


Tongji University

MAURIZIO VASSALLO
Student ID:
Academic Tutor: a

A Collision Avoidance System Using Reinforcement Learning

Contents

1 Introduction **2**
1.1 Autonomous Vehicles Overview 2

2 Reinforcement Learning **4**
2.1 Introduction 4
2.2 Reinforcement Learning compared to other methods 4
2.3 Application of Reinforcement Learning 5
2.4 Formal definition 6
2.5 Q-Learning 8

3 Project Development **10**
3.1 Environment 10
3.1.1 Simple Track 10
3.1.2 Complex Track 10
3.2 Agent 10
3.3 State 11
3.4 Policy 12
3.4.1 Deep Q-Network 13
3.4.2 Double Deep Q-Network 14
3.4.3 Duelling Double Deep Q-Network 15
3.5 Actions 16
3.6 Reward Fuction 17
3.7 Implementation 19
3.7.1 Client 19
3.7.2 Server 20

4 Results **21**
4.1 Deep Q-network 21
4.2 Double Deep Q-network 22
4.3 Duelling Double Deep Q-network 22
4.4 Comparison between models 23
4.5 Analysis 25

5 Conclusion **26**
5.1 Possible future works 26

6 References **27**

1 Introduction

This report is drawn up after the development of the thesis project at Tongji University (同济大学) for the Double Degree project Politong.

The thesis project aims to develop a machine learning algorithm that allows an autonomous vehicle to learn to drive and to avoid obstacles.
The research will focus on Reinforcement Learning methods.

1.1 Autonomous Vehicles Overview

Nowadays it is possible to hear more and more about Autonomous Driving Vehicles. The autonomous cars (also known as a self-driving cars or a driverless cars) are a vehicle that are capable of sensing their environment and to navigate without human input. The ability of autonomous vehicles to operate without human intervention depends on their level of technological sophistication, in accordance with the current six-degree autonomy scale proposed by the International Society of Automotive Engineers (**SAE**)[2]. There are 6 levels of driving automation, from level 0 (no automation) to level 5 (full automation); intermidiate levels (1 to 3) are considered semi-autonomous[3, 4].
Currently we are between level 2 and 3, so even if the current technology is behind the famous Level 5 of driving automation, there is lot of work to make it happen.

According to research firm, autonomous vehicles will match or exceed human safety by late 2020s and fulfill all mobility needs in 2040s to 2060s. Optimists predict that by 2030, autonomous vehicles will be sufficiently reliable, affordable and common to displace most human driving, providing huge savings and benefits. However, there are good reasons to be skeptical. There is considerable uncertainty concerning autonomous vehicles development, benefits, costs, travel impacts and consumers demand. Considerable progress is needed before autonomous vehicles can operate reliably in mixed urban traffic, heavy rain and snow, unpaved and unmapped roads, and where wireless access is unreliable[1].

The idea behind these cars is quite simple: outfit the vehicles with sensors that can track all the objects nearby and make the cars understand the world around them. Autonomous vehicles are driven using technology such as GPS, odometry (usage of data from motion sensors to estimate change in position over time), radars, laser lights and other devices[4]. These sensors themselves do not make the car 'smart', what make it autonomous are the big computers inside and the algorithms they are running. Usually these softwares run neural networks: these take as input the sensor recordings, elaborate them and output some values like steering angle, accelerate, brake or other important values. Even if the idea

behind these technological innovative vehicles is simple the implementation is not so straightforward, for some reasons: not enough hardware computation, not enough training data, problems with handle different weather conditions (fog, rain, snow, etc.), the current regolation remains in a nascent stage.

Even if this tecnology is not diffused yet, there are many potential benefits that autonomous vehicles could introduce in our society:

- **Transportation Safety**. The most notable predicted benefit of autonomous vehicle technology is a substantial reduction in the human and economic toll of traffic accidents. Indeed, impairment, distractions, and fatigue alone account for over 50% of all fatal crashes. The use of autonomous vehicles could significantly reduce the incidence of such crashes, as vehicles with no-human operators are never drunk, distracted, fatigued, or otherwise susceptible to human failings.

- **Access to Transportation**. Another important potential benefit of autonomous vehicle technology is increased mobility for populations currently unable or not permitted to operate traditional vehicles. These populations include older citizens, disabled, people too young to drive and people without a driver's license.

- **Traffic Congestion and Land Use**. Autonomous vehicles could reduce congestion and change the way in which cities are planned. Most car are moving only for 5% of their lives, for the 95% they are parked[5]! For this reason lot of space is dedicated to parking lots; the same space that could be used for different purposes (green spaces, etc.).

- **Energy and Emissions**. Autonomous vehicle technology has the potential to reduce both energy consumption and pollution thanks to efficiencies gained through smoother acceleration and deceleration and increased roadway capacity.[6]

To do: find a way to link this part with teh following one!!

2 Reinforcement Learning

2.1 Introduction

Humans and animals learn through a process of trial and error. This process is based on a reward mechanism that provide a response to our behaviors. The goal of this process is to incentivize the repetitions of actions which trigger positive rewards and disincentivize the repetition of actions which trigger negative ones. Inspired by how animals and humans learn, **Reinforcement Learning** is built around the idea trial and error from an interaction with the environment

Reinforcement Learning tries to solve the problem in which a decision-maker, **Agent**, can perform some **Actions** inside a world, called **Environment**. The agent senses the environment through the **State** and for each state the agent has to perform an action. These actions result in an effect: they change the agent's state and give it a feedback, the **Reward**. The reward is a value which indicated if the action performed is good, then the reward is positive, or it is bad, the reward is negative. Given these rewards the agent understand what action to perform in a given state to get a positive reward.
This cycle of state-action-reward is repeted many times. The goal of the agent is to find a **policy** such that the actions performed lead to the maximum reward possible. In particual the agent wants to maximize the cumulative reward, the sum of rewards over a period of time. The difference between maximizing the instant reward and cumulative reward is an important distinction because: choosing an action which lead to the maximum next reward could in the next state bring to an end state (game over); instead maximizing the cumulative reward means thinking in a long-term horizon, so that the next states should all have a positive reward (even if the action chosen in one state is not the one that the maximum next reward).

2.2 Reinforcement Learning compared to other methods

Machine learning is the science of getting computers to act without being explicitly programmed. Machine Learning methods are appropriate in application settings where people are unable to provide precise specifications for desired program behavior, but where examples of desired behavior are available, or where it is possible to assign a measure of goodness to examples of behavior[7].
There are mainly 3 methods to train Machine Learning algorithms, each with its advantages and disavantages. The 3 methods are:

- **Supervised Learning**. In this method the algorithm is trained with labeled data; the training examples are of the form (x_i,y_i), with x_i a *n-dimentional* vector and y_i a scalar (it can represent either a class or a

floating value). The learner (either a classifier or a predictor) tries to find a good mapping fuction that maps an input x_i to its correspoding y_i. During this learning process the error between the y_i predicted and the actuall value is calculated and used to make the method learning (usually with Gradient Descent and Back Propagation) and decresing the error over time.
A popular example is to classify and image given the raw pixels.

- **Unsupervised Learning**. The main difference between Unsupervised Learning and Supervised Learning is that in the former method data does not have labels. In this setting, the goal is usually find the relationship between the elements of the dataset. This is done calculating the distance (Euclidian, Hamming, etc.) between the points: if the distance between the points is small, they may share similar characteristics.
A popular example is detecting a person purchase preferences analyzing his shopping list with other people shopping's lists.

- **Reinforcement Learning**. Reinforcement Learning comes into play when examples of desired behavior are not available but where it is possible to score examples of behavior according to some performance criterion[7]. In general, in Reinforcement Learning the goal is to maximized an unknown reward fuction through and trial and error process.
A popular example is a car that learns to drive in a given environment with no prior knowledge given.

Supervised and Reinforcement Learning could be considered similar but there are few key differences, in particual the goal in Supervised Learning is to predict the right label while in Reinforcement Learning the goal is to find an action x^* in other to maximized the reward. For this reason instead of minimizing the error between the predicted class and the real class, the goal is to choose an action that maximize the cumulative reward in a given state.

2.3 Application of Reinforcement Learning

Some possible application of RL:

- **Self-driving cars**. Train a car in a real world can be dangerous (people can be hurt) and expensive (car can hit an object and damage the vehicle). They can be trained in a Reinforcement Learning setting: the car is the agent, the world around it the environment, it has to take some actions (throttle, steer, etc.), the state are the sensors and the goal is to reach a destination avoiding obstacles. In this setting the agent can get positive or negative rewards depending on the action chosen.

