Lane Keeping Using Reinforcement Learning

Project Report

Ahmed Wael | 201500862

Maha Ezzat | 201400601

Nadine Amr | 201501328

Delivered to:

Dr. Mohamed El-Shenawy

Table of Contents

Abstract: In our project, we aim at tackling one of the most crucial problems of this century; the problem of self-driving cars. We targeted the aspect of lane-keeping together with maximizing the car's speed, and we adopted a Q-learning algorithm. We tried out four different scenarios where we altered the hyper-parameters and the way the reward function is calculated in an attempt to arrive at the best possible model. In this report, we present the exact algorithm used and discuss our results, as well as possible future work.

I. Introduction and Problem Importance

The research and development in the field of self-driving cars have been accelerating non-stop for several decades now, ranging from driving assistants with minimal human intervention to fully autonomous cars. Automakers are doing their very best to join the race of the rapidly advancing technology and enhance the currently used or proposed methodologies or even come up with new ones.

As a result of the infinite number of possible scenarios self-driving cars ought to be capable of dealing with, machine learning is the ultimate tool used in development [1]. Several algorithms are adopted and attempts at improving them are endlessly ongoing [1]. Consequently, this is a very hot research area that has strong and direct implications on the market.

II. Objectives and Problem Formulation

The main basic fundamental tasks of a self-driving car are to be capable of detecting the lane in which it is driving, keep following this lane without adverse deviations, deciding which is more rewarding: to stay in the same lane or to move to a neighbor one, thus being able to maximize its travelling speed, all without colliding with other cars ahead or in neighbor lanes, any other objects or pedestrians, or the roadsides.

The objective of our project is to start tackling the most fundamental of those tasks: lane detecting, lane keeping, and speed maximizing, by means of training a simulated autonomous vehicle using Q-learning, which is a reinforcement learning algorithm. The fact that the whole training and testing processes are to be simulated using a vehicle equipped with the necessary sensors and actuators makes our work as close as possible to the real world; thus facilities duplication on a real hardware and gives more meaning to our results [2].

Consequently, the initial problem at hand can be stated as: training a simulated autonomous vehicle to successfully detect its lane and follow it without crashing to the roadsides while maximizing its speed.

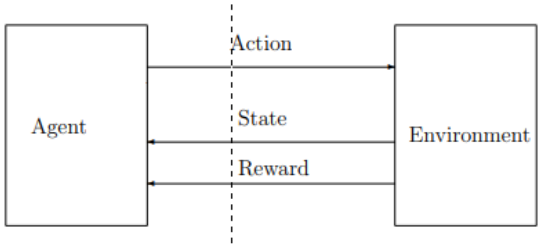
III. Learning Algorithm

Owing to the fact that our agent will have to deal with an unknown environment and learn to take the correct respective decisions, our problem is best considered as a reinforcement learning problem.

1. Reinforcement Learning

Reinforcement learning is a branch of machine learning in which an agent has to

deal with an environment whose dynamics are unknown to it, and is expected to learn the right decisions to take in various scenarios. This is achieved by means of the agent getting guided by a reward which it receives from the environment upon executing different actions in different situations; thus learning to choose to take the actions that maximize its cumulative reward.



2. Q-Learning Algorithm

One of the simplest, most famous reinforcement learning algorithms that saves us a great deal of computational cost is the Q-learning algorithm [3][4]. This is partly due to the fact that it is a **model-free** algorithm, which means that it does not need to use a model of the environment unlike many other model-based reinforcement learning algorithms [3]. Another crucial property of the Q-learning algorithm is that it is an **off-policy** algorithm, which means that the policy our agent uses in choosing the actions it computes in different situations does not necessarily have to be the same policy that it is optimizing [3][4]. In fact, this grants us the opportunity to introduce some **exploration** in our search for the optimal policy [3][4].

