. observation, reward and reset functions are defined.

4.3.4 Observation, reset and reward functions

Observation function gives back speed, angle and laser readings. Reward for steering only with constant speed Then the reward function is defined to negative the difference between the two side ranges and the summation of the front sensor. Each range is the average of 10 laser reading. the vehicle learned how to minimize the negative value of side sensors. In our experiments, it learned to keep itself in the middle of the road and the associated actions. It learned also those actions that make it have the maximum front sensor that is achieved by turning with the road.

we have also added negative reward on slow move so it learns to have faster speed

$$\begin{aligned}\text{Reward function} \;=\; & const_1 * (FrontLR - 15) \\ & - const_2 * abs(LeftLR–RightLR) - const_3 * (slowMove)\end{aligned}$$

constants are hyper parameters we tune them till we have the best performance

4.3.5 Reward for steering with acceleration

For steering with acceleration, we needed to give a reward for the velocity and penalize high drifting in steering. We achieved this by multiplying the reward function by the velocity and subtracting the angle from the reward function

$$\begin{aligned}\text{Reward function} = normalized\_velocity * (const_1 * (FrontLR - 15) \\ - const_2 * abs(LeftLR–RightLR) \\ - const_3 * angle) - const_4 * (slowMove)\end{aligned}$$

We also subtract -1 from the reward for every time step in which the contact sensors are above the ground. To interpret that the car has lost contact with ground which happens a lot in case of random high jerk or acceleration.

The reset stuff function returns the vehicle to the start of the road. Initialize the velocity and angle by zero values. Then the step function is called. This function takes the actions and apply the agent's actions from the network's model to ROS and call the reward function.

4.3.6 Exploration implementation

Using the above functions code runs many episodes. Each episode has certain number of steps that will terminate if it reached the maximum number of steps or reached termination conditions. We define the maximum steps per episode to be 100000. We have put some exploration in the first steps of the whole episodes. Epsilon is defined to be the exploration rate. Steering and acceleration need to be explored as at the beginning there is no model that could move the vehicle. For steering only we needed 7000 step to have sufficient exploration, However in steering with acceleration we needed to have 200000 step of exploration as the action space increased dramatically. Ornstein-Uhlenbeck process was used instead of using normal random function. Using normal random function may generate numbers that are conflicting. It may generate brakes more frequent and with high values, while we need to accelerate more. However, in Ornstein-Uhlenbeck we define the mean of the random function. We also define how fast these actions return to

mean. Ornstein-Uhlenbeck have three variables θ, σ and ν. θ is the mean-reverting variable. σ is the variance of the random process and ν is the mean. We had to tune these three parameters a lot to find the suitable set of numbers. We needed steering to be most of the time straight so we made the mean 0 and we needed to explore small values around the zero. In acceleration we made the mean above the zero to have acceleration most of the time and to have brake not frequent. we made the variance high to try all the possible values. We made a lot of tuning. We see if the actions after the random function make sense or not till we reached smooth performance.

We build one code for testing and training. We used variable train indicator to be 1 or 0. If 1 the agent will load the model, take actions based on the coming states and the model then add the noise at the end of the exploration. This noise is multiplied by the train indicator. If we are testing the model the noise will be multiplied by 0 and no noise will be added. However, if train indicator is 1 the noise is generated by the Ornstein-Uhlenbeck multiplied by the weight of exploration epsilon and then added to the taken actions

After the actions are taken. Old states, actions, reward and new states are saved in an object of class replay buffer made to handle the model update

Every new record is saved in the buffer then we make batch update for the built model of batch size. We choose the batch size to be 32 random records. This batch update represents stochastic update for the model to have faster convergence. The training of the neural network continues after the exploration finishes as we have a large buffer that we made random update from it.

4.4 CNN approach

For CNN algorithm, it is divided into 4 main steps, collecting training and testing data, writing CNN model, put temporary evaluation criteria, modifying cost function, analyzing the training data and inserting a crash avoidance model. These steps are essential to enhance the training phase and the testing results.

4.4.1 Data Gathering

The CNN algorithm is a supervised algorithm. It needs a training data for the learning phase and a testing data for the testing phase. In order to collect train and test data, an algorithm called informed action is written. It depends on laser sensor readings by analyzing the readings into the x and y components, adding them together and using trigonometry to calculate the steering angle that the car should deviate with. By combining the steering angle that is calculated by informed action algorithm to the captured image by the camera sensor, the train and test data can be ready. The captured image is a front image (200x200x1 pixels - grayscale)

captured by the ROS camera sensor.

In the following figure, the image view window is the image captured from the front camera.