- **Games**. Popular example are: chess, Go and Dota. In all these games computers were able to beat the champions.
- **Finance**. Many problems can be formulated as a Reinforcement Learning problem, for example the stock trading. Here the action could be: selling, buying, holding and the goal is to maximize the cumulative return over a period of time.
- many other applications[8].

2.4 Formal definition

Reinforcement Learning can be formulated as a Markov Decision Process (**MDP**), indeed a MDP express the problem of sequential decision-making, where for each state s the dicision maker can choose any action a available in that state s. The process responds moving with some probability to the state s' and giving the decision maker a reward $R_a(s, s')$.
The MDP is defined as a tuple of 4 elements (S, A, P, R), where:

- S is a set of states, called the *state space*.
- A is a set of actions, called the *action space*.
- P is the probability from state s, at time t, of reaching state s', at time $t + 1$ with action a:

$$P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = t)$$

- $R_a(s, s')$ is the immediate reward received after transitioning from state s s to state s', due to action a.

The state and action spaces may be finite or infinite.

The MDP is controlled by a sequence of descrete time steps that create a trajectory υ:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

where the states follow the state transition $P_a(s, s')$. The transition function and the reward function are determined only by the current state, and not from the previous states. This property is called Markov property, which is a characterize the MDP and it means that the process is memory-less and that the future states depends only on the current one and not on its history.

The goal of the MDP is to find a good policy for the decision maker: a function π that specifies the action $\pi(s)$ that will be chosen when in state s. The policy

π found will maximize the cumulative reward over a trajectory υ:

$$G(\upsilon) = \sum_{t=0}^{\infty} R_{a_t}(s_t, s_{t+1})$$

This return value has the problem that all the rewards contribute in the same weight and this can create some problems due to the lack of information. A better return value would be to give some more importance to the short-term memories and giving less importance to the one far in the future. This is solved introducing a **discount factor**, denoted with γ. Then the corrected formula is:

$$G(\upsilon) = \sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})$$

with value of γ satisfying $0 \leq \gamma \leq 1$. A lower discount factor motivates the decision maker to take actions that are close in time instead of actions that a far in the future.

Another important notion in MDP and Reinforcement Learning is the value fuction. While the return $G(\upsilon)$ gives the reward over a trajectory it does not tell much about how good are the single states. The **value fuction** does exactly this, it estimates how good is for the decision maker to be in a given state. The notion of "how good" is defined in terms of future rewards that can the decision maker expect in terms of expected return.
The value fuction $V_\pi(s)$ can be formally defined as:

$$V_\pi(s) = \mathbb{E}_\pi(R_t|s_t = s) = \mathbb{E}(\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})|s_t = s)$$

The expected return when starting at state s and following policy π.

Similary, another notion: the **action-value fuction**, the expected return from state s with an initial action a:

$$Q_\pi(s, a) = \mathbb{E}_\pi(R_t|s_t = s, a_t = a) = \mathbb{E}(\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})|s_t = s, a_t = a)$$

Furthermore, the value fuction and the action-value fuction are called V-fuction and Q-fuction. They are related and satisfy a particular relationship, used in many Reinforcement Learning contests, that for any policy π and state s, the following condition holds:

$$V_\pi(s) = \mathbb{E}_\pi(Q_\pi(s, a))$$

Moreover, the V-fuction can be decomposed in 2 terms:

$$V_\pi(s) = \mathbb{E}_\pi(R_t|s_t = s) = \underbrace{\mathbb{E}_\pi(R_t|s_t = s)}_{\text{immidiate reward}} + \underbrace{\mathbb{E}_\pi(\gamma V_\pi(s_{t+1}|s_t = s))}_{\text{discounted value of next state}} \quad (1)$$

This is the Bellman Equation (1) that defines the value fuction recursively, enabling the estimations of the next states.
Similary it is possible to write Bellman equation for the Q-fuction :

$$Q_\pi(s,a) = \mathbb{E}_\pi(R_t|s_t = s, a_t = a) = \mathbb{E}_\pi[R_t + \gamma Q_\pi(s_{t+1}, a_{t+1})|s_t = s, a_t = a]$$

In this way the V-fuction and the Q-fuction are updated with the values of the successive states without the need to know the trajectory till the end.

2.5 Q-Learning

Q-learning is an off-policy reinforcement learning algorithm that tries to find the best action to take given the current state. It's considered off-policy because the Q-learning function learns from actions that are outside the current policy, like taking random actions and that the updates can be made independently from the policy that has gathered the expirence. In other words, this means that off-policy algorithms can use experiecens collected in the past to improve the policy.
Q-Learning is also a **Temporal Difference** algorithm (TD[9]) and it inherits from the TD learning the characteristics of one-step learning.
These two elements (TD and off-policy property) are important; indeed the goal of Q-learning is to approximate the Q-fuction using the action the maximize the Q-value of the next state. Formally:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \text{max}_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2}$$

where α is the learning rate, that indicated how much the Q-value will be updated, and γ the discount factor, that, as mentioned, gives more importance to the short-term actions and gives less importance to the long-term ones.
This modified version of the Bellman equation updates the value of the current Q-value using only the next step (using the TD one-step learning property) and it is an off-policy algorithm since it uses the max_a to choose a_{t+1}.

Simple Q-learning (also known as Vanilla Q-learning) problems can be solved using a Q-table, where for each state there are all the Q-values for all possible the actions:

States	**Actions**			
	a_1	a_2	...	a_m
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	...	$Q(s_1, a_m)$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	...	$Q(s_2, a_m)$
...	...	...	...	...
s_n	$Q(s_n, a_1)$	$Q(s_n, a_2)$	...	$Q(s_n, a_m)$

Table 1: state-action Q-values

with s_i a *t-dimentional* array (imagine a possible car's state which at each instant record t values from different sensors). During training the Q-values are updated using equation the Bellman equation for Q-learning (2).
This implementation has 2 main problems:

- It is possible to see that the size of the table increases exponentially with the increasing number of states (n) and the number of actions (m), increasing the memory required to store the table.
- The amount of time required to explore each state to create the required Q-table would be increase too.

One way to solve this problem is to use a method that approximate the Q-fuction: one possible candidate is to use neural networks, since they are considered universal approximator fuctions. This new architecture assumes the name of **Deep Q-learning** (DQN). Deep Q-learning replaces the Q-table with a neural network that approximate the Q-value fuction; moreover rather than mapping a state-action pair to a Q-value, a neural netowk maps an input state to an *(action, Q-value)* pair.
This solution overcomes the 2 problems mentioned above.

In this contest, the objective is to change the neural network's weights θ, so that Q_θ is an optimal Q-value fuction given the weights θ.
In order to change the weights, as common with neural networks, the **Gradient Descend** is employed. In particular the weights are changed with the following formula:

$$\theta = \theta - \alpha \nabla_\theta \mathbb{E}_{(s,a,r,s')}[(Q_\theta(s,a) - \text{target})^2]$$

where:

$$\text{target} = r + \gamma \text{max}_{a'} Q_\theta(s', a') \tag{3}$$

is the value the neural network should predict;

$$Q_\theta(s,a)$$

is the value the neural network prediction;

$$\mathbb{E}_{(s,a,r,s')}[(Q_\theta(s,a) - \text{target})^2$$

is the average Mean Square Error over a batch iteration;

$$\nabla_\theta$$

is the gradient of the loss w.r.t the neural network's weights.

In this way over many interaction the weights are updated to reduce the loss and increase the cumulative reward.

3 Project Development

The goal of the project is to use some Reinforcement Learning algorithms to teach an actor to avoid obstacles. The **Agent** is put in an **Environment** it does not know and it will learn a good **Policy** in order to avoid the walls using a trial and error approch. The agent will be given with positive or negative **Rewards** depending on the **Actions** it performs.

3.1 Environment

The are 2 main environments: a simple track and a complex one. Both track where modelled in **Blender** version 2.93[10].
These models are used inside of Unreal Engine 4 [11], a powerful game and physics engine.

3.1.1 Simple Track

This simple track is used for training and validation. Since this is used for training it does not have long straightaways or sharp turns. During training the track is run clockwise and counterclockwise so that the agent learns to turn right and left.

3.1.2 Complex Track

This complex track is used for testing. This will have more long straightaways or sharp turns.

3.2 Agent

The agent is a car available on the Vehicle Variety pack [12] asset of Unreal Engine 4. From this packet only the 'SportCar' content is used and some settings are modified (drag coefficient and other physical values).

