POLITECNICO DI TORINO

Master's Degree in Software Engineering

Master’s Thesis

A comparison of Different Machine Learning Techniques to Develop the AI of a Virtual Racing Game

Supervisor

Prof. Andrea Bottino

Dott. Francesco Strada

Candidate

Alessandro Picardi

Academic Year 2020/2021

Acknowledgements

First of all, I would like to express my sincere gratitude to Politecnico di Torino and partner organizations, for letting me be part of this incredible and stimulating University. Further, I would like to thank my supervisors Prof. Andrea Bottino and Dott. Francesco Strada for the thoughtful comments and recommendations on this dissertation. I am also thankful to my family that always support me without asking me anything. A special thanks go to Jana, sometimes far in the distances, never far to the heart. To conclude, I cannot forget to thank my 8 friends for all the love and laugh they gave me in these intense academic years.

Abstract

Nowadays machine learning (ML) is a field of study that is applied in numerous fields of application: Image classification, Identity fraud detection, Market forecasting, Customer segmentation and others. Another interesting field of study for ML is Video Game. In a virtual environment we can train an Artificial Intelligence (AI) and not script it with thousands of lines of code. A ML-AI could be a Non Player Character that interact with a human player in a friendly or hostile way. It could be an agent that learn how to perform a task in a single player game. The idea of this dissertation is to create a virtual environment with three different AIs trained with a different approach: Reinforcement Learning, Imitation Learning and Curriculum Learning. These agents are trained to compete against a human player in a racing game.

Contents

List of figures

1 Intro

1.1 Purpose of the research

The goal of this project was to develop virtual environment where 3 different AIs compete with a player to win a car race. The AIs were trained with a particular ML technique: Reinforcement Learning (RL), Curriculum Learning(CL) and Imitation Learning(IL). The three elements were later compared to choose the best learning technique for this task. Previous work has been made on this subject.

Amira Youseef et al used IL and RL combined to create a shooting game where AI agent has more human-like behavior[1] . Kun Shao et al used a particular version of Curriculum Transfer Learning (a different version of CL) to train multiple units in StarCraft micro-management[2]. ShuQin Li et al train a reinforcement learning agent to play Surakarta, a particular chess-like board game[3]. My idea was not to combine or develop a single learning technique but create an environment with 3 different ML agents and a player where they race all against each other. The purpose was to master the learning approaches and show how Machine Learning can create interesting challenge for a human player in the field of gaming. Moreover I wanted to bring some innovations and hints for future development of the library unity ml agents. This is one of the tools I used in the creation of this project, it is an interesting library in continuous and fast development.

1.2 What is Machine Learning

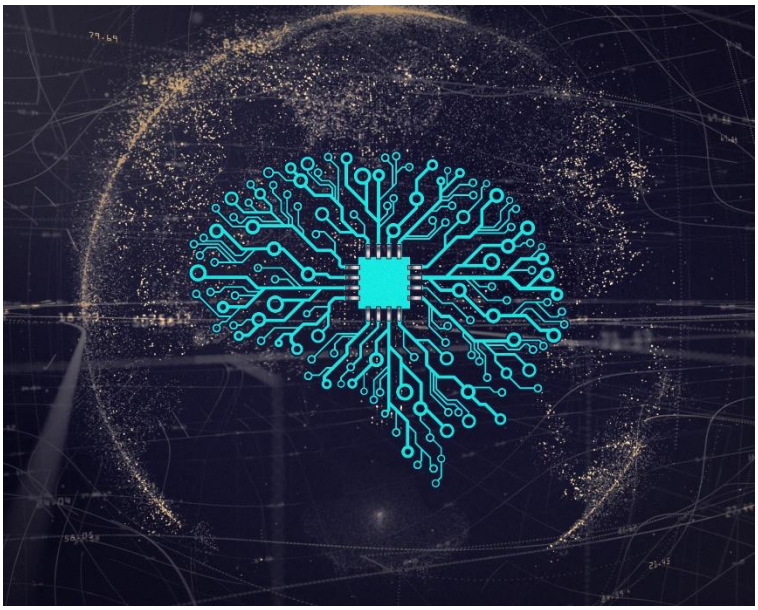


Figure 1: Machine Learning

Machine Learning (ML) is a very popular technique of future prediction or information classification. It can help a lot of people in the process of decision making. The algorithm of ML are trained over batches of examples. With this the build their own experience database and data history. The agent of ML are trained again and again over the examples and in this way they can identify patterns in data and they can make predictions for the future.

With the help of ML, humans can create intelligent systems that are able of taking decisions autonomously.

Machine Learning is a combine different science field: mathematics, statistics and computer science. Math helps us for developing machine learning models; statistics draws inferences from data and computer science deals with the implementation of algorithms.

Building a model and implementing is not enough, we must tune the model in a proper way to have accurate results. To reach an optimum we tune something called hyperparameters.

1.3 The reasons of Machine Learning

The world today is evolving and changing fast and so are the needs of people. Moreover, we are living a fourth industrial revolution of data. In order to get meaningful information from data and learn how people interface with them we need algorithms that can select them and give us a result that benefit us. Machine Learning has changed dramatically several industries like medicine, healthcare, manufacturing, banking and so on. It has become an indispensable part of modern industry.

The computation power, in this scenario Machine Learning has added a new dimension in our perception of information. Machine Learning is around us every day: apps, phones, IoT devices we use on a daily basis are enhanced by Machine Learning.

For example, Google can autocomplete your research while you are writing on the keyboard, this feature is based on your browsing habits. Spotify operates in a similar way: it is capable perform predictions based on your listen history and recommending the music and the podcast that you would want to listen. In this way you can easily discover music and website close to your interest but it will be hard to find a brand new artist or website far from your bubble.

Moreover, Machine Learning is taking over monotonous iterative tasks. This is possible thanks to the great amount of data that we generate everyday.

1.4 How Machine Learning Uses Data

With a massive increase in data, there is a need of a strong system to handle this load of information. Deep Learning for example can generate predictions with a massive amount of data.

Machine Learning algorithms find patterns in data to perform classification and future predictions. When a new input arrives in the model ML uses the pattern found before and try to perform a future prediction. We can optimize our models using various techniques like gradient descent or exhaustive search. In this way the model learns to adapt to new example and give us better results.

1.5 Types of Machine Learning

ML algorithms can be classified into 3 main types:

-Supervised Learning

-Unsupervised Learning

-Reinforcement Learning

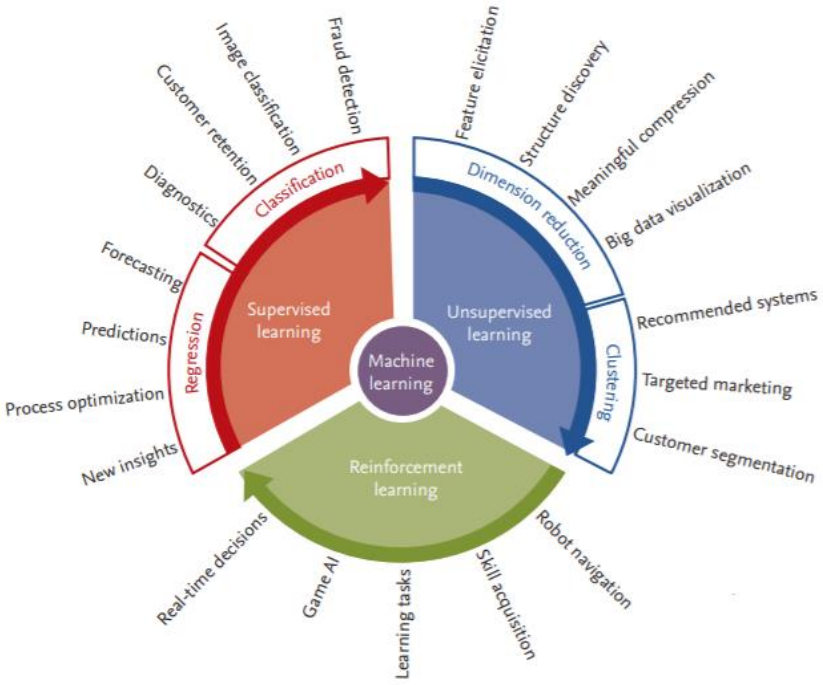


Figure 2: Three types of ML with subsection

2 Theoretical Background

2.1 Reinforcement Learning

In this kind of learning an agent gather observation of the world, it interacts with these performing actions and it receives rewards (or reinforcements) that tell him how well (or bad) it is behaving. For example, in the game of chess rewards can be 0 for losing, 1 for winning and 0.5 for draw. The goal of reinforcement learning (RL) is maximizing the sum of rewards. The reward function may be unknown to the agent. You can try to picture yourself playing a game where nobody explain you the rules. You continue to make moves and at a certain point the referee yells that you have won (or lost). This is reinforcement learning.

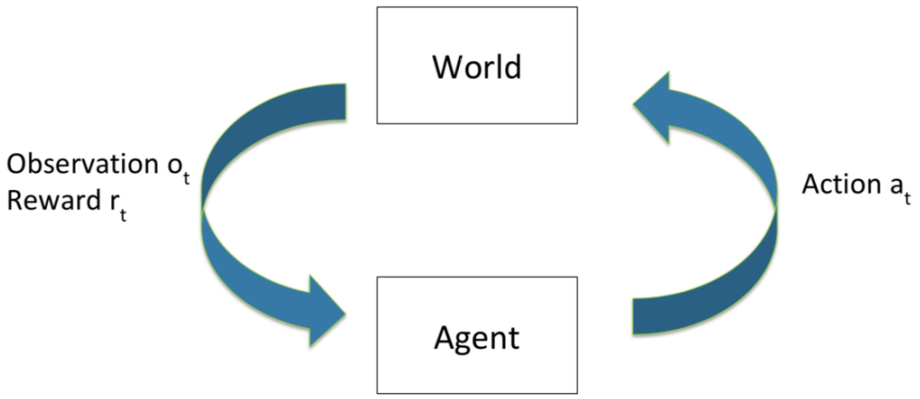


Figure 3: The reinforcement learning cycle

When you design an AI, a good start is formalizing a reward function. This function is often short and easy to formalize. Take a racing game as example, we do not need some crazy and long code to make the agent understand if it has won, lost or crash. Another example could be the game of chess, it is not hard to define that if you do checkmate you win and if you receive it you lose. These two cases are example of Sparse Reward and an agent will have more difficulties learning process. A simpler scenario is Dense Reward.

2.1.1 Sparse Reward

For the majority of the training process a sparse reward function rests 0, only in a few state/action transitions it has a positive or negative value. In the game of chess for example, it could be +1 for a winning game and -1 for a losing game. An RL agent in a sparse reward environment will receive almost no feedback, it will be hard to tell for him about the goodness of his actions.

2.1.2 Dense Reward

This kind of reward function gives a nonzero value to most of state/action transition, in this way, at every time step (almost) the agent gets feedback. If you think of chess again, a dense reward function could be +1 for every piece captured by the agent and -1 for every piece captured by the opponent.

Learning in environments with dense rewards turns is less challenging than with sparse rewards. The reason of that is that the agent will often visit state/action transition that do not give any reward. Indeed, the most possible trajectories will have a zero reward. In this way the agent is obliged to explore a very large set of trajectories before learning a policy.

An example of sparse reward experiment is the “Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards” of Mel Vecerik et al. Here we see one of the biggest companies of Machine Learning trying to solve a sparse reward problem: the insertion tasks, a mechanical arm has to insert a plug in a socket

The researcher at Deepmind created an ad hoc sparse reward function which returned +10 if the plug was within a little distance to the goal. The reward r is defined like this:

$$r = \begin{cases} 0, & \sum_{i \in sites} W_g \left|\left| g_i - x_i \right|\right|_2 > \epsilon \\ 10, & \sum_{i \in sites} W_g \left|\left| g_i - x_i \right|\right|_2 < \epsilon \end{cases}$$

where x_i is the position i^{th} tip site on the plug

g_i is the i^{th} goal site on the socket

W_g contains weighting coefficients for the goal site error vector

ϵ is a proximity threshold

If the robot put the plug in a position that is inside the threshold, the reward of +10 is given and the episode is terminated [5]

2.2 Imitation Learning

There are domains that are too complex to find the right definition of a reward function. If we think of a racing car in a circuit, how can we define the right behavior of this agent? It should arrive first defeating the other opponents, but it should also drive safe and not crash. It should not waste energy/fuel in moment where it is not requested. It should not make the passengers bouncing like a ball (if there is a passenger on board), but it must make an emergency brake if there is a dangerous situation ahead. The process of weighting the factors could be tricky. There is also an essential factor we have forgotten to mention: consider other opponents in the race. If we forgot a factor, the behavior will select a high weight for the omitted factor. This will happen because the behavior is trying to render the remaining factors at their max.

