Reinforcement Learning algorithms applied to Self-Driving Racing Car

David Forte da Ressurreição and **Fábio Miguel Rodrigues Vaqueiro** and **Pedro Duarte Santos Henriques**
Faculdade de Ciências e Tecnologia da Universidade de Coimbra
{ressurreicao, fabiovaqueiro, pedrohenriques}@student.dei.uc.pt,

Abstract

This paper is studying how different kinds of agents of Reinforcement Learning will interact with an agent to achieve in a low quantity of steps, a Self-Driving car. The main objective is to study how different algorithms, like TD3 and DQN, will perform in a learning situation in order to find out the better one to be used as a baseline.

In order to achieve that goal, a racing simulator, TORCS was used with Python and some frameworks like Stable-Baselines3 and Open-AI Gym. Then 3 algorithms were tested and their results were analysed and compared.

Even though we didn't get a well-performing agent, we were capable of proving a reference to some errors done during the development of the agents, which could help someone who is trying to do related research to know where and how to begin.

1 Introduction

Artificial Intelligence(AI) is currently one of the most researched themes in the context of Computer Science, even though it has a lot of past research [Haenlein and Kaplan, 2019]. In that field, Reinforcement Learning is one of the most studied areas being used in a lot of applications like Industry automation, healthcare and Autonomous Driving, on the other hand, motorsport is one of the fastest-growing competitive sporting events worldwide, for example, Formula 1 (F1), the most watched race of the 2020 season, the Hungarian Grand Prix in Budapest had an incredible 110 million spectators.

In that context, our project goal is to help the racing community become even faster, aiming to apply an AI to Autonomous Driving, knowing that Driving requires a lot of skill, and environmental awareness and is, to some extent, a dangerous operation, since the drivers risk their lives trying to achieve the same goal as us.

The most common approach to this problem is through a discrete algorithm, and, even though some implementations use others, it isn't clear if car performance will or not benefit from their use.

In this paper we are going to study how different RI algorithms can interfere with the performance of an automated vehicle, in the racing context, measuring the order of magnitude of the changes. To achieve that, we are going to use the simulator "The Open Racing Car Simulator" (TORCS) change and try to improve our reward function, comparing deeply discretized and continuous types of algorithms, by measuring track and agent training performance variables, like best lap times, median speed around track and number of episodes to reach a state that can complete a certain number of laps without getting out of track. We are going to start analyzing the performance of a Discrete algorithm and then compare it with others like Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO). By doing this, we want to help future researchers in this context to have a baseline to follow and possibly improve their studies with more advanced baseline data.

2 Related Work

In this section, we will briefly aboard history and concepts of Artificial intelligence, Reinforcement Learning and Autonomous driving, talking about Deep Learning, Machine Learning and Neural networks.

2.1 Artificial Intelligence

Artificial Intelligence is a very wide area, with a very hard definition. John McCarthy considered the father of AI, proposed, with other 3 scientists, a project to research[McCarthy *et al.*, 1955] *how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves*, defining themes like natural language processing, neural networks, theory of computation and creativity.

A lot of research is being done since then, nowadays, you will probably observe AI everywhere. If you have a smartphone or a car, recommendation systems built-in content provider apps like Spotify, Youtube and Facebook, and technologies like Lane Assist, were probably developed with AI. If you live in a big city, some everyday interactions like traffic lights can have some AI processing. And it is not only in the everyday interaction context that AI is important, some of the objects we have today, wouldn't be the same without it.

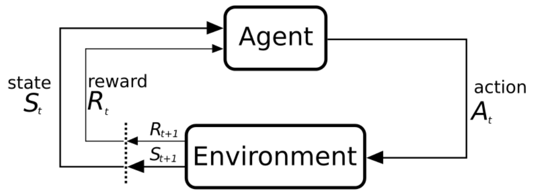

Figure 1: Markov Decision Process Illustration

We can say with some certainty that, the world would never be what it is today without AI.

2.2 Reinforcement Learning

We cannot talk about AI without talking about the most famous subset, Reinforcement Learning.

Reinforcement Learning can be understood as the machine learning implementation of a trial-and-error methodology, similar to how a human baby learns, for example. In order to apply RI, the problem should be structured as a Markov Decision Process.

Markov Decision Process(MDP)

MDP structure, applied in RI, has 5 components that work together in a defined way, as seen in Figure 1[1].

Taking by example a Checkers game:

- **Agent:** It is the operation system, and what would be controlled by RI, the player.
- **Environment:** It is where the agent interacts with the world (real or simulated), the board.
- **State:** Represents the state of the environment at the current time, the position of each piece at a given time.
- **Action:** Represents the action done by the agent in the environment, in the example would be the movement of a piece in the board
- **Reward:** The reinforcement that the agent receives from results, being a consequence of his actions. It is an evaluation of action performance. An example can be the probability of winning, at a given time.