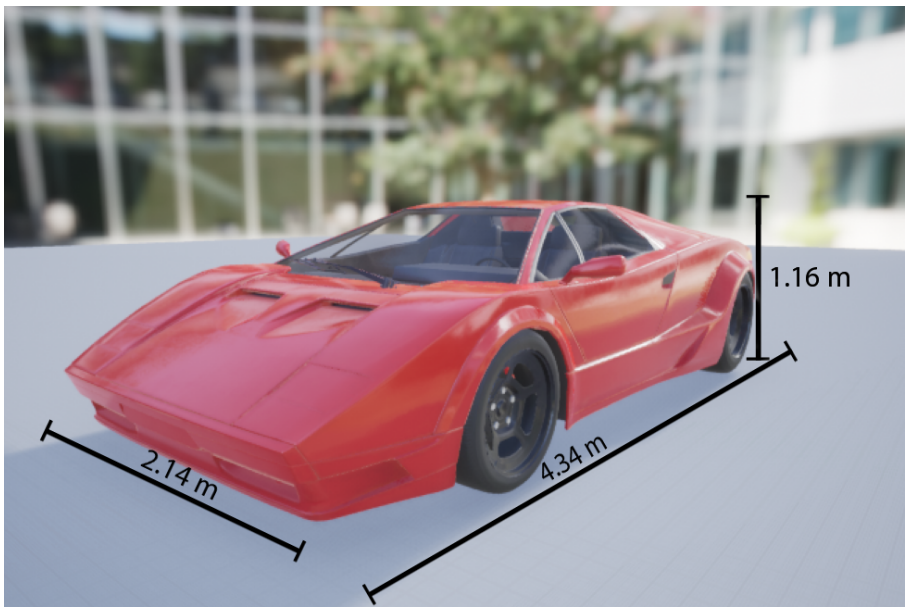


Figure 1: The used car as Agent with its own measurements.

3.3 State

The agent senses the environment around it using some Light Detection and Ranging sensors (**LiDAR** [13]).
The number of LiDAR to use is an important hyper-parameter: in this case good results were found using 32 LiDAR sensors. This 32 sensors are equally spaces in an angle range of 170°, this is another important hyper-parameter. Multiple tests were made to set these hypre-parameters, in particular:

- Too many LiDAR sensors requires more training time, they increase the performance but sometimes the model did not converge.
- With too few LiDAR sensors the algorithm did not have enough information to learn properly.
- Too wide angle decresed the sensor density and decresed the focus on the most important part: the front and the sides of the car.
- Too small angle increased the sensor desity and increased the focus only on some specific region: the front.

So a trade-off between training time and performances were reached with 32 sensors and a 170° angle

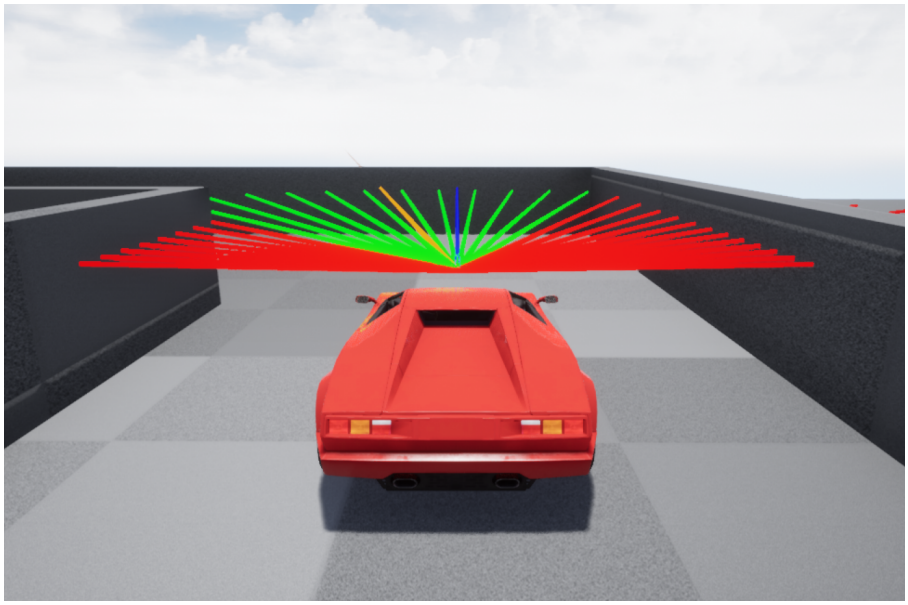


Figure 2: The image shows how the agent senses the world around it. It is possible to see the 32 beams equally spaced on a 170° range.
The beam's color depends wether the ray has hit an obstacle or not: red object is near enough, green no hit.
There are also two other beams (yellow and blue) but they are only for visual puposes, in particular to make humans understand how the agent learns; these will be used in the Reward Fuction 3.6.

3.4 Policy

The policy, as mentioned early, is a mapping from state to action, how the agent chooses an action. The goal is to improve the policy over time in order to increase the cumulative reward.

As common in Reinforcement Learning, the agent needs to explore the environment to get to know the world around it and after some time then apply this knowledge. This is known as the **exploration-exploitation tradeoff**: at the beginng the agent chooses random actions (exploration), after it has learnt it uses what it knows to choose the correct action (exploitation).

This exploration-exploitation tradeoff is implemented with a value ϵ commonly initialized to 1: at each step a random value c (with $c \in [0, 1]$) is drawn, if $c \leq \epsilon$ then the agent chooses a random action and ϵ is decreased otherwise it select the action a with the maximum Q-value given the current state s ($\max_a Q_\theta(s, a)$).
The value of ϵ is decresed with a value $epsilon\_decay = 7.5 \cdot 10^{-5}$; once ϵ keep decreasing the probability that $c \leq \epsilon$ will be lower so preferring exploitation over exploration. This is know as **Epsilon greedy strategy**.

The *epsilon_decay* is an hyper-parameter: the lower *epsilon_decay* is the more the agent will explore the environment increasing the training time.

The performance of the policy depends on the neural network structure. In this project 3 different architecture are used.

3.4.1 Deep Q-Network

As mentioned early Deep Q-networks use a neural network to calculate the Q-values given a state and they use 2 to calculate the next Q-value; in particular the target value is calculated with formula 3, here reported:

$$\text{target}_i = r_i + \gamma \text{max}_{a_i'} Q_\theta(s_i', a_i')$$

and the loss:

$$\mathbb{E}_{(s,a,r,s')}[(Q_\theta(s_i, a_i) - \text{target}_i)^2$$

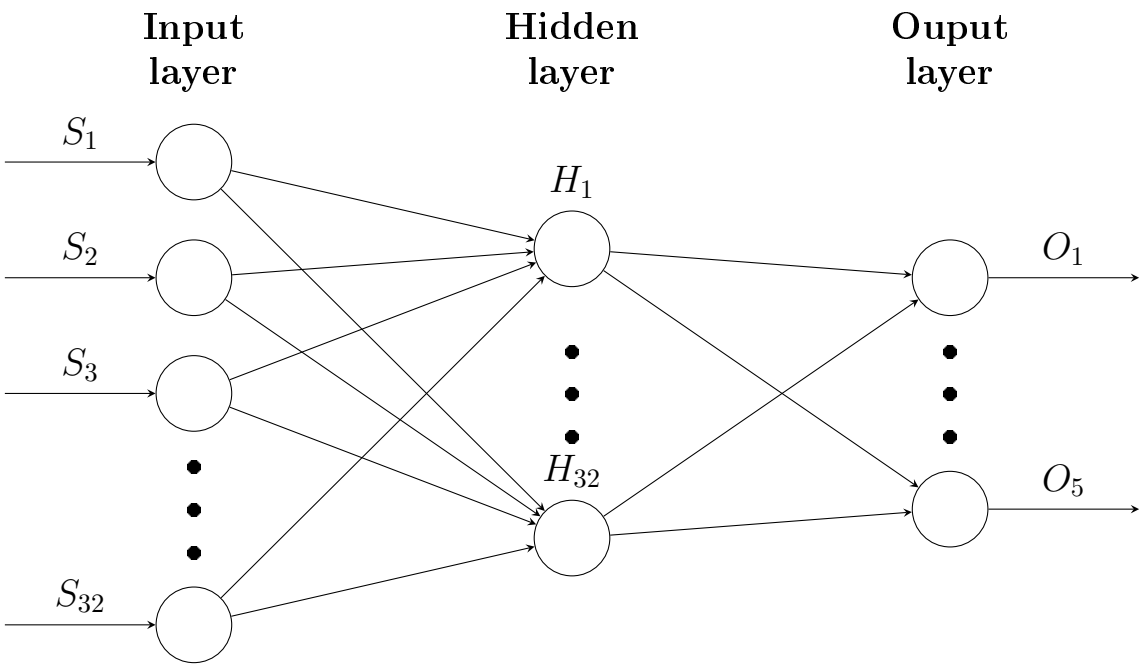


Figure 3: The architecture of the Deep Q-network

One possible problem with this implementation is that both the current Q-value ($Q_\theta(s_i, a_i)$) and the next best Q-value ($\text{max}_{a_i'} Q_\theta(s_i', a_i')$ are calculated using the same network (Q_θ). This can bring to overestimations of the values, resulting in overoptimistic value estimate. (add reference)
One possible solution is to use a Double Deep Q-network.