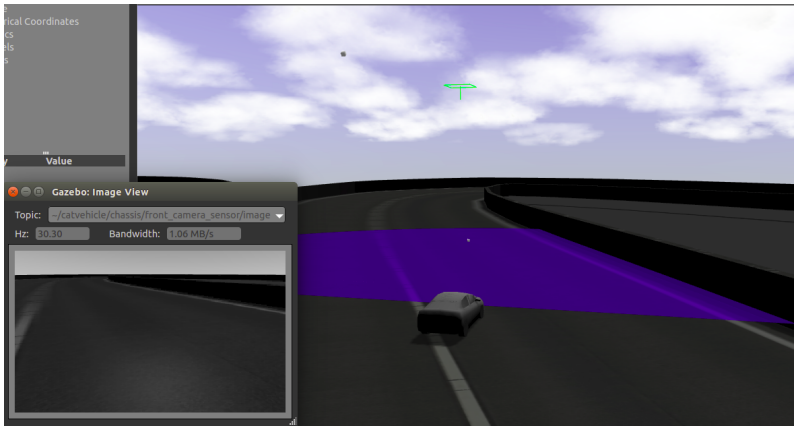


Figure 24: Screen shot From the running simulation

4.4.2 CNN model

As described before in Possible Solutions section, the CNN architecture consists of the input layer, 5 convolutional layers, flatten layer, 3 fully connected layers, and regression layer. The input layer takes an image of size (200 x 200 x 1). Then, the convolutional layers extract features from the image. After that, the flatten layer and the fully connected layers work on data. Finally, the regression layer works as a function approximator. It is just a single neuron but doesn't have an activation function to work a regression function, not a classifier.

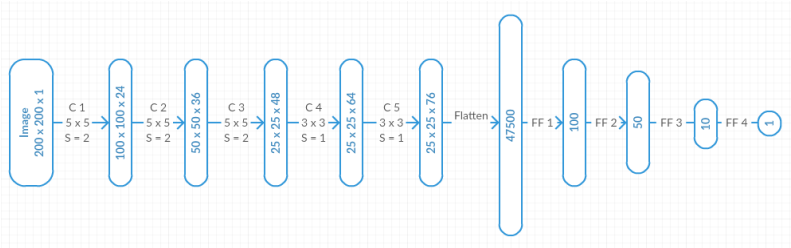


Figure 25: Convolution Neural Network Layers

After implementing the model, the training phase begins with 2 things:

1. The extracted data from informed action algorithm.
2. The cost function which is Root Mean square Error.

$$Cost = \sqrt{\frac{\sum_{i=1}^{n}(Predicted\ Output_i - Exact\ Output_i)^2}{n}}$$

Model evaluation

Before the final evaluation step in our project, a temporary criterion has been adopted to enhance the performance of the algorithm. First, the cost function should decrease as the learning proceeds. This criterion is essential to ensure that the model architecture works properly and fits the training data.

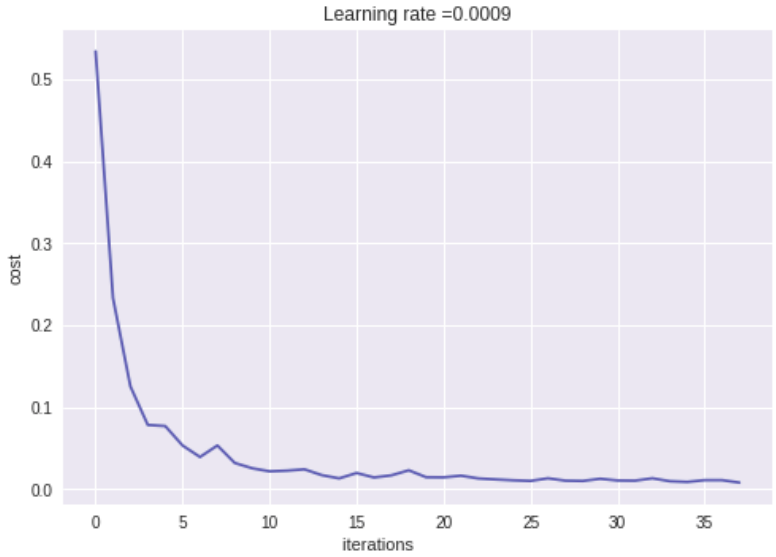


Figure 26: Learning Curve graph

The second criterion evaluates the test phase temporally. The deep learning problems can be classified into two categories, classification and function approximation. The classification problems have a well-defined criterion to evaluate the testing phase as the number of the true and false answers can be computed and used to give an indication of the behavior of the classifier. On the other hand, the function approximation problems have no such criterion.

For example, in our case, the CNN algorithm results in a steering angle. The steering angle will have an error. This error could be critical and could be not. If the evaluation criterion is the same as the cost function which is the root mean square error (RMSE), the critical error will not be noticed. If the car has 10 orders of steering angle which has a critical error that will cause the car to crash while there are 5,000 orders with a negligible error, the RMSE will be negligible and the critical error will be considered as an outlier. That's why the RMSE and the averaging techniques are not appropriate for our case. The CNN algorithm may behave well in all the test track but at a portion of it, the algorithm behaves with a critical error that causes an accident. This portion of error needs to be considered.

In order to consider the critical error, it has been noticed that if the car has more

than 10 critical errors without interruption, then the car will be exposed to an accident. This makes our evaluation criteria (accident counter criteria) is the number of 10 critical errors without interruption in the test data and the error is critical if its absolute is more than 5 degrees.