The Q-learning algorithm starts off by initializing a **Q-value**, **quality value**, of zero for each possible action associated with all the possible states in a **Q-table**. In any state, $\boldsymbol{s}$, when the agent chooses to take action, $\boldsymbol{a}$, it observes the reward it gets from the environment, $\boldsymbol{R(s,a)}$, as well as its new state, $\boldsymbol{s'}$. These new additional inputs to the agent allow it to calculate a sample estimate of the Q-value, $\boldsymbol{Q^*(s,a)}$, associated with the state it was in before taking the action, and that action it took. This sample estimate is calculated as follows:

$$Q^*(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a')$$

where $\boldsymbol{\gamma}$ is a discount rate which we will refer to in a while. After calculating this sample Q-value, we update our current Q-value associated with $\boldsymbol{s}$ and $\boldsymbol{a}$ by means of calculating a running average of the new sample Q-value and all the previous samples Q-values that the agent dealt with.

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha\, Q^*(s,a)$$

where $\boldsymbol{\alpha}$ is a learning rate which we will refer to in a while.

In our discussion, we considered how to update the Q-table upon taking a certain action when in a certain state. However, how does the agent choose which action to take? The answer is that, since Q-learning is an off-policy algorithm, we can adapt an $\boldsymbol{\epsilon-}$**greedy** algorithm in which, according to the value of epsilon or the exploration rate we set, we can either choose an action randomly, or choose an action which has the greatest Q-value in the Q-table; thus allows for the highest expected reward [3].

3. Hyper-Parameters

As was shown in the Q-learning algorithm section, there are several parameters that affect the learning process and that we ought

to choose carefully. Those parameters are basically the discount rate, γ, the learning rate, α, and the exploration rate, ϵ.

- Discount Rate (γ):
  The discount rate is a number between 0 and 1. It determines how much future rewards are worth compared to immediate rewards. As the value of the discount rate decreases, more weight is given to immediate rewards than to future ones, and vice versa.

- Learning Rate (α):
  The learning rate is a number between 0 and 1. It determines how much newly collected samples during training are worth compared to previous ones, or how much newly collected information overrides old ones. As the value of the learning rate increases, more weight is given to freshly collected samples, and vice versa.

- Exploration Rate (ϵ):
  The exploration rate is a number between 0 and 1. It determines how biased our algorithm is towards exploring different combinations of states and actions, or how often are random actions chosen to be executed during training rather than exploiting only the knowledge we have gained so far in our Q-table. As the value of the exploration rate increases, random actions are chosen more often; thus resulting in a more exploratory algorithm.

IV. Simulated Environment Used (TORCS)

1. Overview

TORCS, or The Open Racing Car Simulator, is a multi-agent 3D car racing simulator that is extremely portable [5][6]. It is an open-source simulation platform written in C++ that runs on Linux (32 and 64 bit, little and big endian), FreeBSD, OpenSolaris, Mac OS X, AmigaOS 4, AROS, MorphOS and Windows (32 and 64 bit) [5][7]. TORCS updates the simulation situation every 2 milliseconds (500 Hz), including updating the different mathematical models governing the physics of the race; such as the motion and positioning of the cars and other objects [6].

TORCS's uses are diverse; ranging from ordinary car racing games, to AI ones where users upload their own developed artificial intelligence robots in order to compete with multiple others worldwide [5][6][7]. Owing to its high portability, modularity, extensibility and the fact that it is open-source, TORCS is also considered one of the most influential research platforms [5][6][7]. Some of these researches include human-assisted algorithmic generation of tracks, the application of several computing techniques to various aspects of robot driving, developing intelligent control systems for different car components and developing complex driving agents capable of finding optimal racing lines and successfully dealing with unexpected driving scenarios by means of taking tactical decisions [6][7].

2. SCR Plugin

In normal cases, cars in TORCS are granted full access of the environment information [3][6]. However, when working with autonomous agents, we want them to have access only to information they gain through their sensors, because this is the type of information they could truly acquire if they were in a real environment [3][6]. As a result, we activate the Simulated Car Racing plugin (SCR-plugin) which allows us to have a server act as a proxy for the environment [3][6]. Consequently, the server sends our client the available sensory input and, in turn, receives the desired output of the actuators from it [3][6].