3.4.2 Double Deep Q-Network

In Double Deep Q-learning 2 networks are used: Q_θ (policy network) and Q_{θ^-} (target network). The 2 networks have the same structure and the weights are initialized with the same values using, in this case, a RandomNormal kernel initializer.
In this case target and loss are calculated as follow:

$$\text{target}_i = r_i + \gamma \text{max}_{a'_i} Q_{\theta^-}(s'_i, a'_i)$$

$$\mathbb{E}_{(s,a,r,s')}[(Q_\theta(s_i, a_i) - \text{target}_i)^2$$

With this solution current Q-value ($Q_\theta(s_i, a_i)$) and the next best Q-value ($\text{max}_{a'_i} Q_{\theta^-}(s'_i, a'_i)$ are calculated using the 2 different networks (Q_θ and Q_{θ^-}). This reduces the bias and increases the score.
Over time the loss between Q_θ and Q_{θ^-} is minimized. The weights of network Q_θ are updated every step using the gradient descent formula while the target network's weights (Q_{θ^-}) are update every N steps using **Polyak averaging**:

$$\theta^- = \tau\theta + (1 - \tau)\theta^- \tag{4}$$

This is a soft averaging (an hard averaging would be $\theta^- = \theta$) weighted by the value τ (tau). τ represents how much information it has to be kept from the policy network ($\tau = 1$ only the information of the policy network, $\tau = 0$ only the information of the target network is keep, no update in this case) [14, 15]. The value of τ is usually quite small so that the target network will move slightly to the value of Q-network. Since the value of *tau* is small, the update should be frequent so that the effect will be noticeable.

Both N and τ are hyper-parameters, in this project they are $N = 64$ and $\tau = 0.001$

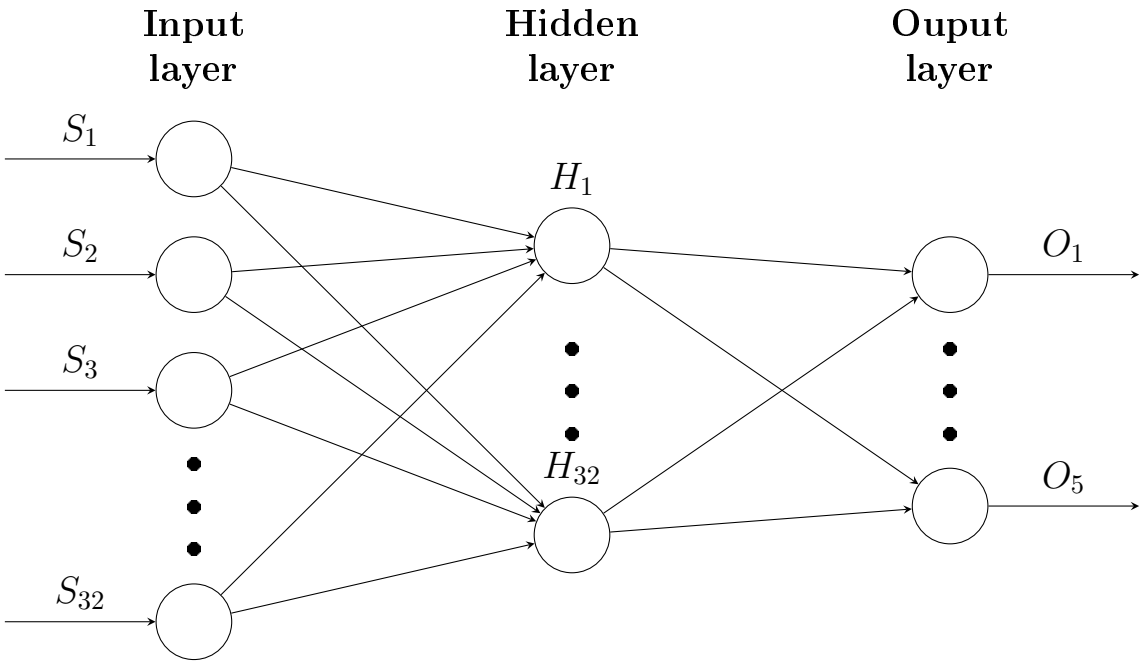


Figure 4: The architecture of the Double Deep Q-network, same as the Deep Q-network. The target network architecture is omitted since identical to this architecture.

3.4.3 Duelling Double Deep Q-Network

The Q-value Q(s,a) tells us how good it is to take an action a being at state s. This Q-value can be decomposed as the sum of $V(s)$, the value of being at that state, and $A(s, a)$, the advantage of taking that action at the state (from all other possible actions). Mathematically, we can write this as:

$$Q(s, a) = V(s) + A(s, a)$$

Dueling Deep Q-network uses two separate estimators for these two components which are then combined together through a special aggregation layer to get an estimate of Q(s,a). By decoupling the estimation, intuitively the Dueling DQN can learn which states are (or are not) valuable without having to learn the effect of each action at each state. This is particularly useful for states where actions do not affect the environment in a meaningful way. [15]
The key insight behind this architecture is that for many states, it is unnecessary to estimate the value of each action choice. For example, in the Enduro game setting [16], knowing whether to move left or right only matters when a collision is imminent. In some states, it is of paramount importance to know which action to take, but in many other states the choice of action has no repercussion on what happens. [17].

With this implementation, the Q-value estimate can be obtained by aggregation:

$$Q(s,a) = V_\beta(s) + A_\alpha(s,a) - \frac{1}{|A|}\sum_{a'} A_\alpha(s,a') \tag{5}$$

where β and α are the weights for the $V(s)$ and $A(s,a)$ networks respectively. Training of the dueling architectures, as with standard DQ-networks, requires only back-propagation. The estimates $V(s)$ and $A(s,a)$ are computed automatically without any extra supervision or algorithmic modifications. In particular this network share the same N and τ as the Double Deep Q-network 3.4.2