4.4.3 Cost function

The RMSE function may not be appropriate to decrease the critical error although it makes the model converges. As a result, the cost function is modified to include the critical error by adding the square of the maximum error among each epoch to the RMSE function. This will guarantee that if the model has critical errors that will cause an accident, then the model is still in the learning phase. Besides, squaring the maximum error gets rid of negative signs and gives higher priority to it.

$$Cost = \sqrt{\frac{\sum_{i=1}^{n}(Predicted\ Output_i - Exact\ Output_i)^2}{n}} + \max error^2$$

This step decreases the number of accident counter criteria.

4.4.4 Data analysis

The histogram of the input data is plotted to see if the distribution of data among the possible values of the steering angles.

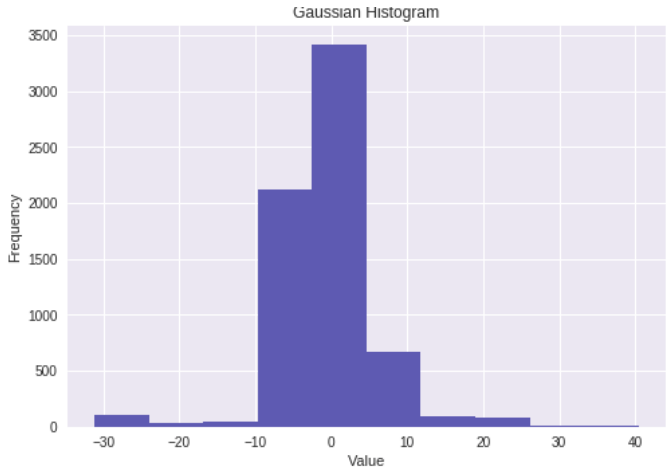


Figure 27: Histogram of the input data

The x-axis represents all the values of the steering angles in the training data and the y-axis is the frequency of them. It is observed that most of the data is between

-10 degrees and 10 degrees while the large steering angles have small input data. This problem is caused because of most of the time, the car move in a straight line which makes most of the data centered around zero degrees. As a result, the CNN algorithm deals with the rest of the data as outliers. This problem makes the car more exposed to accidents. To overcome this problem, some data from -10 degrees and 10 degrees are removed to make the histogram more balanced.

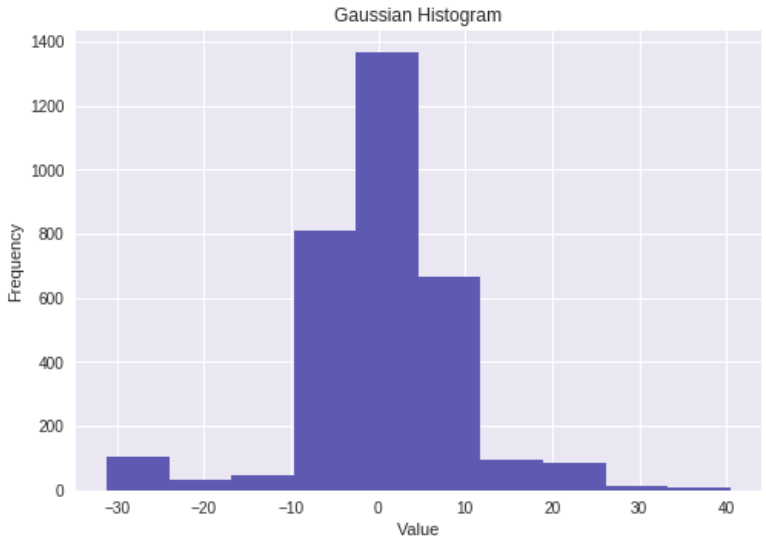


Figure 28: Histogram of the input data with less data between -10 to 10 degrees

This step enhanced the performance of CNN algorithm. The number of accidents according to the evaluation criteria decreased but it still needs to be enhanced. We cannot remove more data to balance the histogram more as the data will be challenging for the CNN algorithm.

4.4.5 Crashing avoidance model

The CNN algorithm is capable of fitting the training data. Any steering angle between -15 degrees and 25 degrees can be predicted. But any angle bigger than 25 degrees cannot be predicted as the CNN trained on a very little data on that range which can be assumed as outliers. This represents two problems, not just one. The first problem is any angle above 25 degrees or less than - 15 degrees cannot be predicted due to the lack of data. The second problem is the CNN is not trained to behave properly when the car is going to crash. The training data has no data represents the behavior that the car should do when is about to crash.

To overcome these problems, a crash avoidance data should be extracted from the simulation. To do so, the informed action algorithm will be modified. First, it will predict the steering angle of the current situation of the car. Second, a random offset angle will be added to the steering angle to make the car exposed to crash

in the wall. Third, the informed action will predict another angle for this situation and save it in the training data. This modification makes the car exposed to crash and then avoid the crash which creates data represent large steering angle and the act that the car should do when it is going to crash.