If we want to eliminate this issue, one technique is to do protracted testing in simulation, find what are the bad behaviors and adjust the reward function so that these are eliminated. Another way is to build a reward function based on other sources of data. One source is the agent itself, the other can be real human racers.

The field of imitation learning has a focused: learn the right behavior observing an expert. We give to the agent a batch of expert examples and then we let it learn from them. The problem of imitation learning has two approaches.

The first is Imitation Learning (IL) in his natural form. The agent takes state-action pairs and use supervised learning on them. The goal of this process is learning a policy $\pi(s)$. There was some success in robotic with this approach, but some issues arose. Imitation learning has a superior limit: the teacher. It will never pass the teacher and his optimal policy $\pi^*(s)$. When humans learn with imitation, we consider them apes to describe their behavior. In a similar way we can consider an ape agent because it use imitation learning. It does not understand the meaning of his actions. Furthermore if the training set has a small deviation this will bring a time increasing error. This could lead to major failure.

A second and smarter approach is Inverse Reinforcement Learning (IRL). How it works? The agent observes the expert's actions and try to understand and formulize an approximation of the reward function of the expert . After a reward function is obtained the agent will derive a policy. In this way we do not need thousands (or even millions) of examples from the expert behavior, the agent will obtain a robust policy also with a few examples. Moreover, if the learner understand that an expert could have a suboptimal behavior it could achieve better results than the expert because it is optimizing a reward function and not copying a behavior.

The main goal now is to find a reward function from the expert rewards. Our point of start could be a rational behavior of the expert.

Here we have the first issue: a lot of rewards functions are like that. One simple example is $R^*(s, a, s') = 0$. Another issue is the hypothesis itself: taking an expert as rational is quite unrealistic. Let's think about a robot that is learning chess. If it observes Magnus Carlsen making a losing move he would assume that Magnus wants to rationally lose.

To overcome the problem of a reward function equal to zero we have to change our approach. Bayes would come in handy. Let D be the observed data, and H_R be the hypothesis that R is the right reward function. The Bayes rules is something like

$$P(H_R|D) = P(D|H_R)P(H_R)$$

$R = 0$ is a very good hypothesis if we consider the prior probability $P(H_R)$ simplicity-based (0 is simple).

$P(D|H_R)$ is infinitesimal if $R = 0$, because it does not explain the choose of the expert: why that particular behavior out of many?

$P(D|H_R)$ will be much larger if R has one only $\pi^*(\mathrm{s})$.

We have not to consider the expert as perfect, he could make some sporadic faults. In this sense we allow $P(D|H_R) \neq 0$ whatever is the value of D (Optimal, Suboptimal or Not-Optimal in reference to R).

So we made the assumption that an agent chooses with a stochastic policy and not with a deterministic policy. This is not done because it represent in a better way human data, but because it allows us to use simpler mathematical methods.

We can find numerous IRL algorithms like Feature Matching, Feature Matching with FLANN or Maximum Entropy IRL.

If an agent understand the expert's actions it can use IRL to learn an optimal or suboptimal policy. Moreover if we have a multiagent environment agents can learn an good policy watching the other policies. This could happen in a cooperative or competitive environment.

A competitive example could be the soccer machine learning: we want to teach to agents how to play soccer. We can give them a set of expert demonstrations, real humans playing the game, then we let them play against each other and they will learn from the expert actions and from the opponent.

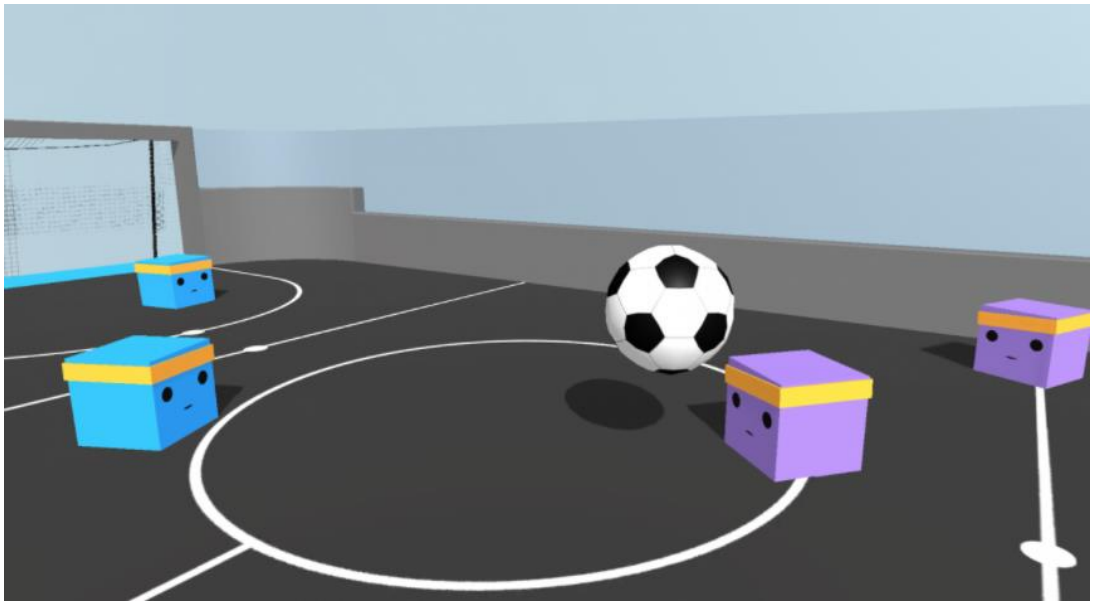


Figure 4: Machine Learning Soccer

A cooperative example could be a car that has to learn to drive in a circuit. We give the agent a set of expert actions, real human racing, and then we let a number N>1 of car running over the circuit. They will try to get an optimal policy from the expert's action and they will help each other to choose the best policy out of the N car racing.

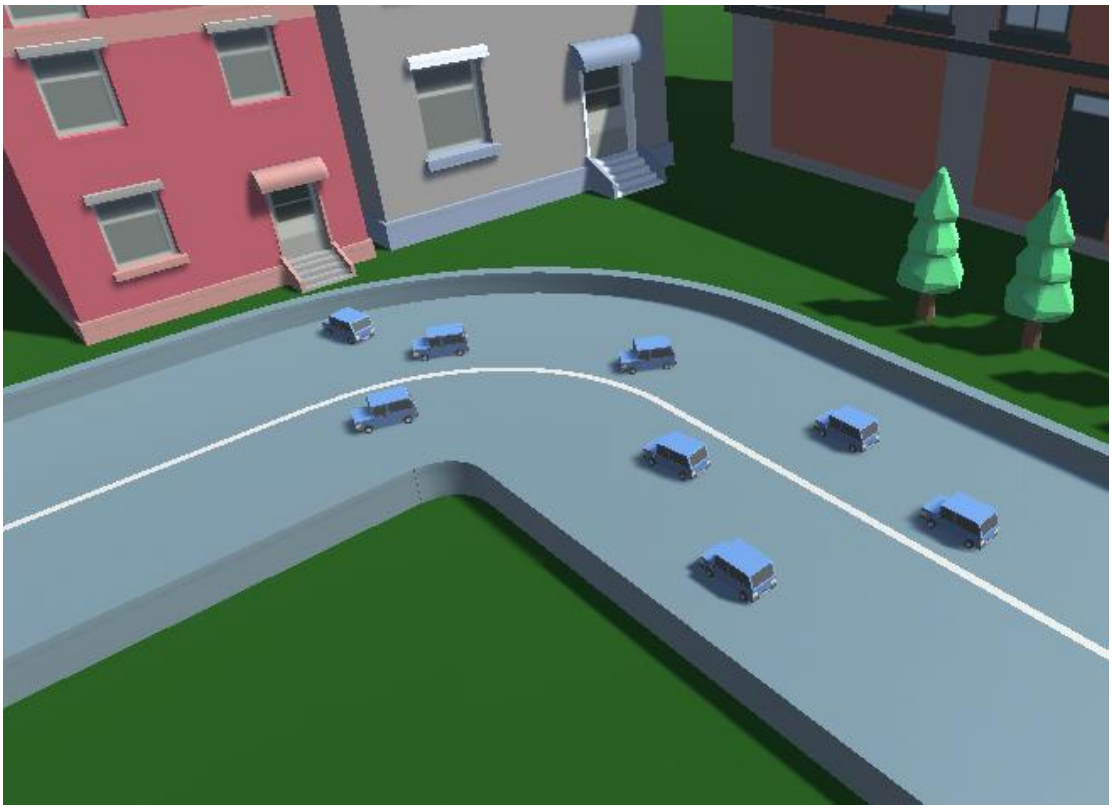


Figure 5: Machicar

The core point in inverse reinforcement learning is that the expert behaves in an optimal or sub-optimal way. This is feasible if the expert does not know that he is been watched. The agent should observe the expert in live action from a hidden camera. This is not a rational situation if the expert knows that a learning agent is watching him.

As an example, think about a robot in driving school that learns from a real human racer how to drive in traffic. The robot is using IRL algorithm and it assume that the teacher is behaving in an optimal way unaware of the agent. But that is not happen: the human racer wants the robot to learn driving methods and tricks in quick way and well enough, so he the human will modify his behavior noticeably. He might stress out the errors to avoid, like accelerating in curves or braking too much without changing gears. He might explain what to do in case of an emergency like a

crash with an obstacle, a fire or a car to car accident. He might drive too carefully or too fast.

These behaviors are not useful and not right when driving alone so inverse RL algorithms will not be able to understand the right reward function. Meanwhile, we need the human to be unaware and the agent to make the right supposition about the expert, it is a cooperative game between machines and humans.

2.3 Curriculum Learning

Animals and humans can learn better and faster if the learning process is composed on examples with a crescent order of difficulties. This process is much better than examples presented at random. These training strategies can be transferred to neural networks and we can call them “curriculum learning”.

In our western society we can say that functional member of the society needs 20 years of training. The learning process is based on a well organized education system: there will be something like a curriculum with different notions at different times, exploiting previously learned notions to facilitate the process of solving new tasks. If we decide the examples and their order we can guide training and accelerate it exceptionally.

Elman in 1993 propose the idea of a curriculum to train a machine in learning grammar. The key point is starting small, learn with simple task or subtasks, and then gently increase the level of difficulty. Elman was using a recurrent network. With his experimental results, Elman showed that innate knowledge of grammar or a massive knowledge of rules is not the most important point in the process of learning correct grammar structures, indeed he presented a better approach: to start with a simpler architecture, restrained in complexity, and then enlarge it as the knowledge enlarges[6].

At an abstract level, a curriculum is a sequence of different lessons. Each lesson of training is associate with a particular set of weights and training examples. In the beginning the weights favor simpler examples. The next lesson is associated will have different weights and examples with an increase of difficulty. In the end of the training the examples will be reweighted uniformly, in this way the last train can be performed on the target training set (or distribution).

Formally, let ξ be a random variable, the learner will use ξ as an example.

Let P(ξ) be the target training distribution, the learner should learn a function of interest from this distribution.

Let $0 \leq W_\delta(\xi) \leq 1$ be the set of weights applied to ξ at step δ in the curriculum progression, with $0 \leq \delta \leq 1$ and $W_1(\xi) = 1$. The corresponding training distribution at step δ is

$$Q_\delta(\xi) \propto W_\delta(\xi)P(\xi) \quad \forall \xi$$

Such that $\int Q_\delta(z)dz = 1$ Then we have

$$Q_1(\xi) = P(\xi)$$

Now let δ increases as a monotonically sequence from 0 to 1

We call the sequence of distribution Q_δ a curriculum if we have an increase of the entropy.

$$H(Q_\delta) < H(Q_{\delta+\epsilon}) \quad \forall \epsilon > 0$$

And $W_\delta(z)$ is monotonically increasing in δ, that is

$$W_{\delta+\epsilon}(\xi) > W_\delta(\xi) \quad \forall \epsilon, \xi > 0$$

To explain better this definition we can consider a simple setting where Q_δ is focused on a limited set of examples and increasing δ implies including new examples into the set. The reinforce of Q_δ increases with δ and the progression of

training distributions corresponds to a sequence of embedded training sets. It all starts with a little set (simpler example) and we continue until the end where we have the target of training set. We desire to see an increase of entropy with the increase of diversity of training examples. And we want the weights of particular examples to increase as they get included in the training set[7].

A previous research on the effectiveness of Curriculum Learning was made by Zhipeng Ren et al. They compare three different algorithms to play 12 different Atari 2600 games. The algorithm considered are Deep Q Network(DQN), Prioritized Experience Replay(PER) and Deep Curriculum Reinforcement Learning (DCRL). The research shows that DCRL outperform the other two algorithms[8]

3 Methodology

3.1 Software and Libraries Used

My goal was to implement in practice three different learning algorithm: Reinforcement, Imitation and Curriculum. In order to achieve this, I created a 3D racing game. In this game all the AI will have a different Neural Network trained with a different learning technique. The player will compete against this AI in a race. To implement and develop the game I used Unity Engine.