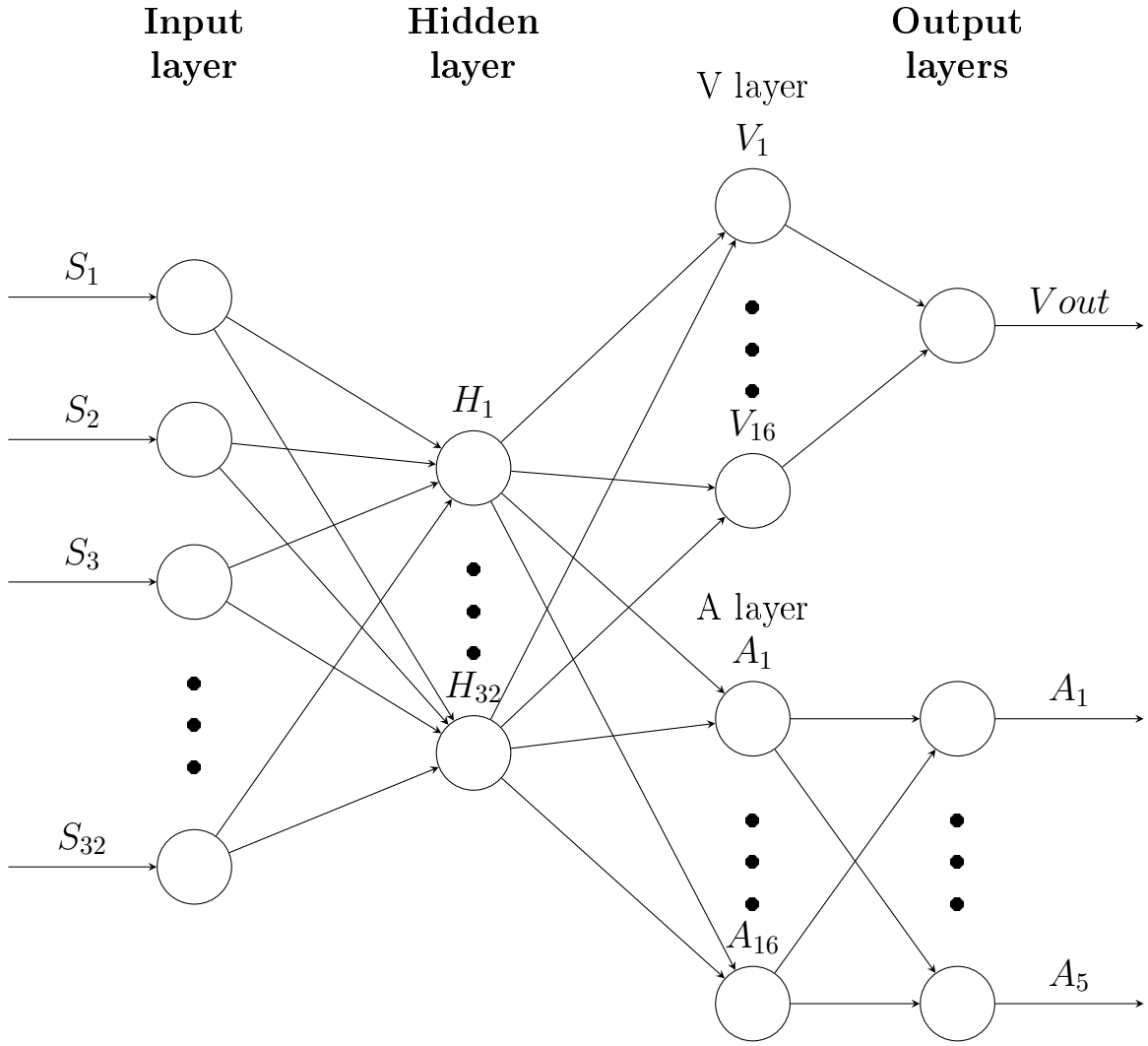


Figure 5: The architecture of the Deep Q-network. The outputs (V_{out} and A_i) are then used in formula 5.

3.5 Actions

There are 2 possible actions the agent is allowed to perform: throttle and steer. These are descretize into 5 actions, from a_0 to a_4:

- a_0, steer to the left:

```
SetThrottleInput(0.3);
SetSteeringInput(-0.9);
```

- a_1, slightly steer to the left:

```
SetThrottleInput(0.6);
SetSteeringInput(-0.5);
```

- a_2, go fully forward:

```
SetThrottleInput(0.9);
```

- a_3, slightly steering to the right:

```
SetThrottleInput(0.6);
SetSteeringInput(0.5);
```

- a_4, steer to the right:

```
SetThrottleInput(0.3);
SetSteeringInput(0.9);
```

The values inside the parenthesis are scales to the maximum values ($i.e$ SetThrottleInput(-0.9) means that the steering angle il $45 * (-0.9) = -40.5°$ angle, turn left with the wheels' angle of 40.5°) and they are hyper-parameters.

3.6 Reward Fuction

Reward function is an important fuction that tells the agent what is correct and what is wrong using rewards and punishments.
Example of reward fuction can be found in the letterature in similar tasks:

- [18] proposes a method where the agent's position and orientation are compared some ideals values. The desired position, is inversely proportional to the distance from the middle of the lane. This reward is maximum equal to 0 when agent is exactly in the middle of the lane and goes to -1 when reaching a maximum distance from lane. The desired rotation is inversely proportional to the difference in angle between the agent and the optimal orientation of the trajectory.

- In a similar way, [19] proposes a reward fuction:

$$r = vcos(\theta)$$

  where the reward r is calculated using the speed v of the car and its angle θ with respect to the road.

- other methods employ some checkpoints along the track. Usually this checkpoint are placed manually bu humans.

These methods need some external information, for example where the middle of the road is, where is the next checkpoint and so on.

The proposed reward fuction in this project tries only to get information from the car sensors.
To explain the reward fuction an angle θ must be introduced. This angle is the between the blue ray and orange ray, as shown in figure 3.6, where the orange ray is the vectorial sum of all green rays and the blue ray is the car's forward direction.

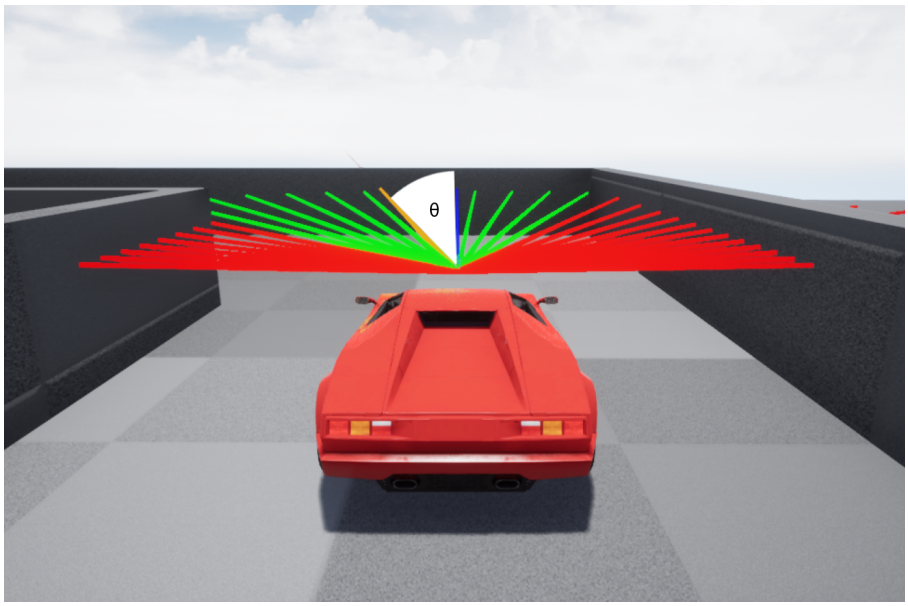


Figure 6: The used car as Agent with its own measurements.

Given the angle θ the reward fuction is defined as follow:

```
if (   (θ > 0 && chosen_action > 2) ||
       (θ < 0 && chosen_action < 2) ||
       (math.abs(θ) < forward_angle && chosen_action == 2))
{
  angle_reward = (number_green_rays - ray_encou_factor) / ray_scal_factor;
}
speed_reward = (agent_speed - speed_encou_factor) / speed_scal_factor;

total_reward = angle_reward + speed_reward;
```

Where *chosen_action* is the action choosen either by exploration or eploitation, *forward_angle* is an hyper-parameter (on it depends when the agent should go forward with the max acceleration), *number_green_rays* is the number of green rays (short reminder: the agent can sense the world with rays, these can be visualized as green and red 3.3), *agent_speed* is the magnitude of agent's velocity vector, *ray_encou_factor* and *speed_encou_factor* are

hyper-parametes and they are encouraging factors, respectively set to 1 and 250 (they push the agent to perform better: for example, the factor$speed_encou_factor$ on the emphspeed_reward means that the agent has to have a speed greater than 250 otherwise the reward is negative; similarly for the emphangle_reward), ray_scal_factor and $speed\_scal\_factor$, set to 10 and 1000, are hyper-parametes and they are scaling factors so that the reward is not too large. Finally the $total\_reward$ is given by the sum of the 2 rewards.
If the car hits a wall the reward is -5.

3.7 Implementation

The whole system can be divided in 2 parts: the environment part and the machine learning part. The environment part in developed in Unreal Engine and it includes the environment itself and the physical agent; from this part the agent senses the world, performs actions and get the rewards. The machine learning part is developed in Python and it is where the neural networks run (the policies, the brain of the agent). The 2 parts communicate through HTTP requests (Unreal Engine is the client and Python is the server).
The main reason why there are two part and that they communicate throught HTTP requests is that: using Python tools (Tensorflow, etc.) is convinient when dealing with machine learning and write everything in C++ would be time consuming and error proning; Unreal Engine does not allow native communication with Python scripts.

3.7.1 Client

The client starts creating the agent and sending some meta-data (the model to be trained: DQN, DDQN DDDQN, etc.) to the server to initialize it. After the initialiazation, the run can start with the first episode. An episode has a fixed length of 2000 steps (Unreal Engine updates) but it can end before reaching the 2000 steps if the agent hits an obstacle. An episode start with the positioning of the agent around a point with an offset of the x and y cordinates (this is done to avoid the agent to start always from start same exact point) and with a rotation of 0° degrees or 180° degrees so that the agent can circle the track clock or anti-clock wise. For each step after the next one:

1. the agent senses the world with LiDAR sensors and obtains the current state s: an array of 32 values that indicates if for each ray there is an obstacle or not

2. the agent chooses an action a, either random or from the neural network. In the former case the epsilon value ϵ is decresed, with the formula:

$$epsilon = epsilon - epsilon\_decay$$

In the latter case an HTTP request with the current state is send to the server that will reply with the best action to perform

3. the agent perform the action chosen gets a reward r, senses the next state s' and evaluate if the next state is an terminal one or not (the agent hits the wall is an terminal state) t, either 0 or 1

4. the information of this step is then send to the server the experience (s, a, r, s', t)

5. the agent is repositionating if the terminal state t is 1 or the current step number is greater than 2000, otherwise the step number is incresed and the loops repeat with point 1.