3. TORCS Sensors

TORCS provides our agent with a great deal of information through the different sensors it supports. Those sensors are:

Sensor	**Definition**
Angle	Angle between the car direction and the direction of the track axis
CurLapTime	Time elapsed during current lap
Damage	Current damage of the car (the higher is the value the higher is the damage)
distFromStartLine	Distance of the car from the start line along the track line
distRaced	Distance covered by the car from the beginning of the race
Fuel	Current fuel level
Gear	Current gear: -1 is reverse 0 is neutral and the gear from 1 to 6
lastLapTime	Time to complete last lap. Opponents: Vector of 36 sensors that detects the opponent distance in meters (range is [0,100]) within a specific 10 degrees sector: each sensor covers 10 degrees, from -π to +π around the car
racePos	Position in the race with respect to other cars
rpm	Number of rotations per

Sensor	Definition
	minute of the car engine
speedX	Speed of the car along the longitudinal axis of the car
speedY	Speed of the car along the transverse axis of the car
track	Vector of 19 range finder sensors: each sensor represents the distance between the track edge and the car. Sensors are oriented every 10 degrees from -π/2 and +π/2 in front of the car. Distances are in meters within a range of 100 meters. When the car is outside of the track (i.e. Track Pos is less than -1 or greater than 1), these values are not reliable!
trackPos	Distance between the car and the track axis. The value is

Sensor	Definition
	normalized w.r.t. the track width: it is 0 when the car is on the axis, -1 when the car is on the left edge of the track and +1 when it is on the right edge of the car. Values greater than 1 or smaller than -1 means that the car is outside of the track
wheelSpinVel	Vector of 4 sensors representing the rotation speed of the wheels

[6].

4. TORCS Control Actions

TORCS allows our agent to take several control actions through the actuators it provides. Those control action are:

Action	Description
Accel	Virtual gas pedal (0 means no gas, 1 full gas)
Brake	Virtual brake pedal (-1 means no brake, 1 full brake)

Action	Description
Gear	Gear value
Steering	Steering value: -1 and +1 means respectively full left and right, that corresponds to an angle of 0.785398 rad
Meta	This is meta-control command: 0 Do nothing, 1 ask competition server to restart the race

[6].

V. Adapted Methodology

In order to adopt the Q-learning algorithm, we need to clearly define our problem's states, actions, how our agent selects between the different possible actions, and the reward function. The code we used can be found attached to this report.

1. States

We chose to represent the state of our agent by means of its speed along its longitudinal axis, the number of the distance sensor with the maximum reading, and the average distance measured by different distance sensors placed at the angles: -40˚, -20˚, 0˚, 20˚ and 40˚.

However, we would then be having an infinite number of possible states. As a result, we discretized the speed and distance values as follows [6]:

- speedList = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150]
- distList = [-1, 0, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 120, 150, 200].

The agent's state was then described by means of 11 bits, such that the first 4 bits represent the binary equivalent of the car's discretized speed, the next 3 bits represent the binary equivalent of the number of the distance sensor with the maximum value, and the last 4 bits represent the binary equivalent of the discretized average distance of the different sensors. Thus, we have a total of 2048 different possible states [6].

2. Actions and Action Selection

The two categories of control actions that our agent can take are to steer and to accelerate. However, since the number of possible actions is infinite, we discretized the action space into 15 different possible ones:

Steer	Accelerate (1)	Neutral (0)	Brake (-1)
0.5(left)	0	1	2
0.1(left)	3	4	5
0	6	7	8
-0.1(right)	9	10	11
-0.5 (right)	12	13	14

[6].

Since we are adopting an $\epsilon - greedy$ policy during training, the action selection process is performed based on the value chosen for the exploration rate, **ϵ**. Another

hyper-parameter named eta was chosen to act as a threshold below which an informed/heuristic action is chosen [6][8].

As a result, the action selection process becomes as follows:
A random number between 0 and 1 is chosen. If it is:

a. below eta → a heuristic action is taken
b. below eta + ϵ → a random action is taken
c. otherwise → the action with the highest Q-value in the Q-table corresponding to the current state is taken

3. Reward Function

The reward given to the agent mainly depends on its position within the track. Secondly, it also depends on the angle between the car's axis and the track axis, as well as the car's speed.