Unity is a very powerful engine to develop videogame in 2D or 3D. It has a primary scripting API in C#. For 2D it allows the importation of sprites and an advanced 2D world renderer. For 3D games Unity can have texture compression, mipmaps and resolution settings for each platform that the game engine supports. It also provides supports for bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion and shadow maps. Unity is used to develop almost 50 percent of the mobile games on the market and 60 percent of the augmented and virtual reality content. It can be used also in the development of movie, automotive, architecture, engineering and construction.

To develop a machine learning game I had to use a machine learning library. Unity fitted my needs with its Unity ML-Agents

Unity ML-agents is an open source library that enables games and simulations to serve as environment to training machine learning agents. The agent can be trained using reinforcement learning, imitation learning, curriculum learning or other ML methods through a Python API.

The toolkit is composed of 5 different elements: A training Environment, a Python Low-Level API, the External Communicator, the Python trainer and the Gym Wrapper. The training environment is the scene where we built our own game. Here the agents observe and take actions. The Python Low-Level API is an external tool

that manipulates the machine learning brain during training. The external Communicator communicates with the Python API and the learning environment. The python trainer lives in the Python API and runs all the algorithms involved in training.

To plot all the graphs about the performance of my Neural Net I used TensorFlow, an open source software library for dataflow and differentiable programming. It is a symbolic math library, and it can be used for machine learning application such as neural networks

The combination of Unity, the unity ml agents library and the TensorFlow library is the toolkit I used to develop my game

To test my AI and plot the metrics I used MATLAB software

3.2 The Game Scenario: Machicar (Machine Learning Car)

As mention before, the player will compete against different AI all trained with a machine learning approach. The objective of the game is to complete 3, 5 or 7 laps (decided by user) and to be the first between all the cars. The game has 3 tracks created by a human and 10 tracks randomly generated by an algorithm. Every track has walls on both sides of the street. In every track we have a few obstacles. If the car (player or AI) hit a wall or an obstacle a crash happens.

Every track has a number of checkpoints alongside the road. Whenever the AI or the player crash they will restart from the last checkpoint passed.

In the beginning of the track we have a Start Line. When the player or the AI passed the Start Line for the first time a timer starts. When the player or the AI passed the Start Line for the second time the timer is stopped and recorded, this will be the first

lap time of the car. In the end the timer is restarted, and it will record the second lap time.

3.2.1 Level Design

- The first track (designed by a human) is elliptical composed of four 90 degree turns. It has 11 checkpoints

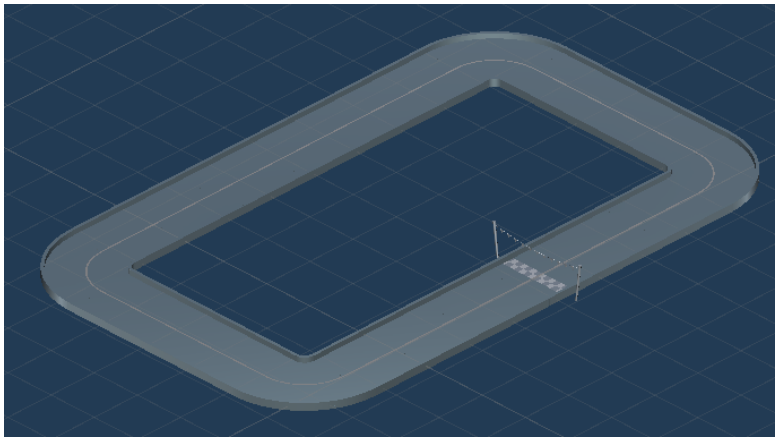


Figure 6: Track 1

- The second track (designed by human) has more turns, and it is longer: six 90 degrees turns, and a long 180 degree turns. It has 17 checkpoints

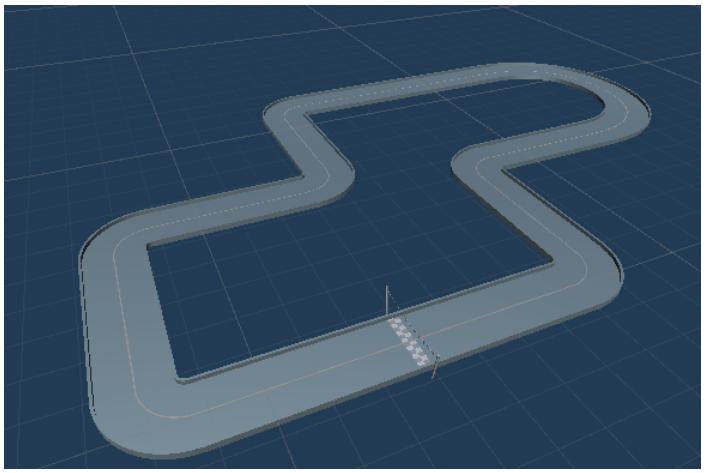


Figure 7: Track 2

The third track (designed by human) is the most complicated, composed of ten 90 degrees turn, one long 180 degree turn and two shorts 180 degree turn

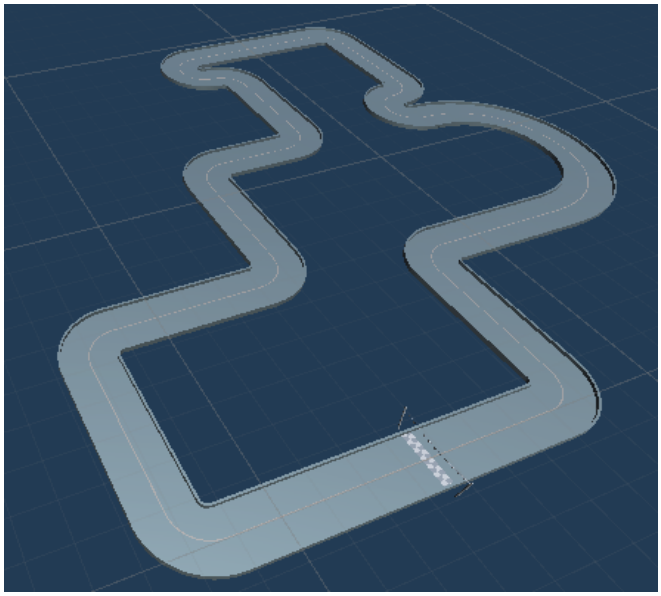


Figure 8: Track 3

These first three tracks were assembled using 4 prefabs from the unity karting microgame free asset

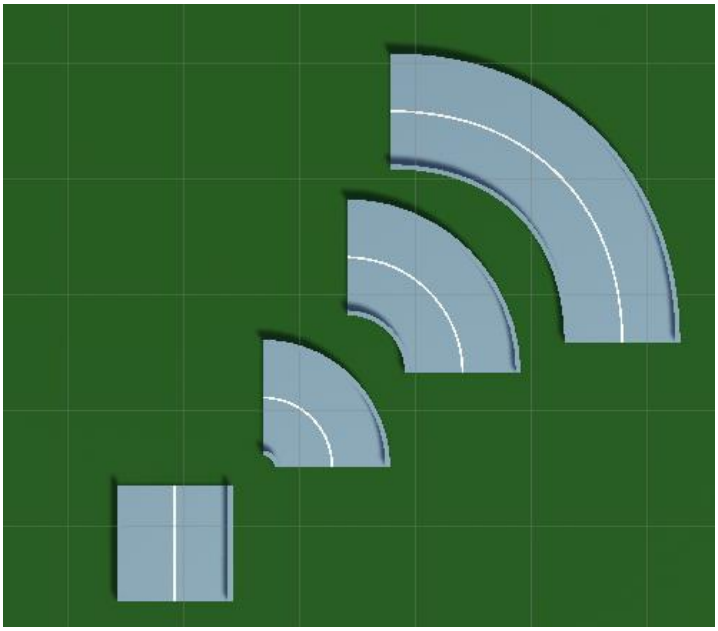


Figure 9: The track parts

-The other 7 tracks are generated using an algorithm developed by Ian Hudson[9] modified by me

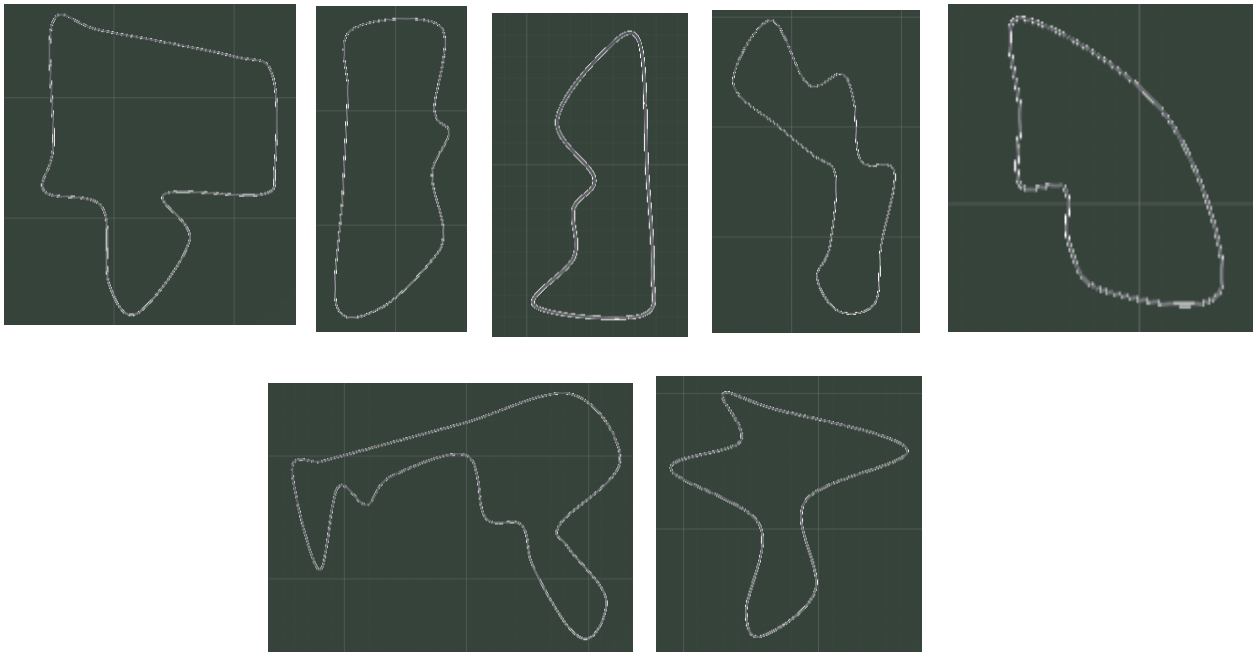


Figure 10: Seven random generated tracks

The algorithm that randomly generate the track is composed by three steps.

1) Create a Voronoi Diagram. It is a partitioning of a plane with n points into convex polygons such that each polygon contains exactly one generating point and every point in a given polygon is closer to its generating point than to any other

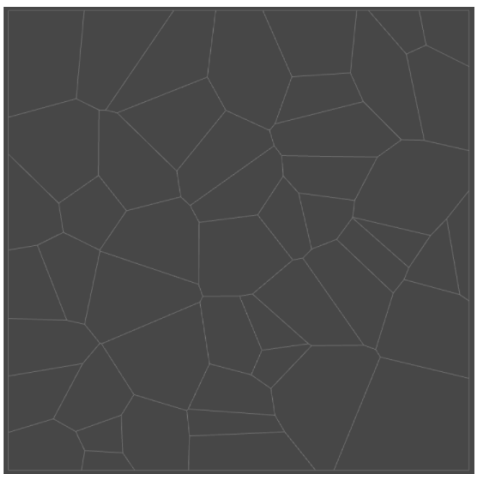


Figure 11: Voronoi diagram

2) Create the spline that will be track. To do this the tool selects a random cell from the Voronoi diagram, then selects one of its edges. From the edge we select the cell that contains the edge. This process continues until a certain number of cells are chosen. After all the cells are chosen the tool gets the outer edge of all the selected cells. This gives us a spline. The tool applies a smooth to the spline. This will be the curve to build the track from.