As referred to in the Introduction, section 1, RI has been applied to a lot of fields.

Nowadays we observe RI in marketing, helping companies to know their audience and consequently create useful user products and increase their ROI (Return on Investment)[Jin *et al.*, 2018].

In Data analysis, where an agent can be substantially better at analysing and processing a set of data when compared to a human.

RI is also being used to study and test the possibility of autonomous driving.

[1]https://en.wikipedia.org/wiki/Deep_reinforcement_learning

2.3 Autonomous Driving

As already referred to, autonomous driving is a field being researched by both academics and the industry.

Recently, we can see technology-directed car manufacturers, like Tesla achieving great progress in this, and traditional automakers are also joining the race, for example, Audi, was the first to achieve level 3 in 5 on the SAE International Scale[Wang *et al.*, 2018a].

Even not Car related corporations are researching it, for example, NVIDIA Corporation, a technology company mostly known for its computer graphical card development, created NVIDIA Drive, a *development platform and reference architecture for designing autonomous vehicles*[2].

2.4 Torcs

Torcs is a widely used simulator to test and improve Machine Learning autonomous driving implementations since it provides a simulator with both open-source software, which makes possible the gathering of environment information (MDP state) and a somewhat realistic physics model.

The simulator had, until 2017, a dedicated championship [3] where teams would develop their cars and compete, to achieve the best performance in track.

Because of that, a lot of implementations using Reinforcement Learning already had been studied, like an implementation of deep deterministic policy gradient, to study *Deep Reinforcement Learning for Autonomous Driving* [Wang *et al.*, 2018b] or the study of Soft Actor-Critic and Rainbow DQN algorithms [Güçkıran and Bolat, 2019].

Our work objective is to study referred implementations and choose different algorithms and set an algorithmic baseline to other researchers know where to start their implementation.

3 Materials

All the materials presented are saved in a Git Hub repository: Reinforcement-Learning-algorithms-applied-to-Self-Driving-Racing-Car

As already referred, Torcs was used to simulate the environment where the agent is going to take part.

The language used to implement the needed resources was Python.

To wrap the environment and make an understandable interface, open AI gym, since it is commonly used and follows the MDP structure.

As for the agent, data, Python was used with the Stable Baselines 3 framework [Raffin *et al.*, 2021].

We are going to explain more in-depth how the communication was made with the game and what implication it has in the algorithm performances.

In order to analyse the differences in the algorithm performances, despising the visual analysis, TensorBoard, a visualization toolkit created for TensorFlow was used [Abadi *et al.*, 2015].

[2]https://www.nvidia.com/en-us/self-driving-cars/

[3]https://torcs.sourceforge.net/

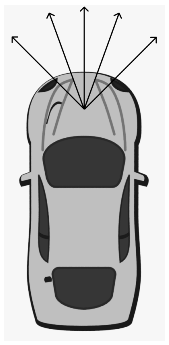
Figure 2: Illustration of Vehicle Sensors

4 Methods

4.1 Torcs and agent environment communication

In the context of game data retrieving was used a Torcs plugin created for the Simulated Car Racing Championship 2015, hosted by researchers from the University of Adelaide and the Politecnico de Milano [4].

The bot perceives the racing environment through a number of sensor readings which provide information both about the surrounding game environment (e.g., the tracks, the opponents, the speed, etc.) and the current state of the race (e.g., the current lap time and the position in the race, etc.). It also provides a UDP interface to connect and transfer data between the implementation and the simulator, so every 20ms (a game tick) the game state is reported through UDP e and the server will wait 10ms for a response [Loiacono *et al.*, 2013].

In order to interact with this interface was designed a Client, that sends the initial configuration, the angles of the distance sensors, like shown in Figure 2, and formats messages to and from the server.

To provide a widely known interface in terms of code to the community, the environment and his proprieties were wrapped in an Open AI Gym interface.

Open AI Gym

The Open AI Gym library is commonly used to represent RI environments because it follows the MDP interface.

In our implementation, only a few observations provided by the plugin were used, from the referred in the [Loiacono *et al.*, 2013], being them the angle of the car with the track, the value of the 19 sensors of track distance and the current speed of the car. Those observations were normalized to be sent to the agent.

In terms of actions, two approaches were considered, discrete and continuous. In the discrete type, the action space is defined by 4 actions, steering right, steering left, braking and accelerating, with the car committing to these actions, for example, when a braking action is sent, that means that the car is only going to brake fully.

In the continuous, 2 different actions were defined, being them steer left/right and accelerating/braking, with ranges -1 to 1. In this approach, the car can, for example, accelerate and steer at the same type and can control the amount of action that it does, this means that can partially accelerate or partially brake.