Listed below are the possible scenarios and the corresponding rewards [6]:

a. If the car is stuck → it takes a reward of -2 and the meta control action is set to 1 so that the episode restarts
b. If the car is out of track → it takes a reward of -1.5
c. If the car is neither stuck nor out of track → it takes a reward that is a combination of its position in the track, the angle its axis makes with the track's axis, and its speed.

VI. Results and Discussion

We have trained our agent using four scenarios each with a different combination of values for our hyper-parameters as well as our reward function parameters, in an attempt to arrive at the optimal policy for our problem. A video demonstrating the episodes of the training process of different scenarios can be found at: https://docs.google.com/document/d/1OALkU5eQKnTdfFyD007BSKgrQjmbtnHtr8woxtA_4_A/edit#.

- **Scenario 1**

The model was trained with following hyper-parameters:

- Learning Rate = 0.01
- Epsilon = 0.2
- Discount Rate = 0.99
- Reward:
  - **TrackPos <= 0.75**

Reward=Rspeed+RtrackPos+Rangle

  - **0.75<TrackPos<0.98**

Reward=0.5*(Rspeed+RtrackPos+Rangle)

  - **TrackPos>=0.98**

Reward = -1.5

  - **Stuck**

Reward = -2

Where:

- Rspeed = numpy.power((speed/float(160)),4)***0.05**
- Rtrackpos = numpy.power(1/(float(numpy.abs(trackpos))+1),4) ***0.7**
- Rangle = numpy.power((1/(((float(numpy.abs(angle)) /40)+1)),4)***0.25**

Comments: Agent performance was very poor. Additionally, many episodes were terminated by accident, and it was observed that the agent drifted to the left

most of the times. It also drove with low speeds.

- **Scenario 2**

The model was trained with following hyper-parameters:

- Learning Rate = 0.5
- Epsilon = 0.2
- Discount Rate = 0.9
- Reward:
  - **TrackPos <= 0.75**

Reward=Rspeed+RtrackPos+Rangle

  - **0.75<TrackPos<0.98**

Reward=0.5*(Rspeed+RtrackPos+Rangle)

  - **TrackPos>=0.98**

Reward = -1.5

  - **Stuck**

Reward = -2

Where:

- Rspeed = numpy.power((speed/float(160)),4)***0.05**
- Rtrackpos = numpy.power(1/(float(numpy.abs(trackpos))+1),4) ***0.8**
- Rangle = numpy.power((1/((float(numpy.abs(angle)) /40)+1)),4)***0.15**

Comments: Agent performance was moderate; however it was observed that the agent drifted 180 degrees at many episodes. The agent's speed increased from first scenario.

- **Scenario 3 (Best Scenario)**

The model was trained with following hyper-parameters:

- Learning Rate = 0.1
- Epsilon = 0.2
- Discount Rate = 0.99
- Reward:
  - **TrackPos <= 0.7**

Reward=Rspeed+RtrackPos+Rangle

  - **0.7<TrackPos<0.9**

Reward=0.5*(Rspeed+RtrackPos+Rangle)

  - **0.9<=TrackPos<1.5**

Reward= -abs(trackpos)

  - **TrackPos>=1.5**

Reward = -1.5

  - **Stuck**

Reward = -2

Where:

- Rspeed = numpy.power((speed/float(160)),4)***0.05**
- Rtrackpos = numpy.power(1/(float(numpy.abs(trackpos))+1),4) ***0.8**
- Rangle = numpy.power((1/((float(numpy.abs(angle)) /40)+1)),4)***0.15**

Comments: Agent learned to drive and stabilize itself quite well. It was able to complete 8 laps straight, and its speed was relatively high.