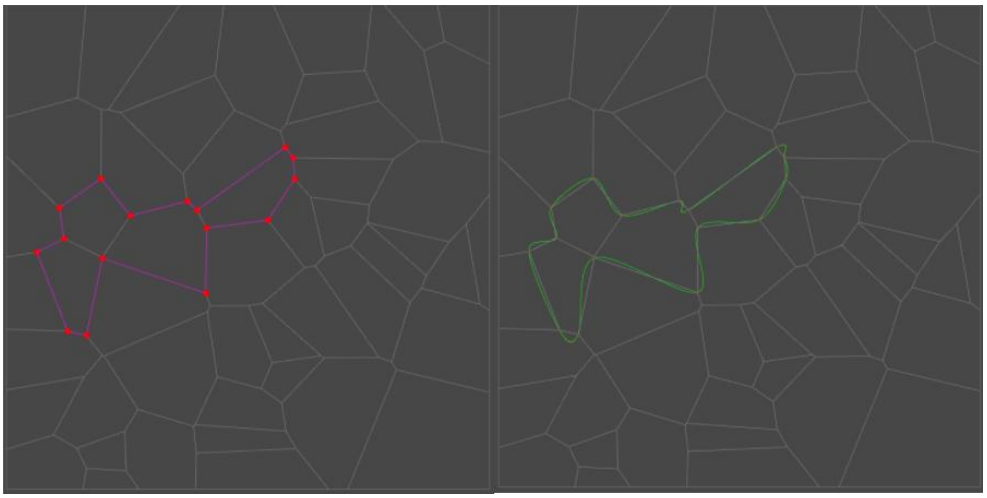


Figure 12: The generated spline

3) The final step is to build the actual mesh of the track. This is achieved by adding vertices to both part of the spline.

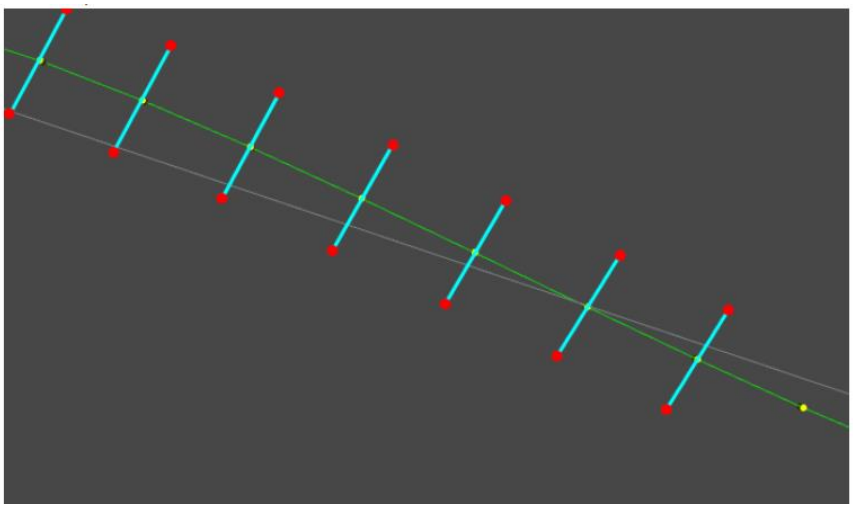


Figure 13: Vertices at the side

Here I changed the code of Ian Hudson. I got the oPoints (oriented points) of the spline. The oPoints are the one in the middle between the two red dots in the image above. They are oriented with the z axis tangent to the spline, the x axis perpendicular to the spline and the y axis perpendicular and pointing up. The z axis of the oPoint give us the track direction (clockwise). In the code I cycled through all the oPoints, at both side of each oPoints I instanciated a wall prefab. Every fifty oPoints I instaciated a checkpoint in the middle. At the first oPoint I instaciated the Starline.

Below we can see a snippet of the code I insert in the SplineCreator.cs file

```
//...
//cycle through all the oPoints
for (int i = 0; i < oPoints.Length; i++)
{
    //at the first oPoints I instantiate the startLine
    if (i == 0)
    {
        GameObject sl = Instantiate(_startLine, oPoints[i].Position, oPoints[i].Rotation);
        sl.transform.Rotate(0, 90, 0);
    }


       //every 150 oPoints I instatiate a checkPoint
    if((i % 150) == 0) {
        if(i!= 0) {
            GameObject checki = Instantiate(_checkPoint, oPoints[i].Position,
oPoints[i].Rotation);
            checki.transform.Rotate(0, 90, 0);
        }
    }


      //I cycle through the two points at the side of the oPoint
    for (int j = 0; j<pTwoDShape.Vertices.Count; j++)
    {
        Vector3 fwd = oPoints[i].Rotation * Vector3.forward;
        Vector3 pos = oPoints[i].Position + (pTwoDShape.Vertices[j].x * fwd);
        //in this way I got the exact position of the points at the side of the oPoints
        pos.y = oPoints[i].Position.y + pTwoDShape.Vertices[j].y;
        //pos is the position of one of these points
        Instantiate(_wally, pos, oPoints[i].Rotation);//I instatiate the wall
        //...
    }
}
//...
```

In this way the tracks are generated with a beginning (the Start Line) a series of waypoints (checkpoints) and borders on the side.

In the end the code Ian defined each vertex has an UV coordinate. In this way the mesh can have a texture applied to it. After all the vertices have been defined, the tool creates triangles and so the mesh is ready.

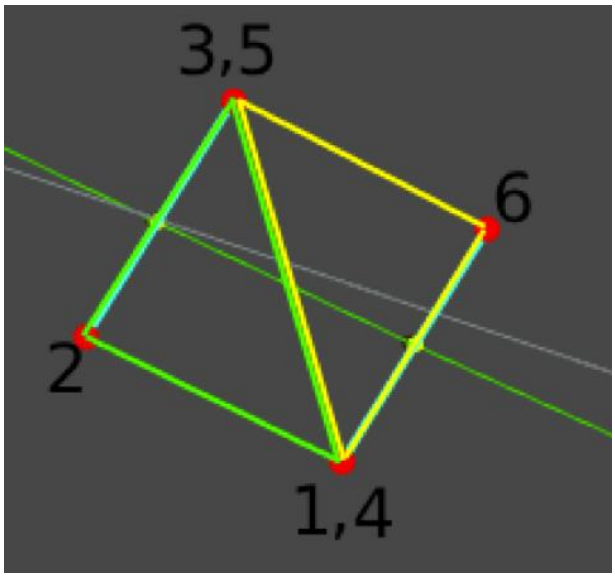


Figure 14: UV coordinates generated

3.2.2 Obstacles

Around every track are randomly placed 4 kind of obstacles

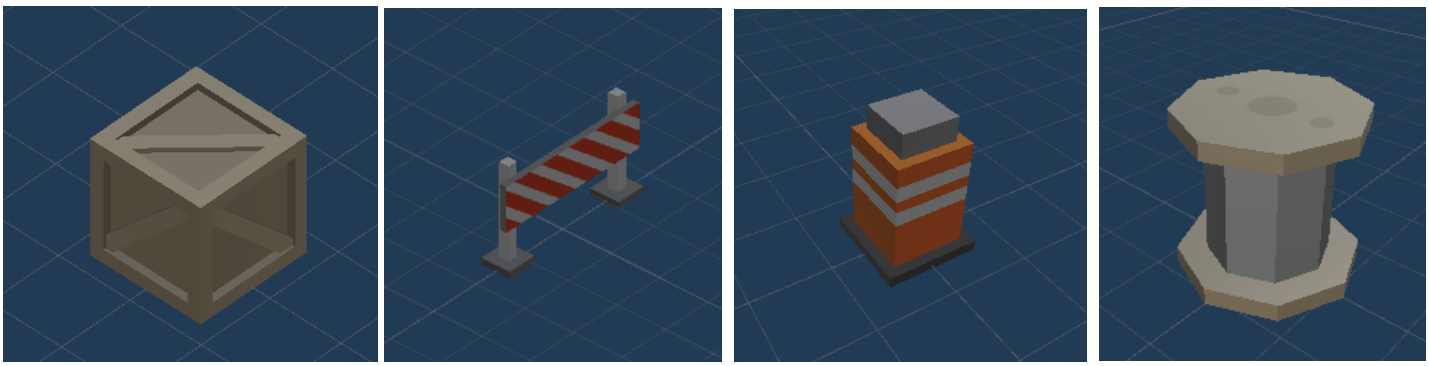


Figure 15: 4 different obstacles

3.2.3 Player Control

The player controls the car with 4 buttons:

“W” increases the throttle value and move forward

“S” decreases the throttle value and move backward

“A” decreases the turn value and move left

“D” increases the turn value and move right

3.2.4 Neural Network For AI

The cars’ AI are neither hard scripted nor have a Finite State Machine system. They have been trained with ML technique and with the same brain they can drive alongside all the tracks. How do I achieve this goal? How the same AI can be implemented to work on different tracks with a good performance? Neural Networks (NN) is the answer. Train a neural network with a good training process can give you a brain (AI) that has a good performance on different tracks and it can be challenging to play against the player.

A neural network is composed of an input layer, one or more hidden layers and an output layer. Each car in the game has the same NN composed by a layer of 16 unit in input 2 hidden layers of 128 unit and one unit in the output layer.

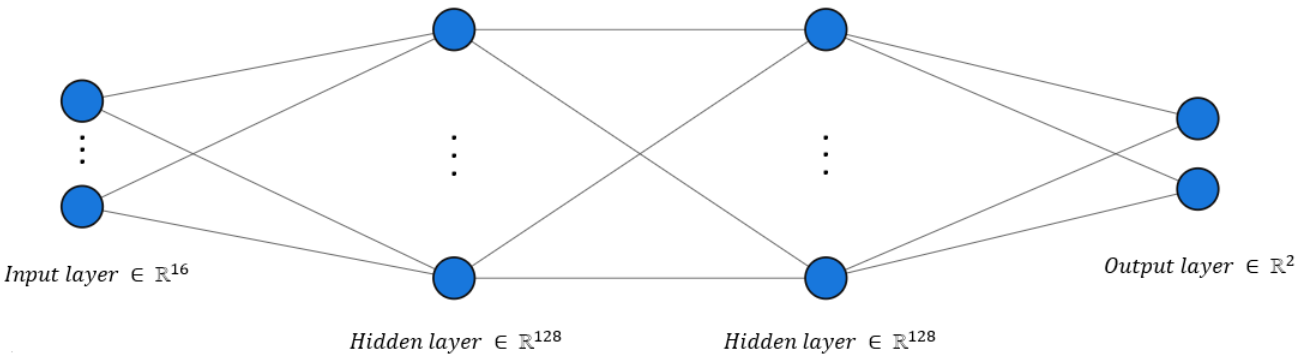


Figure 16: Car's neural network

Intuitive explanation

What are these numbers? The 16 inputs unit are the distances calculated by the raycasts. A raycast is a ray casted from an origin, going towards a direction with a fixed length. In my case the rays start from the car and go in 16 different direction with a certain angle around the y axis (0, 10, 20, 45, 90, 135, 160, 170, 180, -10, -20, -45, -90, -135, -160, -170 degrees). Their length is 20 meters (or unity's unit). The raycasts go all around the car in order to perceive all the obstacles around it. If the raycast hits a car, a wall or an obstacle the neural network receives the distance from this object. If ray does not hit an object the neural net receives the value 20 (maximum distance) from that ray

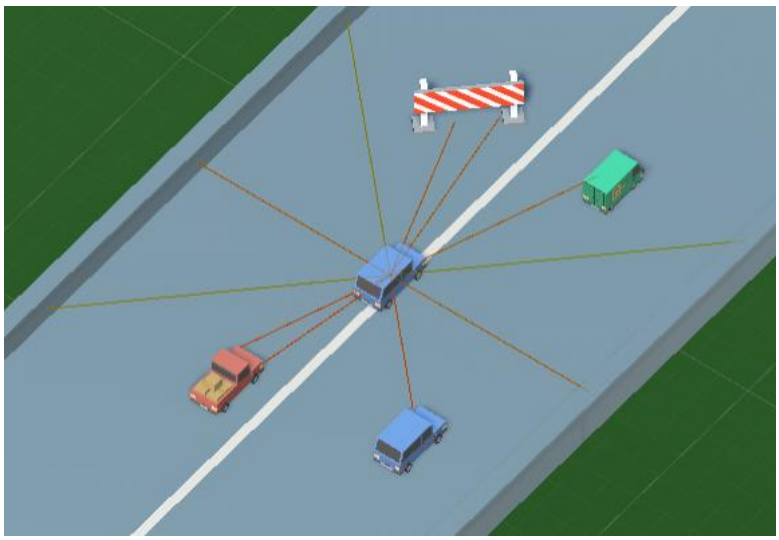


Figure 17: The raycasts around the car

The hidden layers make all the calculation to give us an output that maximize the reward. The output layer consist of two nodes: the throttle value, varying from -1 (full brake) to +1(full throttle); the turn value, varying from -1 (45 degrees turn on the left on the steering wheel) to +1 (45 degrees turn on the right on the steering wheel).

The reward function gives positive value if the car has a positive throttle value. It gives negative value if the car hit a wall or an obstacle or a car.

During the process of training we will form a policy π that maps from state s to the best action a. The policy is the strategy that the agent employs to determine the next action based on the current state.