[4]https://cs.adelaide.edu.au/~optlog/SCR2015

4.2 Agent

As said before, the agent was developed using the python library Stable Baselines 3 [Raffin *et al.*, 2021]. The library provides a set of reliable implementations of reinforcement learning algorithms using PyTorch, making possible an easy implementation of different algorithms like TD3, SAC, DQN and DDPQ among others. In our case, the algorithms chosen were:

- **DQN:** A deep implementation of a Q Network, discrete.
- **A2C:** A synchronous, deterministic variant of Asynchronous Advantage Actor Critic (A3C), in a continuous form.
- **TD3:** TD3 is a direct successor of DDPG and improves it using three major tricks: clipped double Q-Learning, delayed policy update and target policy smoothing.

The main focus of this paper is to optimize and compare the referred agents to evaluate their differences and performance parameters and results.

4.3 The Reward Function

The correct choice of a reward function is one of the most significant decisions when developing a successful agent. In the context of our problem, we decided to have the reward function being:

$$reward = speed * \cos(angle) \tag{1}$$

For context, the speed variable represents the longitudinal speed of the car and the angle means the angle existing between the longitudinal dimension of the car with the track centre. We can easily describe our function being how much the car can maintain in the centre of the track, with the most speed possible. That happens because if the car direction is different from the track direction, the reward will be lower

We are primarily trying to compare the algorithms, so, in the beginning, the track performance shouldn't be the focus.

5 Experiments

5.1 Experimental Setup

To measure and get metrics about the algorithm performance, a tensorboard setup was made and a call back to backup the model every 10000 steps was created. In that way, we were able to measure the model improvements not only using statistics but also visually seeing the model taking action, at different moments in time.

To train the model, the track represented on Figure 3 was used. The choice of this track was based on her variety since it seems to have a lot of different characteristics for the agent to learn, with wide and tight corners for both left and right and with both long and short acceleration sectors.

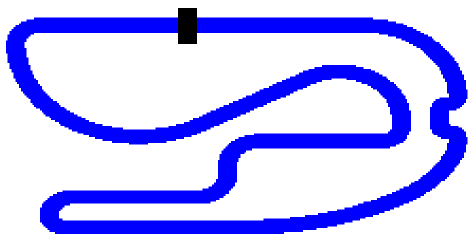

Figure 3: Track used in Agent Training and Analysis (Road Track - "E-Track 1")

5.2 DQN

Deep Q-Network is an Algorithm first introduced in the article *Playing Atari with Deep Reinforcement Learning* [Mnih *et al.*, 2013]. As the name explains, it is a deep version of the Q-Learning algorithm.

Q-Learning is a RI algorithm based on a matrix, the Q-Table shaped as [Actions, States], where each cell stores the result of the agent performance (Equation 2). In every iteration, the cell value, for the current environment state is updated, and an agent will choose the best action for the current environment state. For the agent to improve, the best action isn't always chosen, the agent has a parameter, epsilon, which defines how much the agent should explore (choose a random action), or exploit (perform the best action).

$$Q(s,a) := Q(s,a) + \alpha\left(r + \gamma \max_{a' \in A} Q(s',a') - Q(s,a)\right) \tag{2}$$

Deep Q-Learning, as referred, is an improvement to the Q-Learning algorithm, where a memory with a few previous results is used and instead of a Q-Table, a Deep Network is used to estimate the values of the table and that network is trained.

Stable Baselines 3 DQN implementation is based on the algorithm 1, presented on the implementation of the referred paper [Mnih *et al.*, 2013].

The first experiment done was using the Stable-Baselines 3 default parameters, to establish a well-designed reference for further tests.

In the second experiment, was made a try to increase the complexity of the network, in order for it to not overfit with the current data.

For the third and last experiment, a try to repeat the first experiment was made with a few more steps, and a higher exploration time, in order to give the car more time to achieve some progress with different actions.

Experiments Parameters used present in Table 1, labeled as follows:

- **Nr. Steps:** Number of training steps
- **LR:** Learning rate
- **ER:** Exploration rate range (Variable Exploration through the experience)
- **EF:** Exploration fraction (Fraction of experience steps where exploration will decrease until reaches a final state.
- **Net Arch:** Architecture of neural network used, being the number of elements, the number of layers and the number itself, the number of neurons of each layer

Nr. Steps:	LR	ER	EF	Net Arch
1000000	0.0001	$1 - 0.05$	0.1	$64, 64$
1000000	0.0001	$1 - 0.05$	0.1	$256, 256, 256, 100$
2500000	0.0001	$1 - 0.05$	0.4	$64, 64$