- **Scenario 4**

The model was trained with following hyper-parameters:

- Learning Rate = 0.1
- Epsilon = 0.4
- Discount Rate = 0.99
- Reward:
  - **TrackPos <= 0.5**

Reward=Rspeed+RtrackPos+Rangle

  - **0.5<TrackPos<0.8**

Reward=0.5*(Rspeed+RtrackPos+Rangle)

  - **TrackPos>=0.8**

Reward = -1.5

  - **Stuck**

Reward = -2

Where:

- Rspeed = numpy.power((speed/float(160)),4)***0.3**

- Rtrackpos = numpy.power(1/(float(numpy.abs(trackpos))+1),4) ***1.1**
- Rangle = numpy.power((1/((float(numpy.abs(angle))/40)+1)),4)***0.6**

Comments: The agent's performance and speed were greatly improved relative to scenarios 1 & 2. However, it was observed to spend a considerable amount of time on the edge of the lane.

- **Summary and Comparison**

Measurement Results	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Reference
Training Episodes	1800	1800	1626	1510	1500
Number of Episodes to Complete one lap	----------	232	304	133	1200
Number of Episodes to Build the Model	----------	----------	1626	----------	1500
Maximum Speed of the Car	90 km/h	120 km/h	130 km/h	140 km/h	160 km/h
Time to Complete One Lap	----------	3:02:06	1:05:14	1:23:25	1:24:09

[8].

- **Discussion**

As mentioned in the previous section, four models were developed each with a different combination of values for our hyper-parameters as well as our reward function parameters, in search for the optimal policy for our problem, and to teach the agent to detect its lane, keep tracking it, and maximize its speed. The first two scenarios showed the poorest performance and were not capable of achieving satisfactory results. As for the fourth scenario, although it showed a significant improvement in performance, it was still not able to build a respectable model.

On the other hand, the third scenario was an utter success. The chosen parameters resulted in a stable model in which the agent was able to complete 8 consecutive laps with minimal oscillations. Thus, the third model was the one that achieved the optimal policy, allowing the agent to stably keep its lane.

VII. Future Work

The Q-Learning algorithm adopted in this project discretizes the states and actions in order to limit them to a finite number to be able to represent them in the Q-table. However, a huge amount of information represented in a multifarious number of states and actions is lost in this discretization process. Consequently, we suggest that a continuous model is built instead, in order to enhance the performance demonstrated in our current project. This can be done using deep reinforcement learning algorithms such as DDPG (Deep Deterministic Policy Gradients).

VIII. Conclusion

In conclusion, this problem at hand is one that constantly has the spotlight shed upon as a result of its utter importance and influence on the century's hot areas of research. Extensive research is ongoing nonstop where hundreds of different algorithms are tried out and millions of scenarios are attempted. Hopefully, this project might be our first step towards contributing to this fruitful promising field.

IX. References

[1] A. Davies, P. Martineau, M. Molteni, A. Watercutter, M. Simon, E. Pao and L. Mallonee, "What Is a Self-Driving Car? The Complete WIRED Guide", *WIRED*. [Online]. Available: https://www.wired.com/story/guide-self-driving-cars/. [Accessed: 18- Oct- 2018].

[2] "Learning to drive in a day.", *Wayve*. [Online]. Available: https://wayve.ai/blog/learning-to-drive-in-a-day-with-reinforcement-learning. [Accessed: 18- Oct- 2018].

[3] D. Karavolos, *Q-learning with heuristic exploration in Simulated Car Racing*. Amsgerdam: University of Amsterdam, 2013.

[4] N. Shimkin, *Learning in Complex Systems - Reinforcement Learning – Basic Algorithms*. 2011.

[5] B. Wymann and E. Espié, "torcs › News", *Torcs.sourceforge.net*, 2016. [Online]. Available: http://torcs.sourceforge.net/. [Accessed: 15- Dec- 2018].

[6] A. Raafat, "A-Raafat/Torcs---Reinforcement-Learning-using-Q-Learning", *GitHub*, 2016. [Online]. Available: https://github.com/A-Raafat/Torcs---Reinforcement-Learning-using-Q-Learning. [Accessed: 15- Dec- 2018].

[7] "TORCS", *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/TORCS#Competitions. [Accessed: 15- Dec- 2018].

[8] M. Abdou Tolba, *AUTONOMOUS DRIVING USING DEEP REINFORCEMENT LEARNING*. Cairo: Faculty of Engineering at Cairo University, 2017.