Formally

The action space A defined as follow

$A = (a_1, a_2)\ with\ (a_1, a_2) \in \mathbb{R} \cap [-1, +1]$

(a_1 is the throttle value and a_2 is the turn value)

The state space S is defined as follow

$S = (s_1, \dots, s_{16})\ with\ (s_1, \dots, s_{16}) \in \mathbb{R} \cap [0, +20]$

These are the raycasts shoot at a maximum 20 units of distance

The reward function R is

$-0.5 \quad if (s_1, \dots, s_{16}) < 1.3$ (the car is too close to a wall, an obstacle or a car)

+ $a_1 * 0.005 \quad if\ a_1 > 0$ (the car has a positive throttle value)

$+0.1\ if\ the\ car\ passes\ through\ a\ checkpoint$

$+1\ if\ the\ car\ finishes\ a\ lap$

3.2.5 The Policy Update, Policy Gradient Method And PPO

To update the policy π in my project I use a policy gradient method called PPO, Proximal Policy Optimization. We have to define what is a policy gradient method first.

Policy Gradient Methods are a subclass of Policy-Based methods that evaluate an optimal policy's weights using the gradient ascent algorithm. This algorithm starts with guessing the value of the weights that maximize the expected return. Then it evaluates the gradient at that point that indicates the direction of the steepest increase of the function of expected return, and so it makes a small step towards that direction. The algorithm loop through this process until it reaches the maximum expected returns.

The family of Policy Gradient Method are wide, in literature we can find several examples of those: Trust Region Policy Optimization (TRPO), Advantage Actor Critic (A2C), Cross-Entropy Method (CEM), Vanilla PG (VPG).

A new Policy Gradient Method was presented in 2017: Proximal Policy Optimization (PPO).

PPO tries to find the best improvement step on a policy using the data available, without stepping so far and cause a performance collapse. PPO is a family of first order methods that use some workaround to keep the new policies not far from the old one. The variant I use was PPO-Clip. This one relies on a particular clipping in the objective function and it removes stimuli for the new policy to get far from the old one. Here I explain the algorithm as it was presented by openAI.

"PPO-clip updates policies with the parameter θ

$$\theta_{k+1} = arg\max_{\theta} E\left[L^{CLIP}(\theta)\right]$$

Where L^{CLIP} is the surrogate clipped objective function and it can be defined as

$$L^{CLIP}(\theta) = \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t, clip\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)A_t\right) \quad (23)$$

Where ϵ is a small hyperparameter that indicates how much the new policy is getting far from the old one. A_t is an estimator of the advantage of the new policy at timestep t. $\pi_\theta(a_t|s_t)$ is the new stochastic policy at timestep t. $\pi_{\theta_{old}}(a_t|s_t)$ is the old stochastic policy at timestep t.

(23) can be rewrite in a simplified version like that

$$L^{CLIP}(\theta) = \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t, g(\epsilon, A_t)\right)$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & if\ A \geq 0 \\ (1-\epsilon)A & otherwise \end{cases}$$

Now we take the two separate cases: A $\geq$ 0, we have a positive advantage with the new policy; A $<$ 0 we have a negative advantage

$if\ A \geq 0$ The objective function reduces to

$$L^{CLIP}(\theta) = \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1+\epsilon\right)A_t$$

Because the advantage is positive the objective function will increase if $\pi_\theta(a_t|s_t)$ increases. The min in this function puts a limit on how much the objective will increase. If $\pi_\theta(a_t|s_t) > (1+\epsilon)\pi_{\theta_{old}}(a_t|s_t)$ the min takes action and this term hits the ceiling of $(1+\epsilon)A_t$. In this way the new policy will not benefit by going too far from the old one.

$if\ A < 0$ The objective function reduces to

$$L^{CLIP}(\theta) = \max\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon\right)A_t$$

Because the advantage is negative the objective function will increase if $\pi_\theta(a_t|s_t)$ decreases. The max in this function puts a limit on how much the objective will increase. If $\pi_\theta(a_t|s_t) < (1-\epsilon)\pi_{\theta_{old}}(a_t|s_t)$ the max takes action and this term hits the ceiling of $(1-\epsilon)A_t$. Again the new policy will not benefit by going too far from the old one.

That is why this algorithm is called Proximal Policy Optimization, because we are moving from an old policy to a proximal new policy. PPO achieves this disadvantaging new policy from going too far from the old one.

PPO-Clip Pseudocode

1. Input: initial policy parameters θ_0, initial value function parameters Ω_0

2. for i = 0,1,2, … do

3. Collect set of trajectories $D_i = \{\tau_k\}$ by running policy π_i in the environment

4. Compute rewards R_t

5. Compute advantage estimator A_t based on the current value function V_{Ω_i}

6. Update the policy by maximizing the PPO objective

$$\theta_{i+1} = argmax \frac{1}{|D_i|T}\sum_{\tau\in D_i}\sum_{t=0}^{T}\min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t, g(\epsilon, A_t)\right)$$

via stochastic gradient ascent

7. Fit value function by regression on mean squared error:

$$\Omega_{i+1} = arg\min_{\Omega}\frac{1}{|D_i|T}\sum_{\tau\in D_i}\sum_{t=0}^{T} min(V_\Omega(s_t - R_t)^2$$

via some gradient descent algorithm

8. endfor”[10][11]

3.2.6 The Agent Training and Implementation

The unity ml-agents library offers you a powerful tool for developing machine learning application. We have to follow a workflow in order to make everything work. We are skipping the detail of the setup that you can find here https://github.com/Unity-Technologies/ml-agents/blob/release_9_docs/docs/Installation.md

In unity ml we have a few crucial components on the timeline:

The episodes are composed by multiple steps. At each step the agent collects observations, send these to the policy and receives a vector of actions in response, in the end we have the calculation of the reward.

The "Decision Requester" let you decide the decision period, or rather how long each tuple observation-action-reward must be in term of steps. It also let you decide if your agent can take actions in between steps. In my case I have

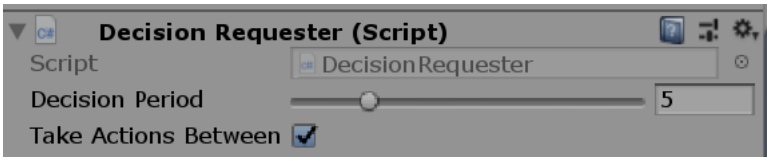


Figure 18: Decision Requester

So every 5 steps we have a full cycle observation-action-reward, and the agent can take actions in between.

My episodes do not have a fixed step length, meanwhile they end when certain conditions happen. I have set a maximum number of steps in the training process, when the agent reaches the max step number the training stops.

The workflow I follow to develop my agents is the following:

1 Implement the Agent in c#

2 Set the unity environment and the parameter of unity's hierarchy

3 Set the hyperparameter in an external .yaml file

4 Launch the training from a command prompt

1) First I scripted “MyAgent” derived from the superclass Agent (this is one of the main class of the library).

```
public class MyAgent : Agent
```

Then I override inside the class the four main functions for an Agent:

```
public override void OnEpisodeBegin(){...}

public override void CollectObservation(VectorSensor sensor){...}

public override void OnActionReceived(float[] vectorAction){...}

public override void Heuristic(float[] actionsOut){...}
```

The first method OnEpisodeBegin() set up the Agent instance at the beginning of an episode. In my case I restart some variables that belong to the car: the velocity, the angular velocity, the throttle value, the steer value is all set to zero. If the car has finished a full lap, the episode is ended and the car restart from the initial position. The position of the car is reset to the initial position if it has not passed even a checkpoint, it is reset to the position of the last checkpoint passed otherwise.

```
public override void OnEpisodeBegin()
{
    if (hasFinishedLap)
    {
        this.transform.localPosition = startpos;
        this.transform.rotation = startrot;
        hasFinishedLap = false;
        StartLineFlag = false;
    }
    if ((hasCollided || lapOverTime > overTime))
    {
        hasFinishedLap = false;
        StartLineFlag = false;
        GetComponent<Rigidbody>().velocity = Vector3.zero;
        GetComponent<Rigidbody>().angularVelocity = Vector3.zero;
        action_throttle = 0;
        action_steer = 0;
        hasCollided = false;

        if (lastCheckpointPassed != null)
        {
            this.transform.localPosition = lastCheckpointPassed.position;
            this.transform.rotation = lastCheckpointPassed.rotation;
        }
        else
        {
            this.transform.localPosition = startpos;
            this.transform.rotation = startrot;
        }
    }
}
```

The second method CollectObservation(VectorSensor sensor){...} takes a VectorSensor as argument, these are the data structure that unity ml uses to take observation from the environment. We can take observation as float, int, Vector3 and other datatypes. In my case I use float observations. At every step, the function cycle through an array of raycasts. Every ray is shot in a different direction (around the car) and if one of these hit something that belongs to the wallObstacleCarLayers (if it hit a wall, an obstacle or a car) at raycastDistance (20 units) the distance is registered as a float and sent to the neural network with the function addOservation(float observation). If the distance is less than 1.3f a collision is registered (hasCollided = true).

```
public override void CollectObservations(VectorSensor sensor)
{
    for(int i = 0; i < raycasts.Length; i++)
    {
        RaycastHit hitInfo = new RaycastHit();
        var hit = Physics.Raycast(raycasts[i].position, raycasts[i].forward,
out hitInfo, raycastDistance, wallObstacleCarRaycastLayers.value,
QueryTriggerInteraction.Ignore);
        var distance = hitInfo.distance;
        float distancef = distance;
        if (!hit) distance = raycastDistance;
        var obs = distance / raycastDistance;
        float obsf = obs;
        sensor.AddObservation(obsf);
        if (distance < 1.3f)
        {
            hasCollided = true;
        }
}
```

The third method OnActionReceived(float[] vectorAction) takes a vector of float as argument. This is the response of the neural network after the policy received the observations. As told before the float values are two: the throttle value and the steering value that belong to the continuous interval [-1, +1]. This two float values tare assigned to the variables action_steer and action_throttle. These two variables are read by an external script (PM) that is responsible of the car movement. In OnActionReceived I also set the rewards. If a collision happened (hasCollided == true) the EndEpisode function is called and a negative reward is assigned. If the throttle value is positive a little positive reward is assigned. If a checkpoint is passed a positive reward is assigned. If the lap has finished a positive reward is assigned and the EndEpisode() function is called.

```
public override void OnActionReceived(float[] vectorAction)
{
    action_steer = Mathf.Clamp(vectorAction[0], -1f, 1f);
    action_throttle = Mathf.Clamp(vectorAction[1], -1f, 1f);

    if (hasCollided)
    {
        AddReward(-0.5f);
        EndEpisode();
    }

    if (action_throttle > 0)
    {
        AddReward(action_throttle * (0.005f));
    }

    if (hasPassedOneCheckpoint)
    {
        AddReward(0.1f);
        hasPassedOneCheckpoint = false;
    }

    if (hasFinishedLap)
    {
        AddReward(+1);
        EndEpisode();
    }
}
```

The last method Heuristic(float[] actionsOut) takes as argument a vector of float, this vector is the actions registered by the function itself. The heuristic provides custom decision making logic or manual control of an agent using keyboard, mouse, or game controller input. We can say that the Heuristic is the human control logic. In unity hierarchy we can set the Behavior Type to be Heuristic, in this way our agent will follow the command that we implement in the Heuristic() function. This is mandatory to implement and it can help the developers in debugging and before training to see if the agent can perform the action that we want to teach him.

In my case the heuristic is a WASD style control: W/S to increment/decrement the throttle value, A/D to increment/decrement the steering value.

```
public override void Heuristic(float[] actionsOut)
{
    actionsOut[0] = Input.GetAxis("Horizontal");
    actionsOut[1] = Input.GetAxis("Vertical");
}
```

2) We have to set now our environment in Unity. This will be the gym where our agent will be trained. In my case I put the agent behind the Start Line of the track. The first track where I trained the agent is RandomTrack1

Every Neural Network will have a MyAgent_PM script (Player Movement). This is the script responsible for the car movement. It will read the Throttle and Steer value from the Agent script and will make move the car with the max Speed of 10 and the max Turn Speed of 100 {change script “MyAgentPM”}

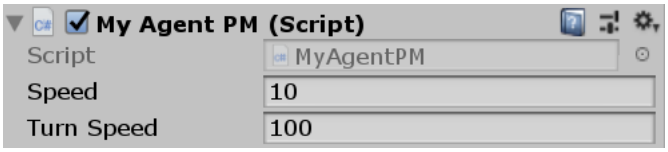


Figure 19: Car movement script

We have then the MyAgent Script. Here we have to set the raycast layer. Every GameObject that the raycasts can perceive must belong to this layer. In our case these are the Wall, Car and Obstacles. In the end we have the raycast array. This must be set to 16 because 16 are the raycasts that we assign from the hierarchy to the inspector.