Table 1: DQN Experiences

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode = 1, $M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t$ = 1, $T$ do
        With probability $\epsilon$ select a random action a
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1}$ = $s_t$, $a_t$, $x_{t+1}$ and preprocess $\phi_{t+1}$ = $\phi(s_t + 1)$
        Store transition ($\phi_t$, $a_t$, $r_t$, $\phi_{t+1}$) in $D$
        Sample random minibatch of transitions ($\phi_t$, $a_t$, $r_t$, $\phi_{t+1}$) from $D$
        Set $y_j \begin{cases} r_j \text{for terminal} \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a_0; \phi) \text{ else} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \phi))^2$ according to equation 3, in [Mnih et al., 2013]
    end for
end for
```

5.3 A2C

A2C, or Advantage Actor-Critic, is a synchronous version of the A3C (Asynchronous Advantage Actor Critic) political gradient method, it was first introduced in the article *Asynchronous Methods for Deep Reinforcement Learning* [Mnih *et al.*, 2016].

As an alternative to the asynchronous implementation of A3C, A2C is a synchronous, deterministic implementation that waits for each actor to finish its experience segment before updating, averaging over all actors. This makes more efficient use of GPUs due to the larger batch sizes (Equation 3), $nenv$ is number of environment copies running in parallel and $nsteps$ is the number of steps to run for each environment per update.

$$batchsize := nsteps * nenv \tag{3}$$

A2C, as referred, is a synchronous implementation of A3C, where to avoid the use of a replay buffer it uses multiple workers. Stable Baselines 3 A2C implementation of each worker is based on the algorithm S3 Asynchronous advantage actor-critic, 2, presented on the implementation of the referred paper [**?**].

In the experiments we let the driver train for approximately 1 hour to 2 hours, we also had in account to train by manually starting every episode or by letting the code do it for us corresponding to 100k or 1M training steps, we used different values for the different parameters, as well as 2 optimizers, RMSprop and Adam. Experiments Parameters used present in Table 2, Table 3, Table 4, labeled as follows:

- **Nr. Steps:** Total number of training steps
- **LR:** Learning rate
- **ER:** Exploration rate range (Variable Exploration through the experience)
- **EF:** Exploration fraction (Fraction of experience steps where exploration will decrease until reaches a final state.
- **nsteps** The number of steps to run for each environment per update

Nr. Steps:	LR	ER	EF	nstep
100000	0.0007	$1 - 0.05$	0.1	10
100000	0.0007	$1 - 0.05$	0.1	100

Table 2: A2C Experiences - Episodes Manual - RMSprop

Nr. Steps:	LR	ER	EF	nstep
1000000	0.0007	$1 - 0.05$	0.1	10
1000000	0.0007	$1 - 0.05$	0.1	100

Table 3: A2C Experiences - Episodes Automatic - RMSprop

Nr. Steps:	LR	ER	EF	nstep
1000000	0.0007	$1 - 0.05$	0.1	10
1000000	0.0007	$1 - 0.05$	0.1	100

Table 4: A2C Experiences - Episodes Automatic - Adam

5.4 TD3

TD3 is the successor to the Deep Deterministic Policy Gradient (DDPG). Up until recently, DDPG was one of the most used algorithms for continuous control problems such as robotics and autonomous driving.

While DDPG can achieve great performance sometimes, it is frequently brittle with respect to hyperparameters and other kinds of tuning. This is caused by the algorithm continuously over estimating the Q values of the critic (value) network.

These estimation errors build up over time and can lead to the agent falling into a local optimum or experiencing catastrophic forgetting. Twin Delayed DDPG (TD3) is an algorithm that addresses this issue by introducing three key features:

Algorithm 2 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

```
Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R \begin{cases} 0 \text{ for terminal } s_t \\ V(s_t, \theta'_v) \text{ for non-terminal } s_t \end{cases}$ // Bootstrap from last state
    for $i \in \{t-1, ..., t_{start}\}$ do
        $R \leftarrow \gamma R$
        Accumulate gradients wrt $\theta'$:$d\theta \leftarrow d\theta + d\Delta_{\theta'} \log \pi(a_t|s_t;\theta')(R - V(s_i;\theta'_v))$

        Accumulate gradients wrt $\theta'_v$:$d\theta_v \leftarrow d\theta_v + \delta(R - V(s_i;\theta'_v))^2/\delta\theta'_v$
    end for
    Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
until $T > T_{max}$
```

- Clipped Double-Q Learning: Using a pair of critic networks, TD3 learns two Q-functions instead of one (hence “twin”) and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.
- “Delayed” Policy Updates: Delayed updates of the actor, TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.
- Target Policy Smoothing: Action noise regularisation: TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

Experiments Parameters used present in Table 5, labeled as follows:

- **Nr. Steps:** Number of training steps
- **LR:** Learning rate
- **ER:** Exploration rate range (Variable Exploration through the experience)
- **EF:** Exploration fraction (Fraction of experience steps where exploration will decrease until reaches a final state.
- **Net Arch:** Architecture of neural network used, being

the number of elements, the number of layers and the number itself, the number of neurons of each layer

Nr. Steps:	LR	ER	EF	Net Arch
1000000	0.0001	$1 - 0.05$	0.4	300, 400
1000000	0.0001	$1 - 0.05$	0.1	64, 64
1000000	0.0001	$1 - 0.05$	0.1	256, 256, 256, 100

Table 5: TD3 Experiences

Algorithm 3 Twin Delayed DDPG (TD3)