3.7.2 Server

Once the server variables are initialized the server starts waiting for the client's requests.
There are 2 main fuctions: fit and predict.

- Fit receives an experience as a string from the client and stores it in a list of experiences, query this list to obtain a batch of experiences with which the neural network is trained: the experiences pass through the network, the best next value is calculated with the neural network (either the policy or target network depending on the policy used), the target value is calculated with the Bellman equation. Using the the target and the best next value the loss is calculated and then back-propagate through the network to change weights.
  In the case of DDQN and DDDQN the weights of the target network are updated every N steps usign formula 4.

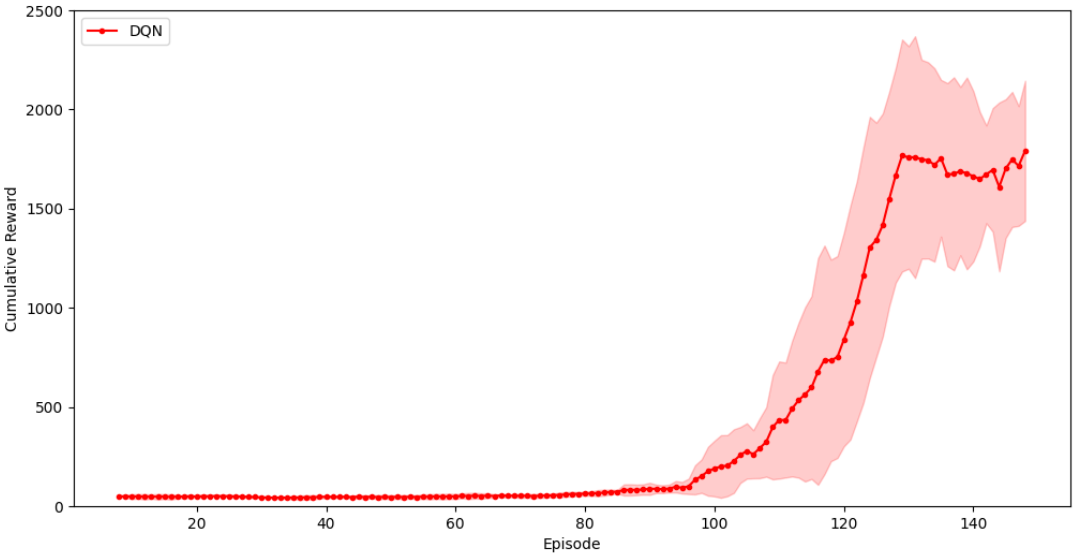
- Predict receives a state as string, this is converted to a variable and passed through the network chosen. The prediction is the action which has the the maximum Q-value and this is send back to the client.

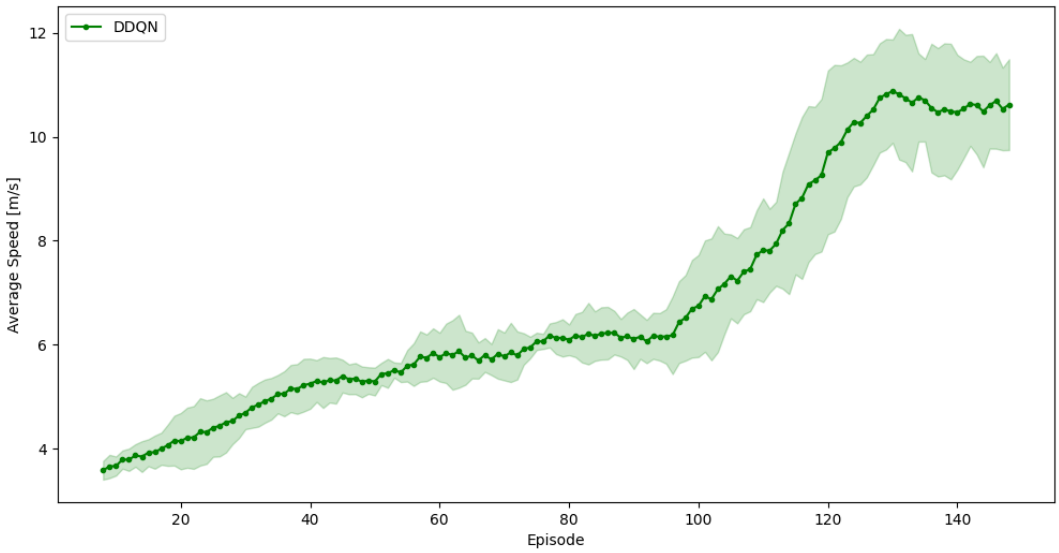
4 Results

The result shown for cumulative reward and the average speed of the 3 models are obtained from 5 different runs: the mean and the confidence intervals are calucalted using the t-distribution ([20]) with a confidence level of 0.95.
A single run is 150 episodes long and it takes about 50 to 70 minutes.

4.1 Deep Q-network

The result obtained with the Deep Q-network:





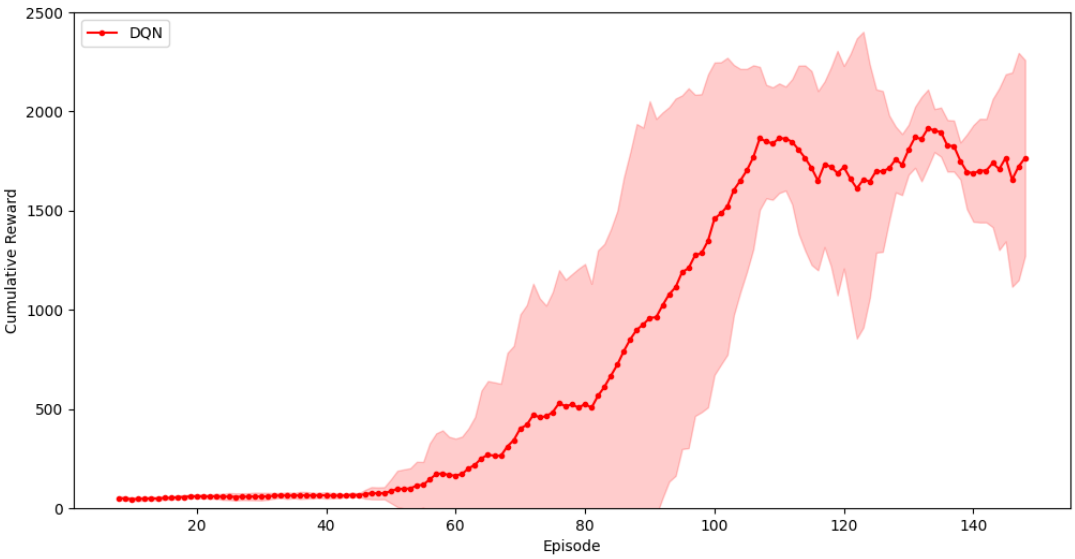
It is possible to see that the model leant to drive succesfully with not too differences between the runs (the confidence intervals are quite small).

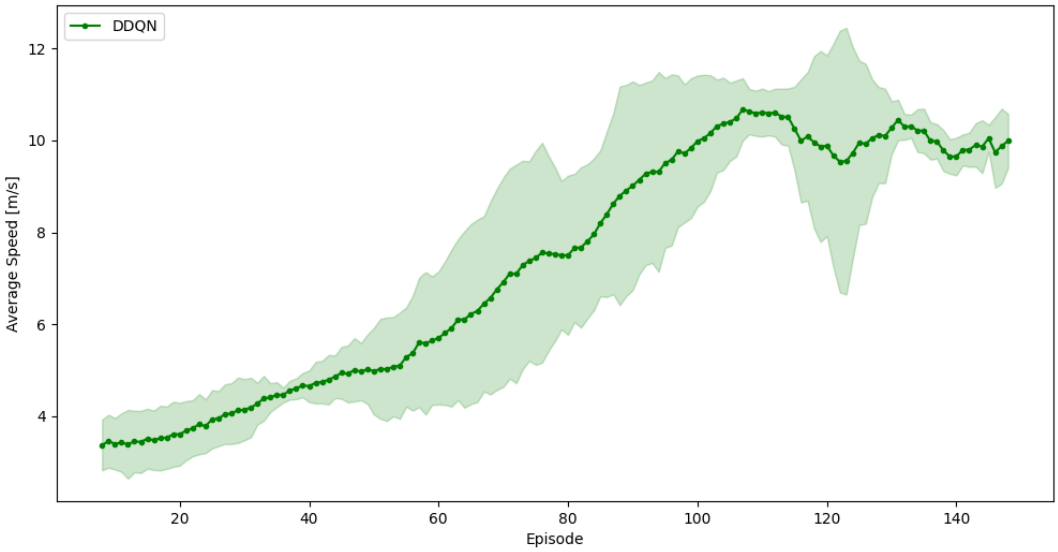
The plot curve are smoothered using an averagin moving window of length 9,

this to have more meaningfull plot since the focus should be on the trend instead of each single run.