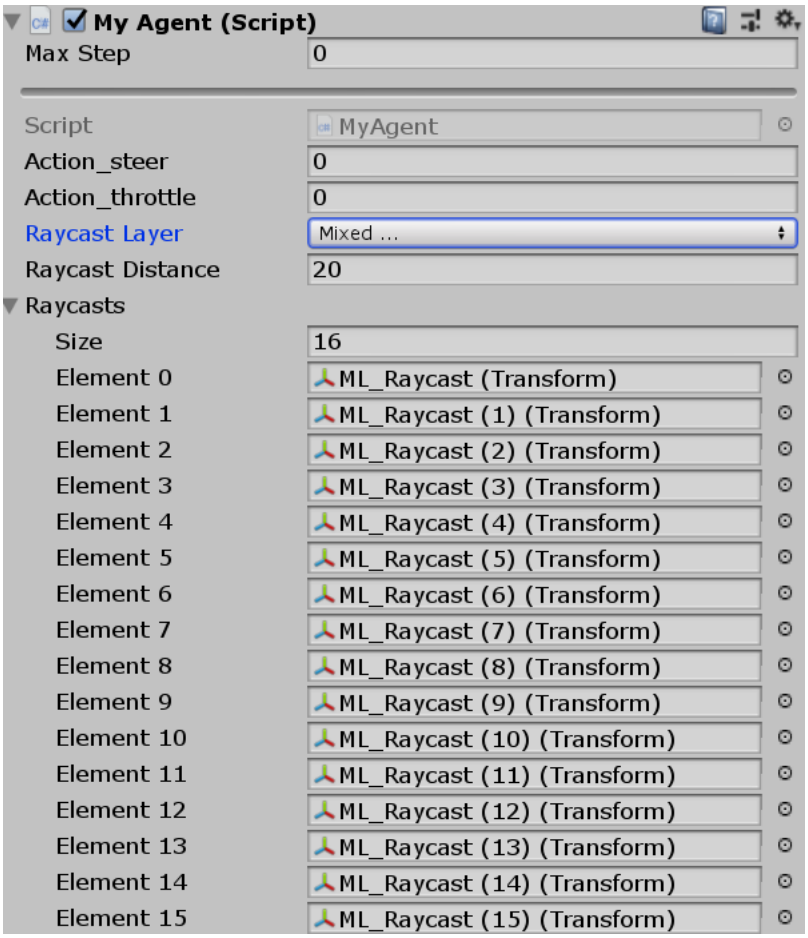


Figure 20: Car Agent script

In the end we have the Behavior Parameters script. Here we set the name of the Behavior we are going to train (MyAgentBehavior). Then we set the Space Size of the vector Observation: 16 like the number of the raycasts. The Vector Action section is about the Actions that the policy will send to the agent after observations are collected. Our space size is 2 (throttle value and steer value) and the space type is continuous because the values are float between -1 and +1. When we perform a training the Behavior Type must be set to default and the Model must be empty. We have to train each brain from scratch. This is a big limitation of unity ml agents that is work in progress and maybe it will see the light in a future update

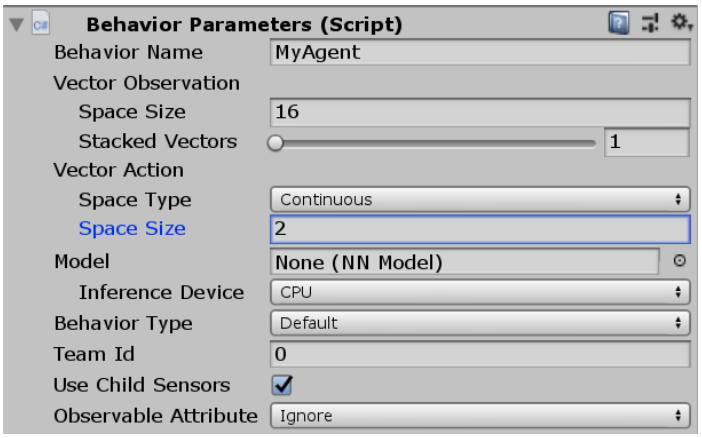


Figure 21: Car Neural Network script

3) After the scripting and the unity parametrization part are done we can focus on the setting of the hyperparameters in the yaml file. This part will be different for the three kind of neural network. Now we give a generalize explanation of the file. In the Unity-ml agents folder we have a config folder. Here we can find or create all the yaml files that will contain the hyperparameter of the NN. In my case I add a section "MyAgentBehavior" in the trainer_config.yaml file. The name of the behavior must be the same exact name of the one we put in the Behavior Parameters script before.

```
    MyAgentBehavior:
        trainer: ppo
        batch_size: 1024
        beta: 1e-3
        buffer_size: 10240
        epsilon: 0.2
        hidden_units: 128
        lambd: 0.95
        learning_rate: 3.0e-4
        learning_rate_schedule: linear
        max_steps: 15.0e5
        memory_size: 128
        normalize: true
        num_epoch: 3
        num_layers: 2
        time_horizon: 64
        sequence_length: 64
        summary_freq: 10000
        reward_signals:
            extrinsic:
                strength: 1.0
                gamma: 0.99
```

The batch_size is the number of experiences in each iteration of gradient descent. The beta is the strength of the entropy regularization, which makes the policy "more random”.

The buffer_size for the PPO is the number of experiences to collect before updating the policy model. It can be explained as how many experiences should be collected before we do any learning or updating of the model.

The epsilon influences how rapidly the policy can evolve during training. It is the threshold of divergence between the old and the new policies during gradient descent updating.

The hidden units are the number of units in the hidden layers of the NN.

The lambd is the regularization parameter γ used when calculating the Generalized Advantage Estimate. It can be explained as how much the agent relies on its current value estimate when calculating an updated value estimate

The learning_rate is the inizialization rate for gradient descent. It is the strength of each gradient descent update step.

The learning_rate_schedule indicates how much the learning rate will change over time

The memory_size is the size of the memory that an agent must keep. This coincides to the size of the array of floating point numbers used to store the hidden state of the recurrent neural network of the policy.

The normalize is a bool value that if set to true will give the agent the command to normalize the vector observation input.

The num_epoch is the number of passes to make through the experience buffer when performing gradient descent optimization

The num_layers is the number of hidden layers in the NN

The time_horizon indicates how many steps of experience to collect per-agent before adding it to the experience buffer. When this limit is reached before the end of an episode, a value estimation is used to predict the total expected reward from the agent's current state.

The sequence_length defines how long the sequences of experiences must be while training

The summary_freq is the number of experiences that needs to be collected before generating and displaying training statistics.

The reward signal section can be extrinsic, curiosity or GAIL (in my case I used only extrinsic and GAIL). This will determine which kind of reward signal are you using. In every case the reward signal must have a strength and gamma section.

The strength indicates the factor by which to multiply the reward given by the environment

The gamma is the discount factor for future rewards coming from the environment. This can be imagined as how far into the future the agent should care about possible rewards.

4) After all this setup has been made we can launch the training. We must open a command prompt, navigate into the unity-ml agents folder, activate the virtual environment, and launch the training process with an id. Below we can see the list of commands in the cmd.

```
C:\>cd ML-Agents

C:\ML-Agents>python-envs\sample-env\Scripts\activate

(sample-env) C:\ML-Agents>mlagents-learn config/trainer_config.yaml --run-id=MyAgent
```

Figure 22: Command prompt training command

After pressing Enter If everything has done properly the unity logo should appears in the command prompt and a message should say “Press Play in Unity”. Now we press play in unity and we let the agent train for a max_step amount of time.

In my case the max_step was 1.5 million. Every 0.5 million I stop the process with CTRL-C in the command prompt. I change the track in the unity scene, and I resume the training from the command prompt with the –resume command. In this way I could train each NN in 3 different tracks (RandomTrack1, RandomTrack2 and RandomTrack3).

When a number max_step of steps is completed the Unity Environment should be stopped and a .NN file should be exported. This is our Neural Network. We take this file, and we drag it to the Model section of the Behavior Parameters script in unity. Now we can press play and see how our Neural Network perform in our environment.

3.3 The Learning Methods

We introduced the car’s neutral network, the algorithm behind their policy, the training process workflow. These are all common features for all the three Neural Networks. We are going now to explain the learning methods in all his differences

3.3.1 Reinforcement Learning Implemented

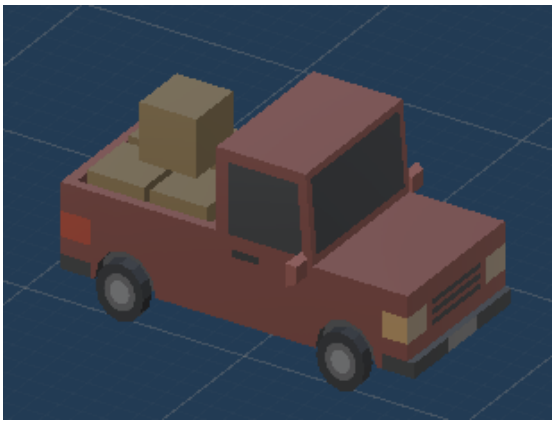


Figure 23: RL car

CARL (Car Agent Reinforcement Learning) is the first neural network I train.

For this neural network I followed the exact same process explained in chapter 3.2.6. 3 random tracks, 0.5 million step each. I change the trainer_config.yaml file. Here I create a CARL section where I set up the hyperparameters (the same as the one in chapter 3.2.6).

In the end I launch the training. The result of TensorFlow are shown below.

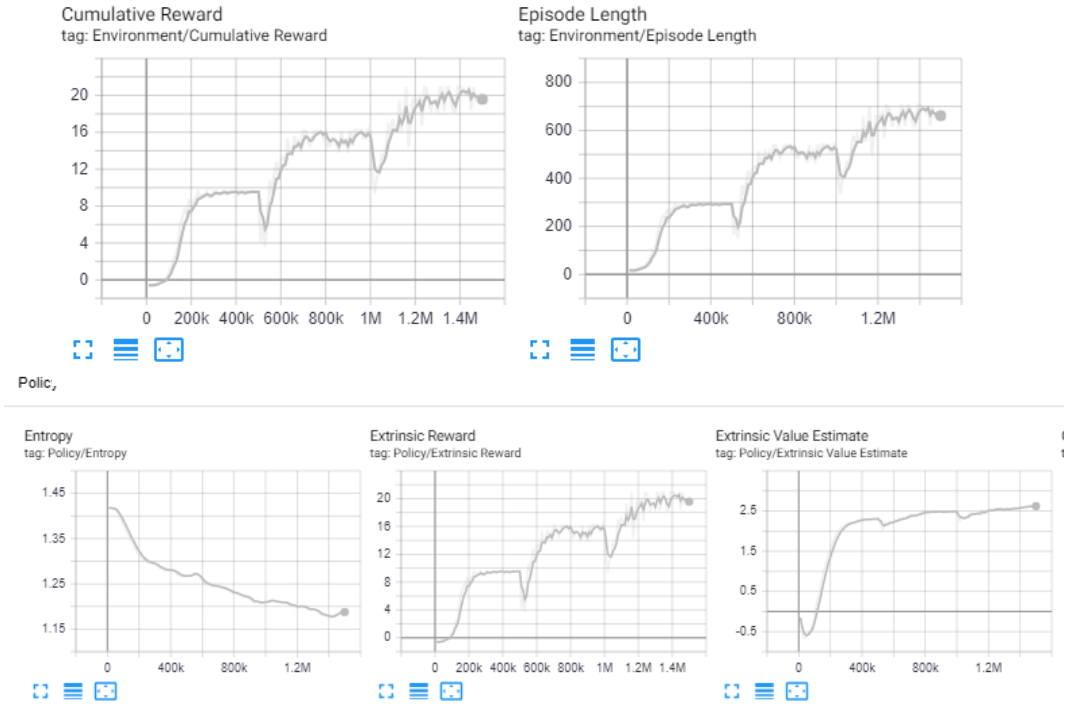


Figure 24: RL training graphs

We can notice from the graph that the “Cumulative Rewards” (the mean cumulative episode reward) grows until a certain threshold and then it becomes stable. The same behavior regards “Episode Length” (the mean length of each episodes, measured in step). That is because the car has learnt how to drive a full lap. Every 500k steps there is a gap and then the graphs become stable again. That is the track has changed.

The Entropy shows how random the decisions of the model are. It should slowly decrease during a successful training process and it indeed decreases. The Extrinsic Reward corresponds to the mean cumulative reward received from the environment per-episode. It increases in the exact same way of the cumulative rewards, that is because this agent is based only on extrinsic reward. The mean value estimate for all states visited by the agent. It should increase during a successful training session and it indeed increases.