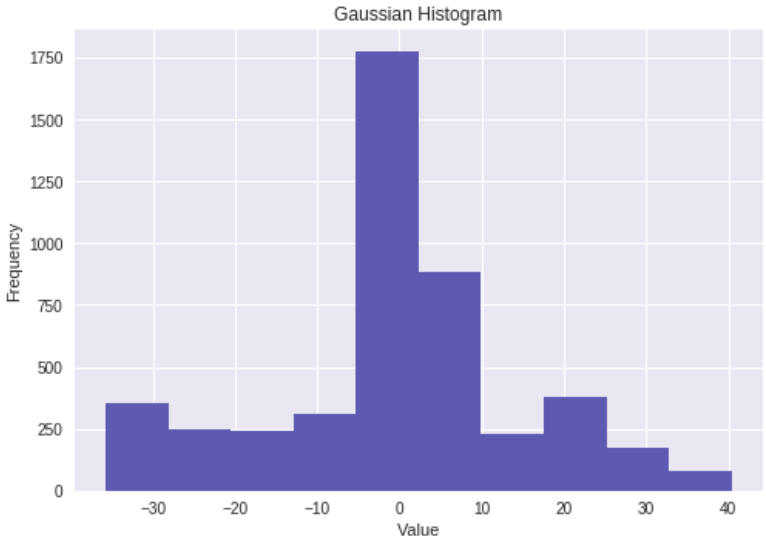


Figure 29: Histogram of the input data

The results from this step make the number of accidents very small according to the evaluation criteria. Besides, if the car goes off the lane and about to crash, it will avoid it based on the crash avoidance training data.

5 Economic Analysis

Lane keeping is a first basic step in creating a full driverless car that is able to change lane or to detect and avoid crashes .In this section we discuss the economical aspect of a self-driving car as an end product. First of all, we mention that 93% of the accidents are caused by human errors (for Statistics & Analysis, 2015), a self-driving car will make the roads safer, leading to a reduction in the repair, legal and human injuries related cost. For example, Egypt loses 14 billion EGP annually due to traffic accidents and is now on the top list of traffic accidents in the world.Using self-driving car may lead to a reduction of the revenue of the insurance sector and an increase in the revenue of the mechanics; as the car needs to be periodically checked for maintenance in order to avoid accidents.

A reliable driverless car (level 5) that doesn't need any interventions from human could make several trip by its own, which will reduce the need for parking as cars will no more need a permanent home during the day. This is a huge economic benefit as Egypt spends 6 billion EGP on parking slots.

Driverless cars are not constrained on personal use, trucking companies could save up to US$500 billion dollars annually by 2025 (Barbier, 2017), as automated trucks could deliver food, commercial products, industrial raw materials which will cut down the salary of drivers and may lead to their extinction, on the other side this will lead to a potential reduction in the cost of goods being sold. The same is applied on drivers of the public means of transport such as the taxi drivers and the bus drivers leading to a positive impact on the travel industry.

Another important impact of self-driving car is the amount of data that could be collected from sensors about every aspect of the day, these data are of huge benefit for IT and data analysis as these data could be worth up to US$750 million by 2030 (Marshall, 2017).

Driverless car will also open the door for disabled individuals to increase their mobility and thus their chances in life and their employment prospects; it's estimated that British individuals with limited mobility could see their earnings increase by £8,500 annually, on average. (Tovey, 2017)

6 Environmental Aspects

Self-driving car is the trend of the future. Studying the environmental aspects helps manufacturers, software developers, and policy makers to determine the factors affecting the environment, and to take them in consideration in order to increase efficiency and help leveraging the negative impact on the environment.

A study made by the Center of Sustainable Systems, in the University of Michigan; has concluded that AV could have a huge positive impact as well as a widely negative influence on the environment. This will largely depend on the decisions to be yet taken by the policy maker and the choices to be made by the technological sector.

For example, an autonomous car is heavier and have more equipment that the ordinary car, there is a bulk of sensors, cameras, processors and communication systems, used in navigation, which consumes more power and dissipates a big amount of heat. The study also found that AVs contribute to the greenhouse effect by 20% more than the human driven car. It stated that the on-board computer systems consumes 80% of the power and represent 45% of the car (Worland, 2016).

In our project we tried to reduce the number of sensors used in the navigation system, our algorithm depends only on one camera and two laser sensors. This technique not only reduces the power consumption and amount of heat dissipated but also the amount of aerodynamic drag applied on the vehicle which helps increasing the efficiency of fuel consumption.

Researchers found that using electric power trains has lifetime greenhouse gas emissions that are 40 percent lower than vehicles powered by internal-combustion engines. Also using Connected autonomous vehicles, that share the data and coordinate the decisions between each other lead to a 14% reduction in energy (TechExplorist, 2018).

7 Discussion and Evaluation

7.1 Evaluation Criteria

After training all 3 different algorithms on the training track, their performance is to be measured and compared on a specific testing track. However, comparing their performances should be based on well-designed evaluation criteria. After reviewing the different evaluation criteria used in literature,The following 5 were chosen.

Training time, which is the time the algorithm takes to converge during training.

Number of completed laps, which is the number of laps the vehicle managed to complete without crashing.

Mean lap time, which is the average time the vehicle takes to finish one complete lap on the testing track.