4.2 Double Deep Q-network

The result obtained with the Double Deep Q-network:

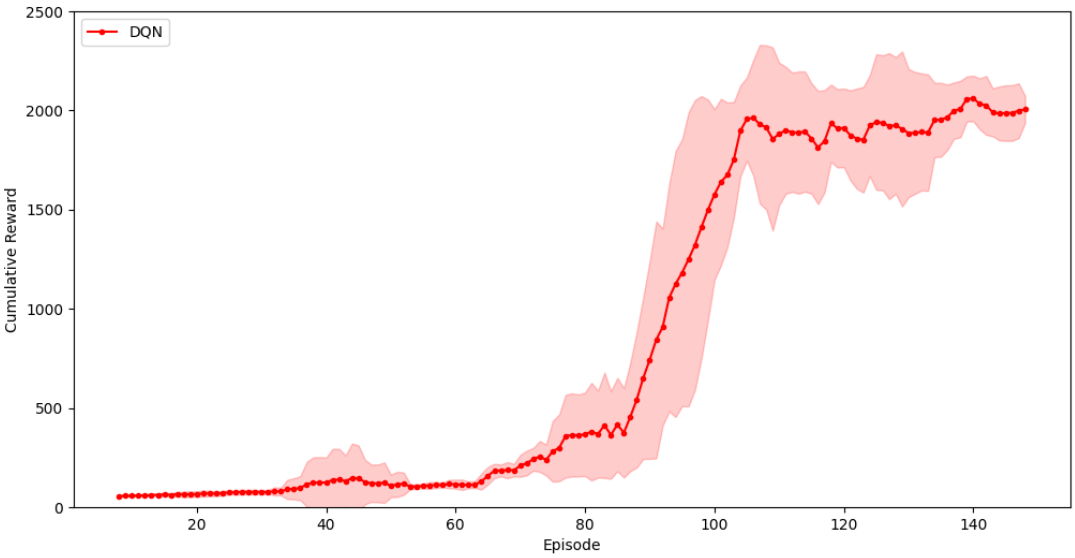


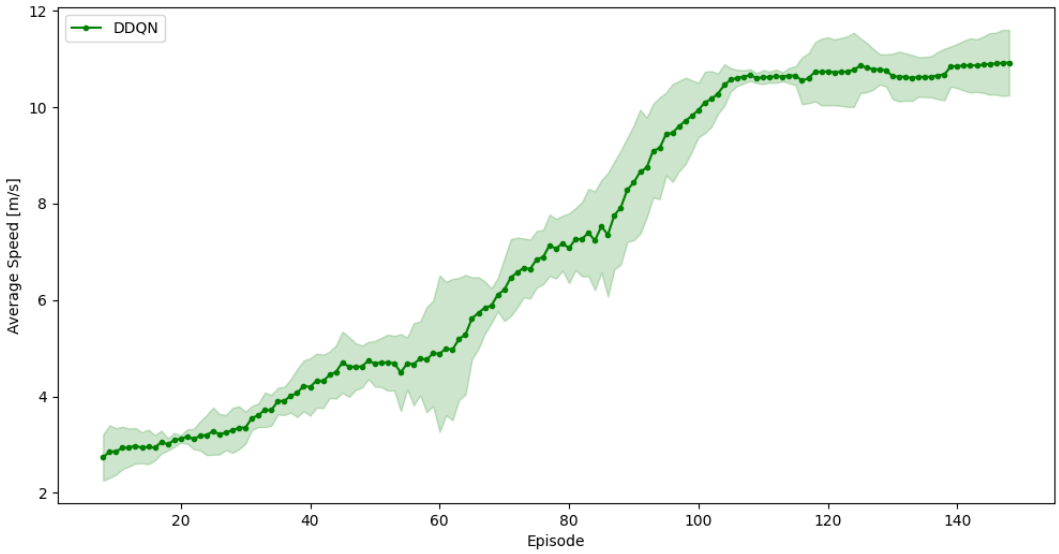


It is possible to see that the model leant to drive succesfully with some differences between the runs (the confidence intervals are large).

4.3 Duelling Double Deep Q-network

The result obtained with the Duelling Double Deep Q-network:

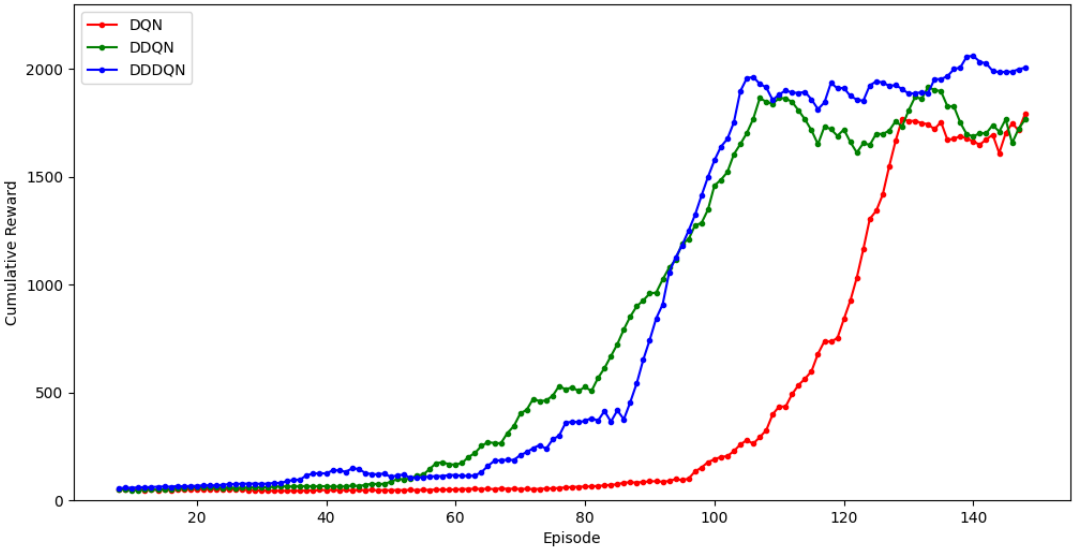


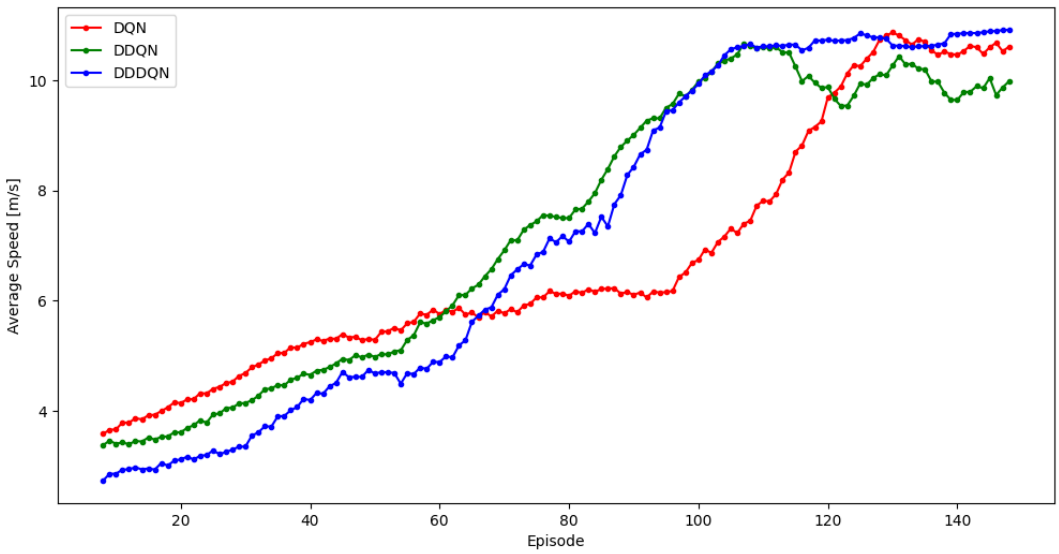


It is possible to see that the model leant to drive succesfully in a stable way (the confidence intervals are small).