3.3.2 Imitation Learning Implemented

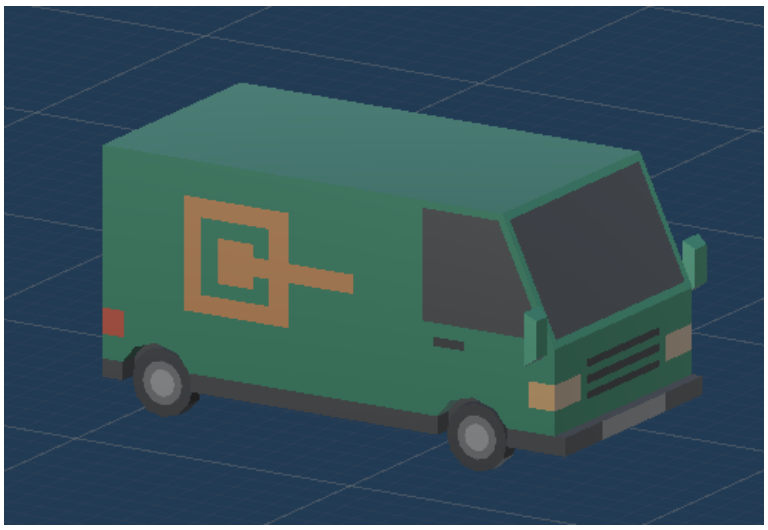


Figure 25: IL car

CAIL (Car Agent Imitation Learning) is the second neural network that came in my mind. Unity ML propose this kind of learning techniques from the version 0.3. Here it is called behavioral cloning and it is based on a simple paradigm: register yourself

as human to perform a task and than the agent will learn from you. The problem of unity's behavioral cloning is that the agent will only learn from human demonstration. If the human demonstrations came from a trackX the neural network will copy the trajectories and it will learn to drive in the trackX as the human. Two problems arose from this: the human demonstrations could not be optimal and if they were, the agent will learn perfectly to drive in trackX, but it will drive badly in other tracks having overfitting. From the Unity ML v0.9 another imitation learning technique was introduced: GAIL (Generative Adversarial Imitation Learning). GAIL solved both of the previous issue because the agent now learns from the human and from the environment itself.

During training, the agent is allowed to act in the environment as usual, gathering observations on its own. At high level, GAIL trains the discriminator (a second neural network) to distinguish if a particular action-observation couple came from the agent itself or from the human's demonstrations. Then the agent is rewarded by how close its observation-actions are close to the human's demonstrations. The goal of the agent is to maximize this reward. The discriminator is updated with the agent's new observations and increases its discriminator ability. GAIL proceeds in an iteratively way: the discriminator gets tougher and the agent gets better in tricking the discriminator by mimicking the demonstrations.

GAIL will give a part of the reward to the agent. The other part comes from the reward function R explained in chapter 3.2.4. In this way we can combine GAIL with Reinforcement Learning. The GAIL reward will have a different (minor) weight from the RL reward[11].

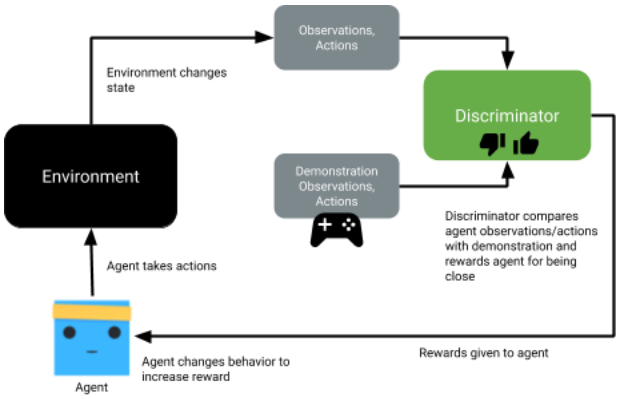


Figure 26: GAIL discriminator scheme

The process is divided like that: first you have to play several times through the game, in my case I recorded myself racing through the first random generated track for 7 laps. In this moment we are saving human's observations and actions. In order to do that I use a demonstration recorder and I save a .demo file.

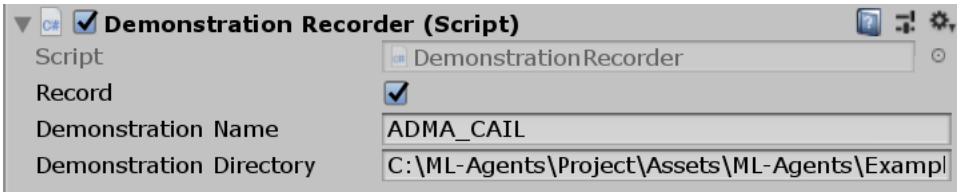


Figure 27: Demonstration Recorder

Later I edit the GAIL.yaml file to create a section for my CAIL neural network. Here I link the .demo file.

```
    CAIL:
        trainer: ppo
        batch_size: 1024
        beta: 5.0e-3
        buffer_size: 10240
        epsilon: 0.2
        hidden_units: 128
        lambd: 0.95
        learning_rate: 3.0e-4
        max_steps: 15.0e5
        memory_size: 256
        normalize: false
        num_epoch: 3
        num_layers: 2
        time_horizon: 64
        sequence_length: 64
        summary_freq: 10000
        use_recurrent: false
        reward_signals:
            extrinsic:
                strength: 1
                gamma: 0.99
            gail:
                strength: 0.2
                gamma: 0.99
                encoding_size: 128
                demo_path: Project/Assets/ML-
    Agents/Examples/MyDemo/demo1_adma_cail/admacail1.demo
```

As you can see the yaml file is exactly like the one in chapter 3.2.6 with the difference that in the reward signals there is a GAIL section. The strength is 0.2, gail reward will be multiplied by this value. In this way GAIL rewards will have a minor impact on the reward overall so that the trained agent will focus on receiving extrinsic rewards instead of exactly copying the demonstrations. The encoding_size is the size of the hidden layer used by the discriminator (that is a NN itself).

In the end I launch the training process. The tensorflow results are shown below

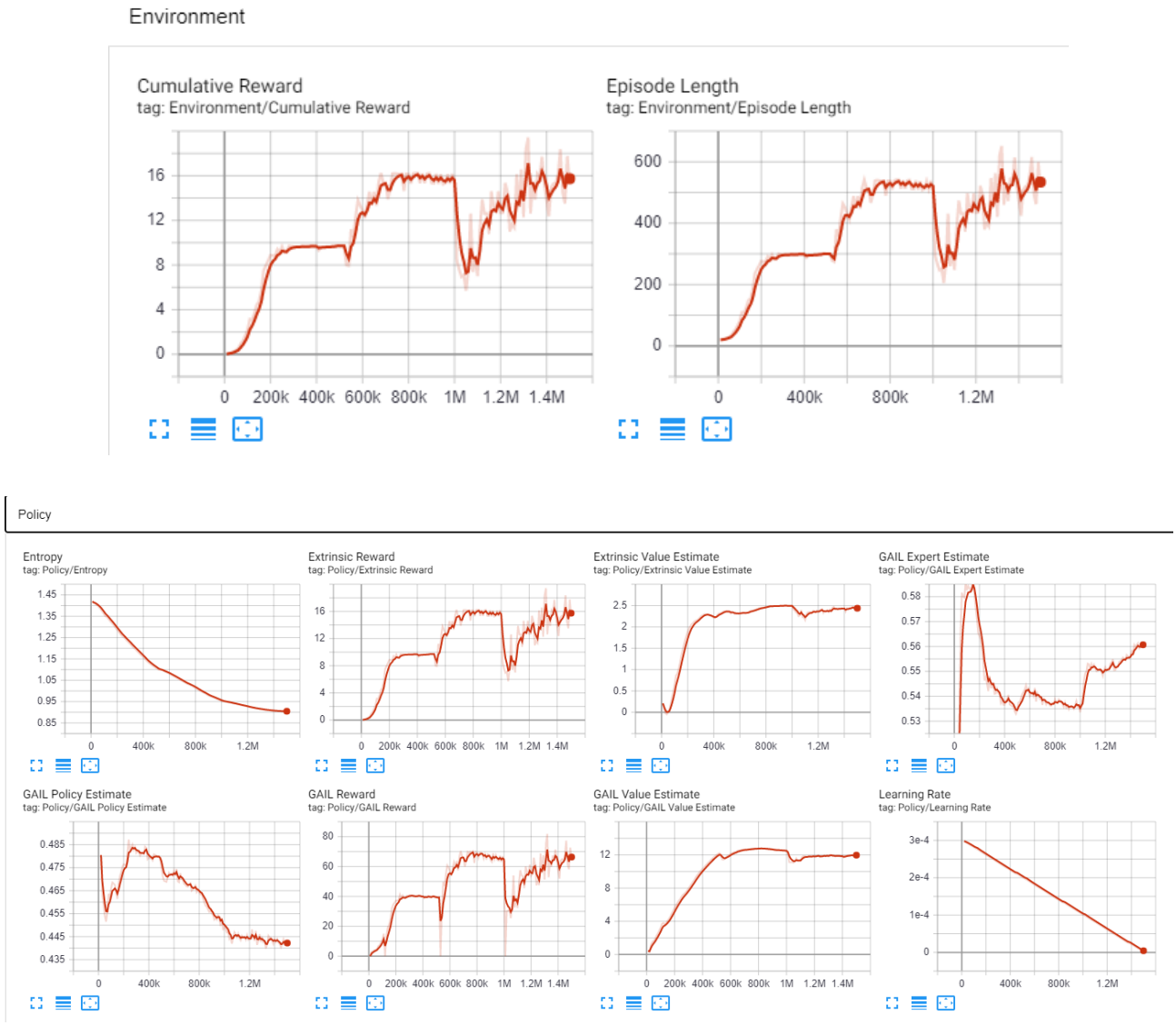


Figure 28: IL training graphs

These graphs are similar to the Reinforcement Learning one. There are some additions: GAIL expert estimate, that indicates the discriminator's estimate for states and actions drawn from my demonstrations; Gail policy estimate that show us the discriminator estimate for state and actions generated by the policy; GAIL reward, here we have the mean cumulative reward generated per episode by the discriminator (and it correctly increases); GAIL value estimate indicates the agent's value estimate for the GAIL reward (and it correctly increases); learning rate determines how large a step the training algorithm takes as it searches for the optimal policy, it should decrease over time and it decreases indeed.

3.3.3 Curriculum Learning implemented

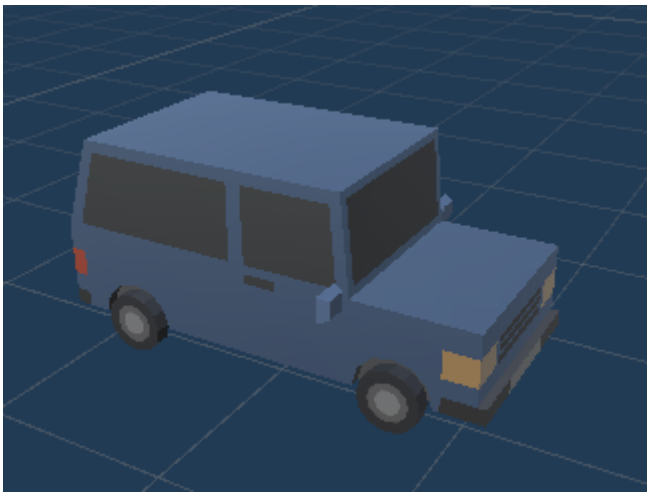


Figure 29: CL car

CACHCUL (Car Agent Checkpoint Curriculum Learning) was the third, the last and the most interesting neural network I train. It is based on the concept of progressive teaching: the agent will learn to solve simple task first and then complex task in a progressive way. In unity environment we use the term 'lesson'. The agent will start from lesson1 where it will learn how to perform a task. When certain conditions have been reached the agent will pass to lesson2 where it will learn how to solve a different (and usually more complex) task.