```
Input: initial policy parameters $\theta$, Q-function parameters x, x2, empty replay buffer $D$
Set target parameters equal to main parameters $\theta_{targ} \leftarrow \theta$ , $\varnothing_{targ,1} \leftarrow \varnothing_1$, $\varnothing_{targ,2} \leftarrow \varnothing_2$
repeat
    Observe state s and select action $a$ = clip($\mu_\theta$(s) + $\epsilon$, $a_{Low}, a_{High}$, where $\epsilon$   $N$
    Execute $a$ in the environment
    Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
    Store $(s, a, r, s', d)$ in replay buffer $D$
    If $'s$ is terminal, reset environment state.
    if it's time to update then
        for j in range(however many updates) do
            Randomly sample a batch of transitions, $B$ = $(s, a, r, s', d)$ from $D$
            Compute target actions
            $a'(s')$ = clip ($\mu\theta(s')$ + clip($\epsilon, -c, c$), $a_{Low}, a_{High}, \epsilon$ $N(0, \sigma)$ )
            Compute targets
            $y(r, s', d) = r + \gamma(1 - d) minQ_{\varnothing_{targ,i}(s',a'(s'))}$
            Update Q-functions by one step of gradient descent using
            $\nabla_{\varnothing} \frac{1}{|B|} \sum_{(s,a,r,s',d)\epsilon B} (Q_{\varnothing_i(s,a) - y(r,s',d)})^2$
            for $i$ = 1,2
            if j mod $policy\_delay = 0$ then
                Update policy by one step of gradient ascent using
                $\nabla_{\varnothing} \frac{1}{|B|} \sum_{s\epsilon B} Q_{\varnothing_1(s,\mu\theta(s))}$
                Update target networks with
                $\varnothing_{targ,i} \leftarrow \rho\varnothing_{targ,i} + (1 - \rho)\varnothing_i$
                $\theta_{targ} \leftarrow \rho\theta_{targ} + (1 - \rho)\theta$
                for $i$ = 1,2
            end if
        end for
    end if
until convergence
```

6 Results and Discussion

6.1 DQN

Experience 1

In experience 1, with the default library hyper-parameters, we can see that the mean reward of each episode has a continuous growth under the 200000 step, when it reaches its max, of proximally 30000, then it drops remaining mainly constant in value 10000 (Figure 4).

Executing the model, we verified that the car goes straight a certain amount, then it turns lightly to the left turning after all the way to the right getting crashing into the wall, without even reaching the first track corner. Even executing the best model, of step 180000, the car would not get to the corner. Observing the results, we can see that the model is getting better, correcting the left turning and going fully to the right.

Visualizing the Loss graph (Figure 5) we can see that the loss is constantly increasing over time, what seems to be happening is that the agent is overfitting the model to the straight at the beginning of the track until exploration drops to the minimum and then it won't learn how to proceed in the corner.

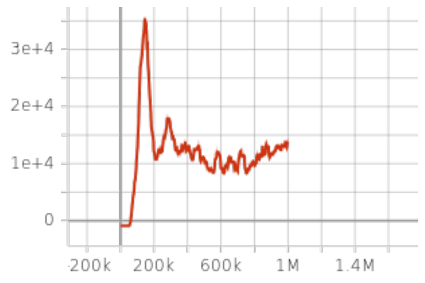

Figure 4: Graph of Reward of the episode (y) by the total number of steps(X) - Experience 1

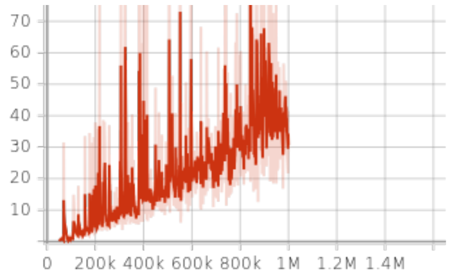

Figure 5: Graph of Loss of the episode (y) by the total number of steps(X)- Experience 1

Experience 2

In experience 2, as referred to previously, the number of layers and their size were increased to experience the effect of a bigger network in a complex action that is driving.

Analysing the Figure 6 we observe that the experience was a failure, not only the final reward was a lot less, when compared with experience 1, as the loss was even higher. When opening testing the model in-game, in his best performance, the car behaviour is weird, with the car constantly accelerating and braking without doing relevant progress.

Inspecting the Figure 7, we can assert that the loss of the algorithm, through the execution is a lot bigger when compared to the one the experience 1 (Figure 5), which confirms statistically that the experience performs worst than the previous one.

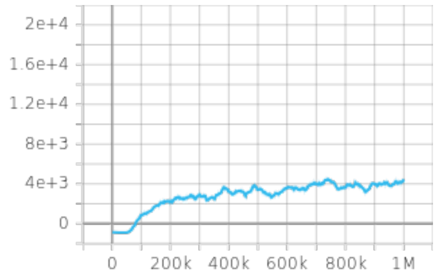