4.4 Comparison between models

The comparison between the different models:





From these plots it is posible to see some differences between the leanrning paths of the models. In particular:

- The Double Q-network takes longer too start getting high reward and its peak is less than 2000.
- The Double Deep Q-network is fast getting high rewards but it stops at around 2000.
- The Duelling Double Deep Q-network is slower than the DDQN but it reaches a reward greater than 2000.

The confidence intervals on these plots are removed to have visually simpler graphs.

Model	Max Score	Improvement[1]
DQN	1791.5	/
DDQN	1915.1	+6.90%
DDDQN	2060.3	+7.58%

4.5 Analysis

It is possible to see that the model Duelling Double Deep Q-network performs better then the other models this is expected since it is a improvement of the previous models.

In particular, the advantage of the dueling architecture lies partly in its ability to learn the state-value function efficiently. Indeed with every update of the Q values in the dueling architecture, the value stream V is updated; this contrasts with the updates in a single-stream architecture where only the value for one of the actions is updated, the values for all other actions remain untouched. This more frequent updating of the value stream in our approach allocates more resources to V , and thus allows for better approximation of the state values, which in turn need to be accurate for temporal difference-based methods like Q-learning to work. This phenomenon is reflected in the experiments, where the advantage of the dueling architecture over single-stream Q networks grows when the number of actions is large [17].

An interesting fact is that, the training phase can be divided in 5 periods:

1. At the beginning the agent does not know anything about the environment and move randomly, usually hitting a wall in the first steps

2. after some episodes, the agent start to learn what action to perform in the different states but it is not confident so the speed is relatively low. It hits a wall after some time, usually because the turn are taken too sharply

3. then the agent, improves and can confidently finish some laps. The speed increase and the turns are performed with some distance from the walls. Still the agent is not able to reach the 2000 steps.

4. the agent is very good, reach the maximum speed and it is able to reach the 2000 steps without hitting a wall. It worth to notice that in this period the agent takes the turns sharp to increase the speed (this is was a bad behivior in period 2 but the agent understands that is the better way to drive to increase the cumulative reward).

[1]Improvement with respect to the previous model. These are calculated with the following formula: $increment = (previous\_model\_max - current\_model\_max)/previous\_model\_max * 100$

Results show interesting zero-shot transferability [21] of a trained agent in the context of simple track and generalize the knowledge to the complex track. Indeed the simple track is used only for training purposed and the complex track is used to testing purposes (the agent is not trained on the complex track).

5 Conclusion

This project was a formative experience that allowed me to discover many new topics. Moreover, it allowed me to understand that there is a substantial difference between the university environment and the research environment: university, as mentioned by some professors during the courses, has the task of giving a general knowledge of the topics and the real task is to give a way of thinking and analysing a problem to find a solution; indeed, during this project I realized that computer related knowledge was not always essential but it was more important to understand what was needed to be done and how to do it.

I am fully satisfied with the project as AI researcher and developer and I believe this project was very useful because it allowed me to deepen and learn about new topics and gave me an idea to understand if the tasks assigned could be addressed in a future university career or in a possible job.
Moreover this was an experience that allows to grow both personally and professionally, not to mention that it is a way to enrich the Curriculum Vitae.

5.1 Possible future works

To conclude this report, here some possible future ideas that could be addressed in a next experience:

- find better hyper-parameters, maybe running a grid search even if it would be expensive;
- perform more runs so that the confidence intervals are smaller;
- set up a actuall robot and then transfer the trained models to the real word and test their performances (check whether the result are the same and model X it is still the best).
- to use camera instead of LiDARs to get the agent's state. Similar projects: [22, 23]
- Test other Reinforcement Learning models. For example an interesting model would be the Deep Deterministic Policy Gradient (**DDPG**) to solve the same task but with a contiuous action space [24].

6 References

[1] Todd Litman. 2021. *Autonomous Vehicle Implementation Predictions.*
https://www.vtpi.org/avip.pdf

[2] SAE. 2016. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles.*
https://www.sae.org/standards/content/j3016_201806/

[3] Monika Stoma, Agnieszka Dudziak, Jacek Caban and Paweł Drozdzie. 2021. *The Future of Autonomous Vehicles in the Opinion of Automotive Market Users.*
https://www.mdpi.com/1996-1073/14/16/4777/pdf

[4] Piotr CZECH, Katarzyna TUROŃ, Jacek BARCIK. 2018. *AUTONOMOUS VEHICLES: BASIC ISSUES.*
https://doi.org/10.20858/sjsutst.2018.100.2

[5] Paul Barter. 2013. *"Cars are parked 95% of the time". Let's check!.*
https://www.reinventingparking.org/2013/02/cars-are-parked-95-of-time-lets-check.html

[6] Jeremy A. Carp. 2018. *AUTONOMOUS VEHICLES: PROBLEMS AND PRINCIPLES FOR FUTURE REGULATION.*
https://scholarship.law.upenn.edu/cgi/viewcontent.cgi?article=1048&context=jlpa

[7] ANDREW G. BARTO and THOMAS G. DIETTERICH. *Reinforcement Learning and its Relationship to Supervised Learning.*
http://web.engr.orst.edu/~tgd/publications/Barto-Dietterich-03.pdf

[8] Yuxi Li. 2019. *REINFORCEMENT LEARNING APPLICATIONS.*
https://arxiv.org/pdf/1908.06973.pdf

[9] Scholarpedia.org *Temporal difference learning.*
http://www.scholarpedia.org/article/Temporal_difference_learning

[10] *Blender.*
https://www.blender.org/

[11] *Unreal Engine 4.*
https://www.unrealengine.com/

[12] *Vehicle Variety Pack.*
https://www.unrealengine.com/marketplace/en-US/product/bbcb90a03f844edbb20c8b89ee16ea32

[13] *LiDAR.*
https://en.wikipedia.org/wiki/LiDAR

[14] Hado van Hasselt and Arthur Guez and David SilverGoogle DeepMind. 2015. *Deep Reinforcement Learning with Double Q-learning.*
https://arxiv.org/pdf/1509.06461.pdf

[15] Swagat Kumar. 2020. *Balancing a CartPole System with Reinforcement Learning.*
https://arxiv.org/pdf/2006.04938v2.pdf

[16] Atari 2600. 1983. *Enduro (video game).*
https://en.wikipedia.org/wiki/Enduro_(video_game)

[17] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas. Google DeepMind, London, UK. 2015. *Dueling Network Architectures for Deep Reinforcement Learning.*
https://en.wikipedia.org/wiki/Enduro_(video_game)

[18] Marin Toromanoff, Emilie Wirbel, Fabien Moutarde. 2020. *End-to-End Model-Free Reinforcement Learning for Urban Driving using Implicit Affordances.*
https://openaccess.thecvf.com/content_CVPR_2020/papers/Toromanoff_End-to-End_Model-Free_Reinforcement_Learning_for_Urban_Driving_Using_Implicit_Affordances_CVPR_2020_paper.pdf

[19] Rohan Chopra and Sanjiban Sekhar Roy. 2020. *End-to-End Reinforcement Learning for Self-driving Car.*
http://eprints.kmu.ac.ir/30207/8/SPRINGER%20book-compressed.pdf#page=65

[20] Wikipedia. Student's t-distribution. *Student's t-distribution.*
https://en.wikipedia.org/wiki/Student%27s_t-distribution

[21] Daniele Gammelli, Kaidi Yang, James Harrison, Filipe Rodrigues, Francisco C. Pereira, Marco Pavone. 2021. *Graph Neural Network Reinforcement Learning for Autonomous Mobility-on-Demand Systems.*
https://arxiv.org/pdf/2104.11434.pdf

[22] Riku Arakawa, Shintaro Shiba. 2020. *Exploration of Reinforcement Learning for Event Camera using Car-like Robots.*
https://arxiv.org/pdf/2004.00801.pdf

[23] Peide Cai, Hengli Wang, Huaiyang Huang, Yuxuan Liu, and Ming Liu. 2021. *Vision-Based Autonomous Car Racing Using Deep Imitative Reinforcement Learning.*
https://arxiv.org/pdf/2107.08325.pdf

[24] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver and Daan Wierstra. Google Deepmind, 2019. *CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING.*
https://arxiv.org/pdf/1509.02971.pdf

[25] Shumin Feng, Bijo Sebastian and Pinhas Ben-Tzvi. 2021. *A Collision Avoidance Method Based on Deep Reinforcement Learning.*
https://www.mdpi.com/2218-6581/10/2/73/pdf