Mean absolute change in steering angle per second, which is the average of absolute change in steering angle per second. This works as an indication of how smooth the steering of the vehicle is and how much unnecessary change is happening in the steering angle. It is known that consecutive fluctuations in steering would be harmful for the vehicle and uncomfortable for the passengers. Its value is calculated using the following equation.

$$\Delta\theta_{av} = \frac{1}{N}\sum_{k=1}^{N}\left|\frac{d\theta_k}{dt}\right|$$

Where $d\theta_k$ is the change in steering angle at time sample k, N is the number of time samples per lap and dt is the time step which is equal to 0.04 seconds.

Mean deviation per lap, which is the percentage of time the vehicle failed to stay within two meters from the centerline of the lane. This criterion gives an indication of how successful the agent is in sticking to the center of the lane.

7.1.1 Training time

CNN was trained on google colab so it have much less time than the other algorithms. This method was only possible with CNN as we extracted the training data from the simulation and then upload them to get the model. However this is not possible with RL algorithms as we need real time processing of data. Every new episode depends on the past experience. We couldn't upload real time to google colab. Q-learning steering with constant speed had training time of 3 hours,while DDPG steering had 6 hours to train. this is due to the training of neural network of the function approximator that takes much time than Q-learning. Q-learning is

just training the Q table of finite set of actions. DDPG steering with acceleration takes more time as the action space increased dramatically due to acceleration.

Table 1: Training time in hours

Training time			
Q-learning	DDPG Steering only	DDPG Steering+Acceleration	CNN
6 hrs	3 hrs	9 hrs	**0.3 hrs**

7.1.2 Number of completed laps

Mainly, this Criterion is added to determine if the algorithm failed to run the vehicle on the test track. However, we managed to develop successful models in all mentioned algorithms in this report. All models succeeded in finishing more than 20 laps with highly acceptable performance in both the training track and the different and more difficult test track. The test track has difficult and sharper turns but with the same width.

Table 2: Number of completed laps

Number of completed laps			
Q-learning	DDPG Steering only	DDPG Steering+Acceleration	CNN
>20	>20	>20	>20

7.1.3 Mean lap time

DDPG with acceleration finished the lap fastest as it learned to accelerate. However DDPG with constant speed have similar time to CNN that have same speed. it only changed a little bit, this may be the result of different steering behavior. Q-learning with the same speed as DDPG and CNN, have the longest time as it's actions are finite and discrete so it loses some time to keep the center of the track. the recorded time is shown in table3

Table 3: Mean lap time in seconds

Mean lap time			
Q-learning	DDPG Steering only	DDPG Steering+Acceleration	CNN
375.48 sec	340.3 sec	**224.01 sec**	347.0 sec

7.1.4 Mean absolute change in steering angle per second

DDPG and CNN have a range of steering angles that the difference is not large. the vehicle could fine tune it's angle to keep the center of the track so the mean is low. However, in Q-learning the action space is discretized so every action makes a big difference. We predicted that DDPG steering with acceleration gives slightly higher mean, however, we found that the mean is large. We refer this mean to the large action space of steering and acceleration. The vehicle model recognized that this large steering for short steps didn't make negative reward so it kept this actions for very short steps. We made sure that these actions are only for short steps. We inspected the video of the testing, we observed smooth performance and we didn't notice that big variation. We also inspected the histogram of the steering actions as shown in figure30 and figure31 we found that the most frequent steering difference are around the zero and a very small percentage that has high values. This model still couldn't apply to a real application. We need to apply Kalman filter to smooth these values. We could also limit the action space for steering in the range that keeps the vehicle from inverting and slipping

Table 4: Mean absolute change in steering angle per second

Mean absolute change in steering angle			
Q-learning	DDPG Steering only	DDPG Steering+Acceleration	CNN
32.6 deg/sec	6.79 deg/sec	57.84 deg/sec	**2.39 deg/sec**