Figure 6: Graph of Reward of the episode (y) by the total number of steps(X)- Experience 2

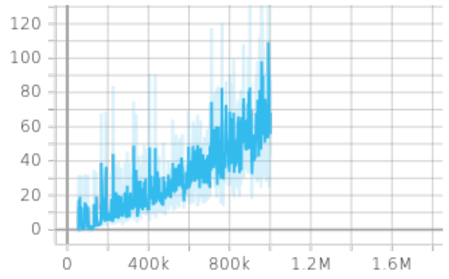

Figure 7: Graph of Loss of the episode (y) by the total number of steps(X)- Experience 2

Experience 3
Experience 3 was created to further explore the results of experience 1 since from the analysis previously made, the experience seems to have the capability to perform better if it has more time to do it, as we saw, from step 180000 to the last one, progress was made with the car trying to keep in track, but failing on that task. We expect that letting the car have higher exploration steps, and exploring more of the possibilities in the environment is going to increase the performance of the agent, by trying more combinations of actions for a given state.

Visualizing the Figure 8 we can see that increasing the number of steps and the exploration rate had a good effect on the model until it dropped at the end of the experience. The fact is, that in this experience, the agent could perform better than in experiences 1 and 2 using the same algorithm, achieving a better reward maximum and more than doubling the best result achieved previously, in experience 1 (Figure 4).

6.2 A2C

In experience 1 and 2 we used the parameters of Table 2. From those we got the reward mean per episode from the training resulting in Figure 10 and Figure 11, respectively.

In experiences 3 and 4 we used the parameters of Table 3. From those we got the reward mean per episode from the training resulting in Figure 12 and Figure 13, respectively.

In experiences 5 and 6 we used the parameters of Table 4 and we change the optimizer to $Adam$. From those we got

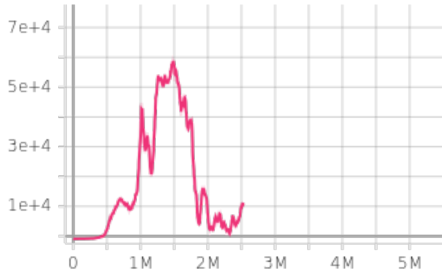

Figure 8: Graph of Reward of the episode (y) by the total number of steps(X)- Experience 3

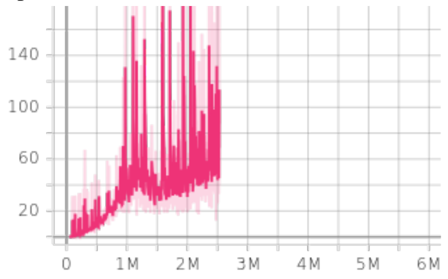

Figure 9: Graph of Loss of the episode (y) by the total number of steps(X)- Experience 3

the reward mean per episode from the training resulting in Figure 14 and Figure 15, respectively.

Experience 1 and 2
In both of the experiences, the mean reward decreases in the beginning, that happens because we are starting the episodes manually, and the first episode has some delay at connecting the server to the client making it accelerate until the connection is made, after that, we can see the value growing in both cases, in Figure 11 the growth looks slower because the car even tho it's moving it just turns right and hit the wall before getting to the first track turn.

Experience 3 and 4
We can see that the mean reward of each episode in experience 1 decreases until proximity 120k step, and then starts to grow to reach 1M steps without getting to positive values, this happened because the car is in constant action of breaking and accelerating making it move slow, therefore having low reward values.

In experience 2 a different result is shown, where the mean reward for each episode increases since the start, at 420k steps it keeps growing but at a smaller rate, we can explain it by running the model and seeing that the car accelerates a lot in the beginning and doesn't turn when the first track turn appears.

Experience 5 and 6
In experience 5 the mean reward values grow and reach positive values unlike experience 3, getting to values close to other experiences when $nsteps$ is 100. These values are reached because the car is accelerating as much as possible to get the reward higher, this effect makes it doesn't turn on the first turn of the track, this can be seen when we can the model.

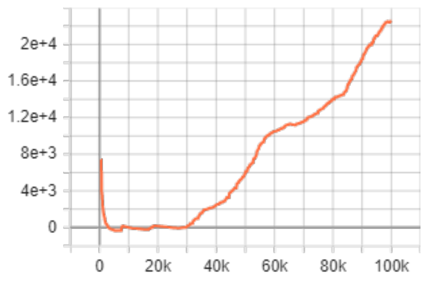

Figure 10: Graph of Reward of the episode (y) by the total number of steps(X) - Experience 1

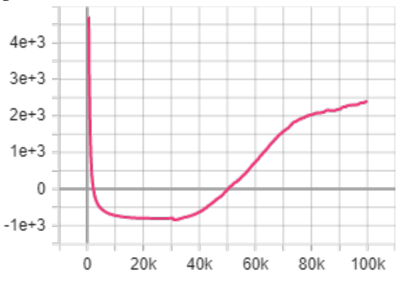

Figure 11: Graph of Reward of the episode (y) by the total number of steps(X) - Experience 2

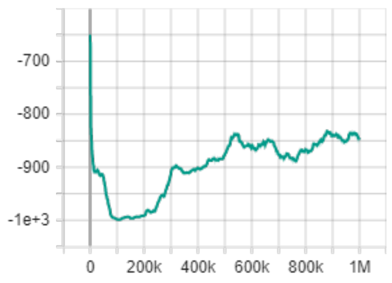

Figure 12: Graph of Reward of the episode (y) by the total number of steps(X) - Experience 3

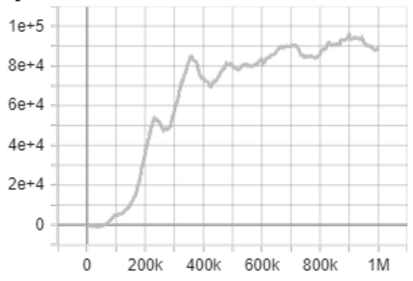