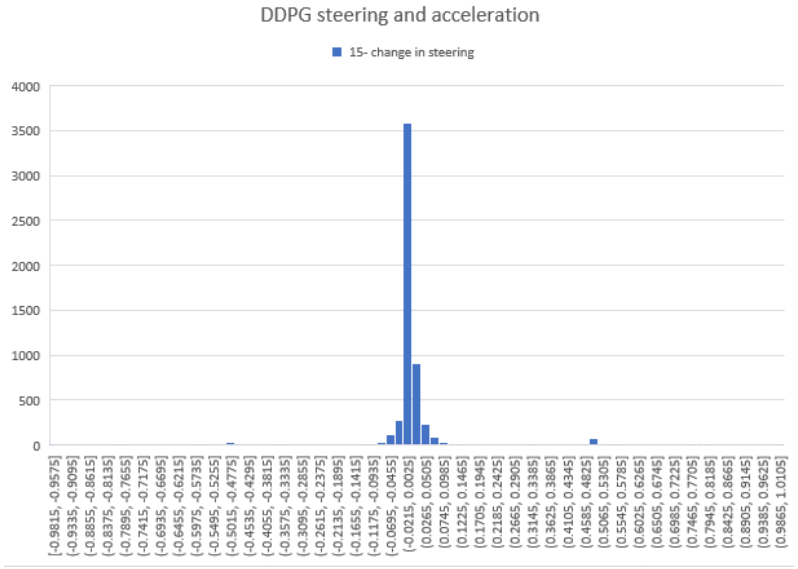


Figure 30: histogram of steering changes for DDPG steering with acceleration

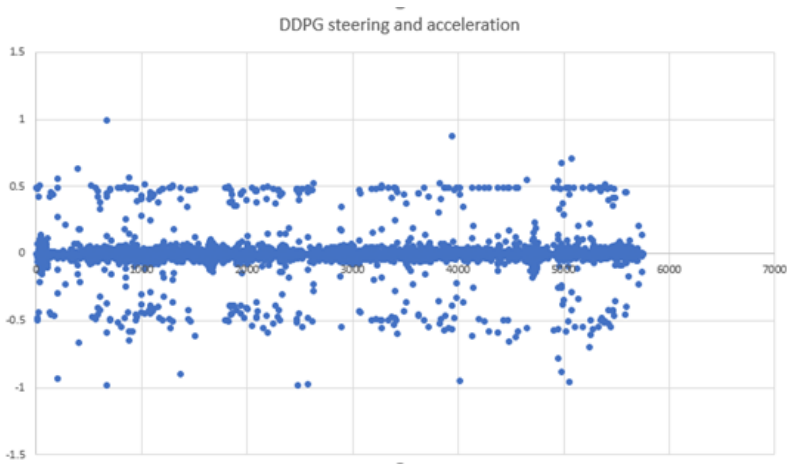


Figure 31: time with steering changes graph for DDPG steering with acceleration

7.1.5 Mean deviation per lap

As shown in figure32 and figure33, trained model of steering only is similar to trained model for both steering and acceleration. Both models keep around the center most of the time. The time spent out of track by 5 meter is 2% of the total time steps for steering only model and about 7% for acceleration and steering model. This is acceptable range as the testing track have many hard turning. the vehicle might have found it suitable to keep away this 5 meter to get the round smoothly. the results are summarized in table5

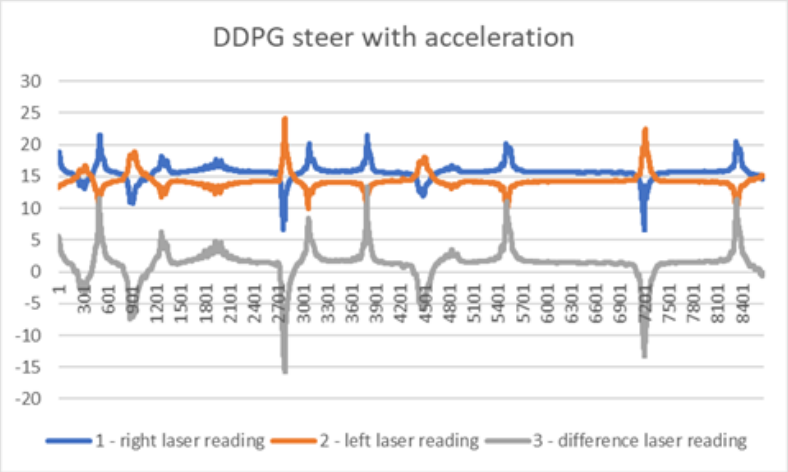


Figure 32: Center line Deviation of DDPG steer and acceleration

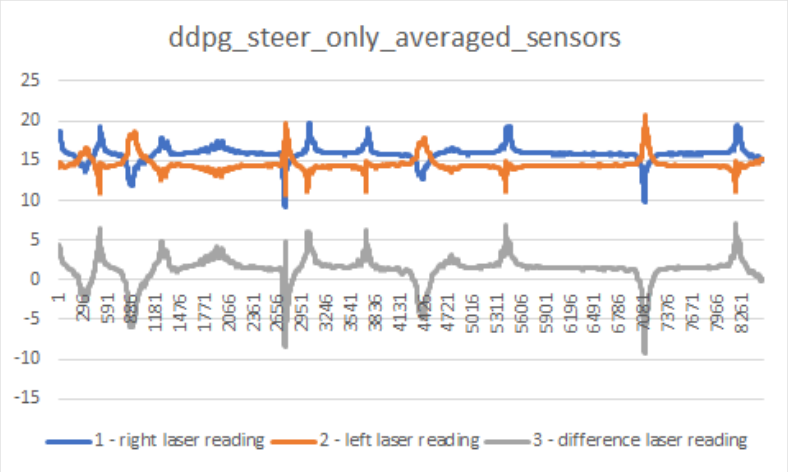


Figure 33: Center line Deviation of DDPG steering only

Q-learning is discrete algorithm. It chooses its actions from finite set of actions. This finite set make the vehicle oscillate between them. It is obvious in figure34 that the vehicle gets out of the track by 5 meter most of the time. Deviation was found to be 57.87%. However in ddpg and CNN there are a range of continuous actions that could easily be taken based on the distance from the center of the track. This behavior guarantees smooth performance that could keep the center of the track.

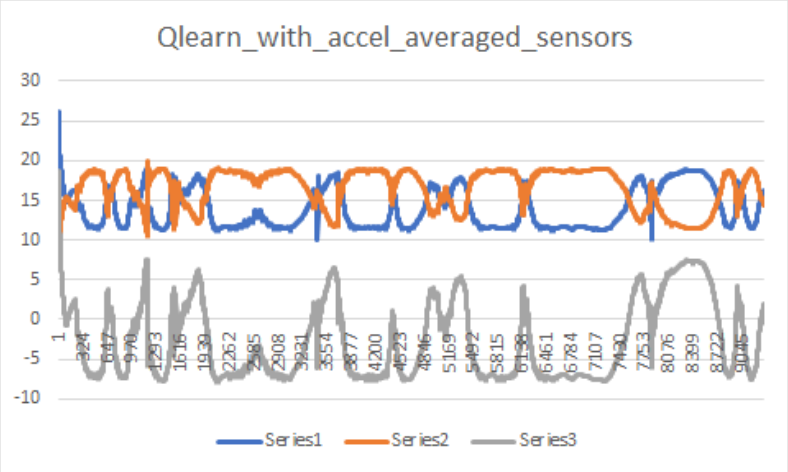


Figure 34: Center line Deviation of DDPG Q-learning

CNN algorithm gave similar behavior to ddpg. It gets out of the track by 5 meters 4% of the total time. The overshots in figure35 is due to the hard turns in the test track. When the CNN model faces a hard turn, the car takes wide turn to change the steering angle smoothly.

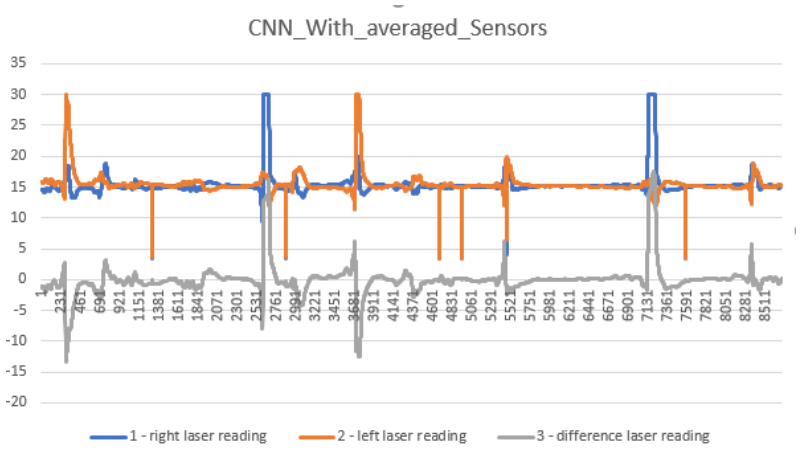


Figure 35: Center line Deviation of CNN

Table 5: Mean deviation per lap

Mean deviation per lap			
Q-learning	DDPG Steering only	DDPG Steering+Acceleration	CNN
57.87%	**2.59%**	7%	4.0%

7.1.6 Final Remarks

We have summarized all the evaluation criteria in table6. We have found that the most suitable algorithm for the real-life application is the DDPG steering only and CNN algorithm. Those algorithms provide comfort for the passenger in the sense of safe steering. Also, DDPG with acceleration could be used, however, it needs more filtering to filter out high acceleration and steering overshoots that could harm the passenger. Q-learning couldn't be used in real life application as there many cases that appear suddenly that might not have associated actions in the Q table.

Table 6: Summary of comparisons

Criterion	Q-learning	DDPG Steering only	DDPG Steering+Acceleration	CNN
Training time	6 hrs	3 hrs	9 hrs	**0.3 hrs**
No. of completed laps	>20	>20	>20	>20
Mean lap time	375.48 sec	340.3 sec	**224.01 sec**	347.0 sec
Steering angle change	32.6°/sec	6.79°/sec	57.84°/sec	**2.39°/sec**
Deviation per lap	57.87%	**2.59%**	7%	4.0%

8 Conclusion and Future Work

To conclude, We managed to build a realistic car model on ROS with the appropriate sensors to interact with the surrounding environment. Then, the lane keeping task is accomplished using three algorithms, CNN, Q-learning and DDPG. Results demonstrated the ability of all algorithms to control the vehicle and to complete the test track autonomously without crashing into the roadsides. Additionally, a comparative study was made to evaluate the performance of each algorithm. Based on our criteria, it is concluded that the behavior of DDPG and CNN agents is better than Q-learning agent. Finally, the DDPG algorithm is enhanced to control not only the steering angle but also the acceleration which decreased the time taken to finish one lap in test track.

For future work, the crash avoidance task can be enhanced by adding more cars in the training and testing phases to come up with a model that is capable of avoiding crashes with other cars. Besides, the tracks can be enhanced by making cross roads to train the car when to brake.

References

Aly, M. (2008). Real time detection of lane markers in urban streets. *In Intelligent Vehicles Symposium, IEEE*.

Anderson, L. (2000). Top 10 linux games for holidays by linux journal. *https://web.archive.org/web/20041206174427*.

Barbier, J. (2017). Self-driving cars will disrupt more than the auto industry. here are the winners and losers. *https://www.cnbc.com/2017/05/03/self-driving-cars-will-disrupt-10-industries-commentary.html*.

Bernhard. (2016). Torcs. *http://torcs.sourceforge.net*.

Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., ... Zieba, K. (2016). End to end learning for self-driving cars.

DARPA-IPTO. (2004). Autonomous off-road vehicle control using end-to-end learning: Final technical report. *Net-Scale Technologies inc.*

Fischer, J., Falsted, N., Vielwerth, M., Togelius, J., & Risi, S. (2015). Monte-carlo tree search for simulated car racing. *United States: Association for Computing Machinery*.

for Statistics, N. N. C., & Analysis. (2015). Traffic safety facts.

Gackstatter, C., Heinemann, P., Thomas, S., & Klinker, G. (2010). Stable road lane model based on clothoids. In *Stable road lane model based on clothoids.*

Gurghian, A., Koduri, T., Bailur, S. V., Carey, K. J., & Murali, V. N. (2016). Deeplanes: End-to-end lane position estimation using deep neural networks.

Hadsell, R., Sermanet, P., Ben, J., Erkan, A., Scoffier, M., Kavukcuoglu, K., ... LeCun, Y. (2009). Learning long-range vision for autonomous off-road driving,. *Journal of Field Robotics*.

Huval, B., Wang, T., Tandon, S., Kiske, J., Song, W., Pazhayampallil, J., ... Ng, A. Y. (2015). An empirical evaluation of deep learning on highway driving. *arXiv, Cornell University*.

Innocenti, C., Linden, H., Panahandeh, G., Svensson, L., & Mohammadiha, N. (2017). Imitation learning for vision-based lane keeping assistance.

Karavolos, D. (2013). Q-learning with heuristic exploration in simulated. *Artificial Intelligence. Amsterdam: University of Amsterdam*.

Kardell, S., & Kuoscu, M. (2017). Autonomous vehicle control via deep reinforcement learning. *Department of Electrical Engineering,CHALMERS UNIVERSITY OF TECHNOLOGY, Gothenburg, Sweden*.

Koutnik, J., Cuccu, G., Schmidhuber, J., & Gomez, F. (2013). Evolving large-scale neural networks for vision-based reinforcement learning. *the 15th Annual Conference on Genetic and Evolutionary Computation*.

LeCun, Y., Muller, U., Ben, J., Cosatto, E., & Flepp, B. (2005). Off-road obstacle avoidance through end-to-end learning.

Loiacono, D., Cardamone, L., & Lanzi, P. L. (2016). Simulated car racing championship: Competition software manual. *arXiv, Cornell University*.

Loiacono, D., Prete, A., Lanzi, P. L., & Cardamone, L. (2010). Learning to overtake in torcs using simple reinforcement learning. *IEEE Congress on Evolutionary Computation*.

Loose, H., Franke, U., & Stiller, C. (2009). Kalman particle filter for lane recognition on rural roads. *In Intelligent Vehicles Symposium,IEEE*.

Marshall, A. (2017). Robocars could add $7 trillion to the global economy. *https://www.wired.com/2017/06/impact-of-autonomous-vehicles*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning.

Narote, S. P., Bhujbal, P. N., Narote, A. S., & Dhane, D. M. (2018). A review of recent advances in lane detection and departure warning system.

Paepcke, S. (2016). Robots:husky. *www.osrfoundation.org/michael-aeberhard-bmw-automated-driving-with-ros-at-bmw/*.

Pomerleau, D. A. (1989). Alvinn: an autonomous land vehicle in a neural network.

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2015). You only look once: Unified, real-time object detection. *arXiv, Cornell University*.

ros.org. (2018a). Robots/husky. *http://wiki.ros.org/Robots/Husky*.

ros.org. (2018b). Ros. *http://www.ros.org/news/robots/autonomous-cars/*.

ros.org. (2018c). Ros. *http://www.ros.org/is-ros-for-me*.

Sadeghi, F., & Levine, S. (2016). Cad2rl: Real single-image flight without a single real image. *arXiv, Cornell University*.

Selby, W. (2018). Formulapi ros simulation environment. *https://www.wilselby.com/formulapi-ros-simulation-environment*.

Sprinkle, J. (2017). Cat vehicle. *https://cps-vo.org/group/CATVehicleTestbed*.

TechExplorist. (2018). Increasing the environmental benefits of autonomous vehicles. *https://www.techexplorist.com/increasing-environmental-benefits-autonomous-vehicles/11836/*.

Teng, Z., Kim, J. H., & Kang, D. J. (2010). Real-time lane detection by using multiple cues. *In Control Automation and Systems (ICCAS),International Conference*.

Tovey, A. (2017). Self-driving cars could lend '£8bn boost to uk economy. *https://www.telegraph.co.uk/business/2017/03/30/self-driving-cars-could-lend-8bn-boost-uk-economy/*.

Tully Foote, O. S. R. F. (2017). Simulated car demo using ros kinetic and gazebo 8. *http://robohub.org/simulated-car-demo-using-ros-kinetic-and-gazebo-8*.

wiki.ros.org. (2018). Ros. *http://wiki.ros.org/ROS/Introduction*.

Worland, J. (2016). Self-driving cars could help save the environment—or ruin it. it depends on us. *http://time.com/4476614/self-driving-cars-environment/*.