Figure 13: Graph of Reward of the episode (y) by the total number of steps(X) - Experience 4

The mean reward for experience 6 starts growing slowly until 545k steps, and after that, it keeps growing and decreasing, from 855k and 980k steps a big growth happens, we assume the cause of it is the car start to be able to reach the first track turn, at the end steps the reward starts to get lower because the car starts to turn right before the turn.

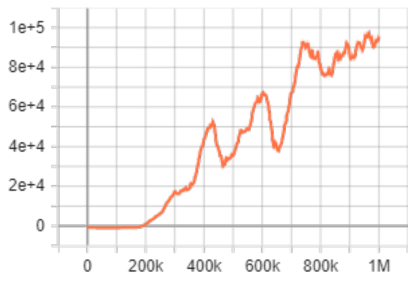

Figure 14: Graph of Reward of the episode (y) by the total number of steps(X) - Experience 5

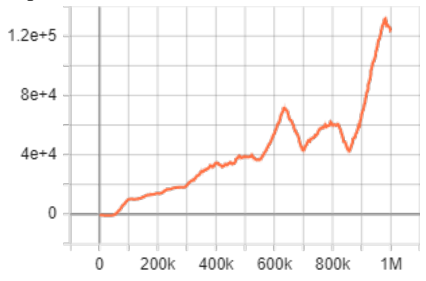

Figure 15: Graph of Reward of the episode (y) by the total number of steps(X) - Experience 6

6.3 TD3

Experience 1

In experience 1, with the default library hyper-parameters, we can see that was a slow and time-consuming process, where the reward values obtained are very similar. Noting slight progress in the path of the car on the track, however, as soon as it reached the first corner, the result was always the same, not making any significant changes in the path before getting there. Thus, the values always are all for around 180/190.

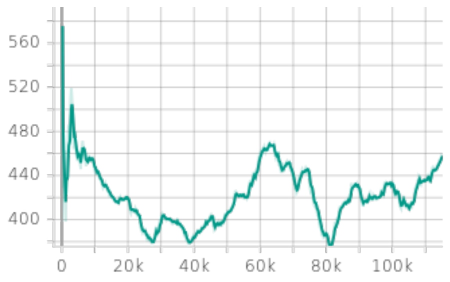

Figure 16: Graph of Reward of the episode (y) by the total number of steps(X)- Experience 1

Experience 2

In experience 2, we used the parameters of Table 5. From those we got the reward mean per episode from the training resulting. We can see that the mean reward of each episode in experience 1 decreases until proximity 200k step, and then starts growing until 230k and then start decreasing again, and this process will happen repeatedly at 150k steps intervals, and then start to happen at smaller and smaller intervals until it reaches 900k steps where it will suddenly go up.

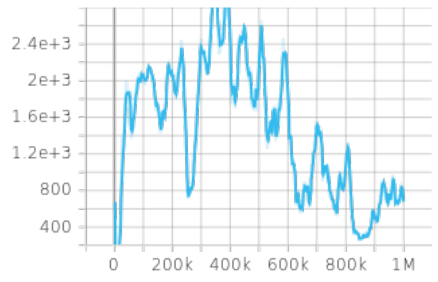

Figure 17: Graph of Reward of the episode (y) by the total number of steps(X)- Experience 2

Experience 3

In experience 3 a different result is shown, where the mean reward for each episode will decreases repeatedly in the first steps, thus keeping the reward function stable until 450k steps, because the behaviour of the car was a little strange at the start, starting right away by turning to one side and not progressing on the track. After that, the car learns to just accelerate and not turn, starting to rise spontaneously and progressively the reward function, but with some decreases in process, reaching a maximum reward value when the 1M steps are reached.

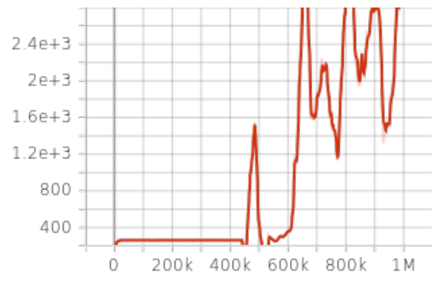

Figure 18: Graph of Reward of the episode (y) by the total number of steps(X)- Experience 3

6.4 Geral Analysis

Observing all the achieved results we can clearly say that the proposed objectives of the paper were clearly not achieved. In fact, we weren't capable to make real progress in the analysis of the algorithmic performance and the car racing performance.

Looking back, we can affirm that a few errors in the development and modelling of the testing were made.

Beginning, for example, the choice of the simulator, Torcs, being a community-developed simulator and has not been supported since 2017 was a bad choice because the interface is old and can produce some errors when working with newer platforms like the ones used.

Related to the previous point, the game plugin used to retrieve information from the game was made in 2015, to work with a TORCS implementation from that year, which also can produce some errors nowadays.

As for the implementation of the model, used Stable-Baselines3, we agreed that it is a good library, but more fitted to work well with simpler problems with the base parameters, and the way to change and improve those parameters is quite complex. The use of another library, like Keras-RL[Plappert, 2016] with a well-defined and customized model, would probably be an improvement in order to achieve the expected results.

Due to limited computational resources, we weren't capable of giving more time to the training, which would mean that probably we would achieve better results with a longer experience because the agent would have more time to improve itself and to adapt to the environment.

6.5 Algorithm Comparison

Even though a few errors were made in the implementation, we can still try to compare the performances of the different algorithms.

Beginning with the comparison between Discrete, and Continuous algorithms, we can observe that the ones that are continuous were capable of archiving better rewards, even being more complex. That could be a result of the capability to proceed to small movements, getting the car stable at the middle line and consequently increasing the reward, when compared to Discrete movements that fully commit the car to one direction.

Regarding the two continuous algorithms studied, we can conclude that the maximum value reached was using TD3, but when comparing all experiences we conclude that A2C is faster and achieves more constant values, this happens because it uses parallel actor-learners and we can see in the referred paper [Mnih *et al.*, 2016] parallel actor-learners have a stabilizing effect on training. TD3 was a better algorithm for general performance and the ability to transfer learning to other markets, since is an off-policy algorithm, it uses all of the history to learn at each timestep, getting higher results with the experiments.

7 Conclusions

In the end, it was not possible to obtain a successful car and none of the training could take the first curve of the track. Still, with the use of different algorithms, like DQN, A2C and TD3, it was possible to train and make the cars learn and evolve to the point of reaching that curve.

We were able to use and analyze different Reinforcement Learning algorithms mentioned before, and we were also able to improve our reward function throughout the tests, so at the beginning it made the cars turn at the start, and at the end we got the cars running in the correct direction of the track, so even not concluding the track it was possible to accomplish some objectives.

References

[Abadi *et al.*, 2015] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[Güçkıran and Bolat, 2019] Kıvanç Güçkıran and Bülent Bolat. Autonomous car racing in simulation environment using deep reinforcement learning. In *2019 Innovations in Intelligent Systems and Applications Conference (ASYU)*, pages 1–6. IEEE, 2019.

[Haenlein and Kaplan, 2019] Michael Haenlein and Andreas Kaplan. A brief history of artificial intelligence: On the past, present, and future of artificial intelligence. *California management review*, 61(4):5–14, 2019.

[Jin *et al.*, 2018] Junqi Jin, Chengru Song, Han Li, Kun Gai, Jun Wang, and Weinan Zhang. Real-time bidding with multi-agent reinforcement learning in display advertising. In *Proceedings of the 27th ACM international conference on information and knowledge management*, pages 2193–2201, 2018.

[Loiacono *et al.*, 2013] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Simulated car racing championship: Competition software manual. *CoRR*, abs/1304.1672, 2013.

[McCarthy *et al.*, 1955] J. McCarthy, M. L. Minsky, N. Rochester, and C.E. Shannon. A proposal for the dartmouth summer research project on artificial intelligence, 1955.

[Mnih *et al.*, 2013] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[Mnih *et al.*, 2016] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[Plappert, 2016] Matthias Plappert. keras-rl. https://github.com/keras-rl/keras-rl, 2016.

[Raffin *et al.*, 2021] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[Wang *et al.*, 2018a] Jiadai Wang, Jiajia Liu, and Nei Kato. Networking and communications in autonomous driving: A survey. *IEEE Communications Surveys & Tutorials*, 21(2):1243–1274, 2018.

[Wang *et al.*, 2018b] Sen Wang, Daoyuan Jia, and Xinshuo Weng. Deep reinforcement learning for autonomous driving. *arXiv preprint arXiv:1811.11329*, 2018.