To define a curriculum the first step is to decide which parameters will change during the process. In my case the 'config-number' is changing with the crescent value of 0, 1 and 2. The config number will change the raycast and the collision matrix of the agent:

Config Number = 0 -> the agent will have raycasts that perceive only walls; the agent will ignore the collision with cars and obstacles

Config Number = 1 -> The agent will have raycasts that perceive walls and obstacles; the agent will ignore the collision with cars

Config Number = 2 -> The agent will have a raycasts that perceive walls, obstacles and cars; the agent will not ignore collisions.

```
private void FixedUpdate()
{
    //Car and car dont collide; Car and obstacle dont collide
    if (config_number == 0)
    {
        Physics.IgnoreLayerCollision(11, 13, true);
        Physics.IgnoreLayerCollision(11, 11, true);
    }
    //Car and car dont collide; Car and obstacle collide
    else if (config_number == 1)
    {
        Physics.IgnoreLayerCollision(11, 13, false);
        Physics.IgnoreLayerCollision(11, 11, true);
    }

    //Car and car collide; Car and obstacle collide
    else
    {
        Physics.IgnoreLayerCollision(11, 13, false);
        Physics.IgnoreLayerCollision(11, 11, false);
    }
}
...
public override void CollectObservations(VectorSensor sensor)
{
    if (config_number == 0)
    {
        //the same as the on in chapter 3.2.6 but with a raycastLayer that is only
        //wall
    }
    else if (config_number == 1){
        //the same as the on in chapter 3.2.6 but with a raycastLayer that is only
        //wall and obstacle
    }
    else{
        //the same as the on in chapter 3.2.6 but with a raycastLayer that is wall
        //obstacle and car
    }
...
```

We will define config_number as a reset parameter for the Academy.

```
...
public override void Initialize()
{
    m_ResetParams = Academy.Instance.EnvironmentParameters;
}
...
public override void OnEpisodeBegin()
{
    config_number = m_ResetParams.GetWithDefault("config_num", 2);
}
...
```

In the end we will create a section for CACHCUL in the trainer_config.yaml (same as CARL but with a different behavior name) and we will create a new yaml file in the folder config\curricula that will be our actual curriculum

```
    CACHCUL:
        measure: progress
        thresholds: [0.33, 0.66]
        min_lesson_length: 100
        signal_smoothing: true
        parameters:
            config_num: [0.0, 1.0, 2.0]
```

Here we define the threshold that the agent has to passed in order to change from a lesson to another. I use the measure progress, that is the ration of steps/maxsteps. This means that at 33% and at 66% of the training we will have a have a lesson change.

The min_lesson_length is the minimum number of episodes that should be completed before the lesson can change.

The signal_smoothing is a bool that if set to true weight the current progress measure by previous values.

In the parameters section we put the variable that we want to change during the training process when the threshold is reached. In my case the variable is config_num, that change from 0 to 1 at 33% of the training and from 1 to 2 at 66% of the training.

The results of the training are shown below

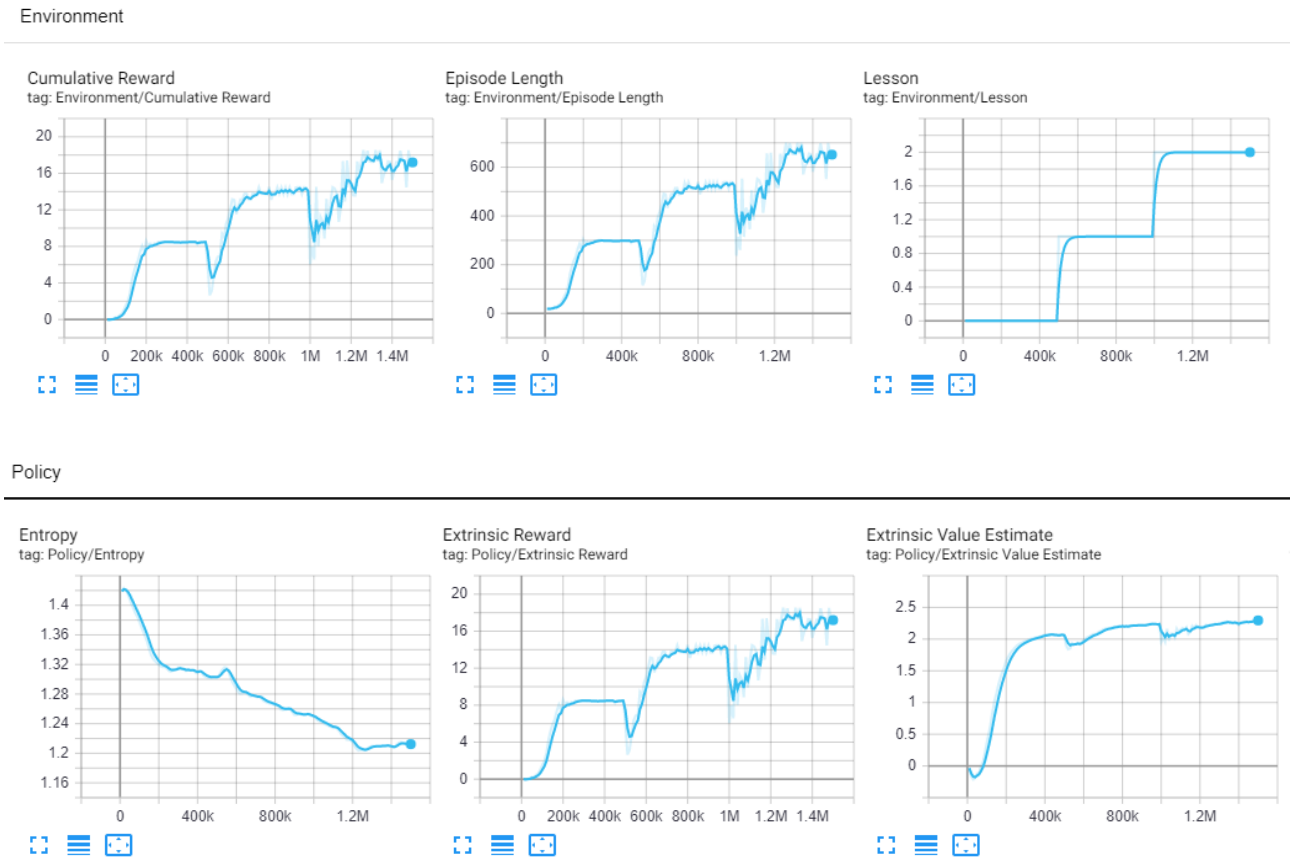


Figure 30: CL training graphs

The graphs follow the same behavior as the one of CARL. The only difference is the Lesson graph. This graph shows us that every 500k there is a change of lesson.

4 Tests

To test evaluate the performance of my cars I used three different metrics. I found this metrics in the book “Analysis Techniques for Racecar Data Acquisition”. In this way I got an objective and quantitative way to compare the three different neural networks. I test my metrics in a multi agent environment.

I gather the data of 5CL, 5RL, 5IL and 1 real player running for 7 laps on 3 different scenarios on 2 kind of tracks: SEEN and UNSEEN.

UNSEEN is a track that where the agents and the humans never performed a training. The agents were trained on other tracks (with the same process explained in chapter 3.2.6), the same for humans, I explained them the control of the game, I let them train on a few tracks and then I pose them in front of a brand new track.

SEEN is the same track, this time agents and humans had a pre training. For agents I train them like in the process explained in chapter 3.2.6 (one of the three tracks where they performed the training is the SEEN). For humans I let them take confidence with the track and then I gather data on their personal best performance.

In this way I got data of 5 CLs, 5 RLs, 5 ILs and 5players over 7 laps, over 3 scenarios for the SEEN track. The same for the UNSEEN track.

Then I made an average of each agent (and human) measure over 7 laps, an average of this over 5 different agents, an average of this over 3 different scenarios for the two kind of tracks.

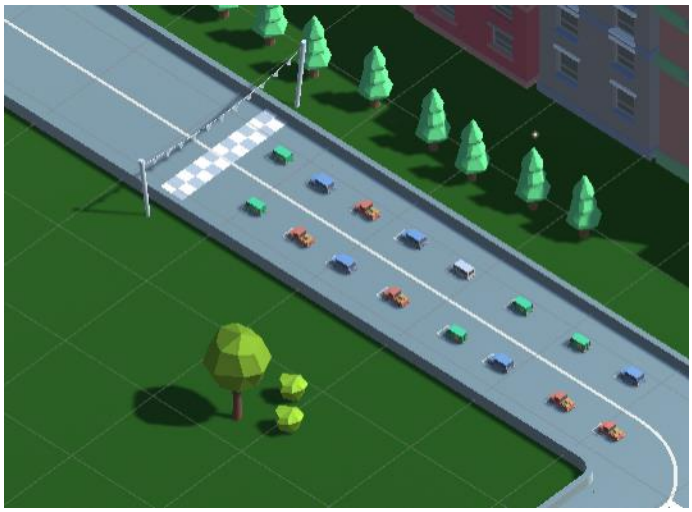


Figure 31: Scenario 1

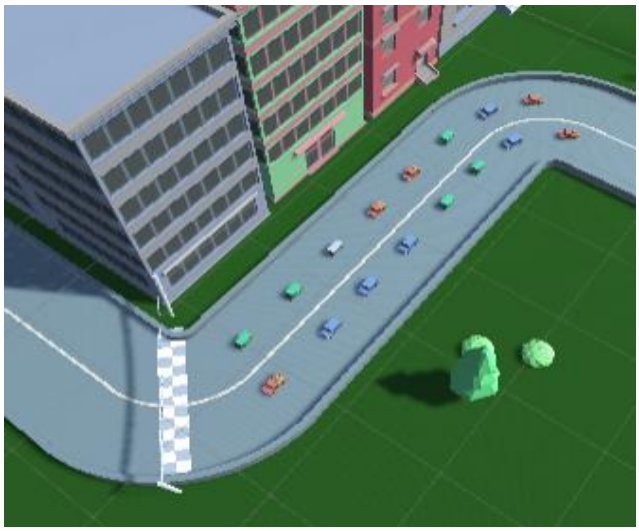


Figure 32: Scenario 2

Figure 33: Scenario 3

4.1 Metric 1: Lap time

When the car passed for the first time the Start Line a timer is started. When the car passed for the second time the Start Line the timer is stopped, and the value is saved. The value saved is the lap time of the first lap. We make the car run for 7 different laps and we saved 7 different lap time. If at the lap n the lap time of car x is less than the lap time of car y means that the car x is performing better than the car y in the lap n. To have metric over the full race we do the average of the 7 different value. Minor the value better is the performance.

4.2 Metric 2: percentage of lap at full throttle, medium throttle and low throttle

When the Start Line is passed three timers start:

-0.9-1.0 timer, this will be incremented every frame if the 0.9 < throttle value <=1

-0.5-0.9 timer this will be incremented every frame if the 0.5 < throttle value <=0.9

-0.0-0.5 timer this will be incremented every frame if the 0.1 < throttle value <=0.5

We plot these values on a 3D graph. On the x axis we have the number of the lap,

on the y axis we have the three different timers, on the z axis we have the amount of second of a single throttle value over a single lap. If a car has a 0.9-1 timer greater than the other two timer means that the car will be faster and better in term of performance. To have metric over the full race we do the average over the 7 different laps. In this way we will have the average of time a car passed at full throttle, the average of time a car passed at average throttle and the average of time a car passed at low throttle. Greater the full throttle value better is the performance.

4.3 Metric 3: number of crashes

In the beginning of the first lap we set the value crashNum at zero. Any time the car hits a wall or an obstacle a crash is registered and the value crashNum is incremented by one. At the end of the first lap we save the value of crashNum and we reset this variable to zero. The saved value is the number of crashes over the first lap. Let us say hypothetically that at lap n the car x registered K crashes and the car y registered L crashes. If K < L the car x is performing better and safer than the car y at lap n. To have a metric over the full race we do the average of the 7 different values. Minor the value better is the performance.

TRACK 1

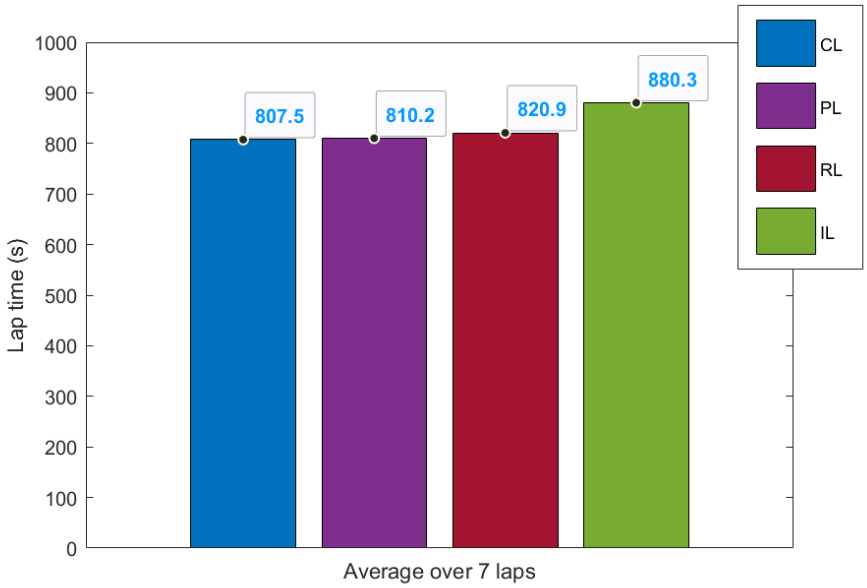
Figure 34: Track1

TRACK1 SEEN

Metric1 Lap time

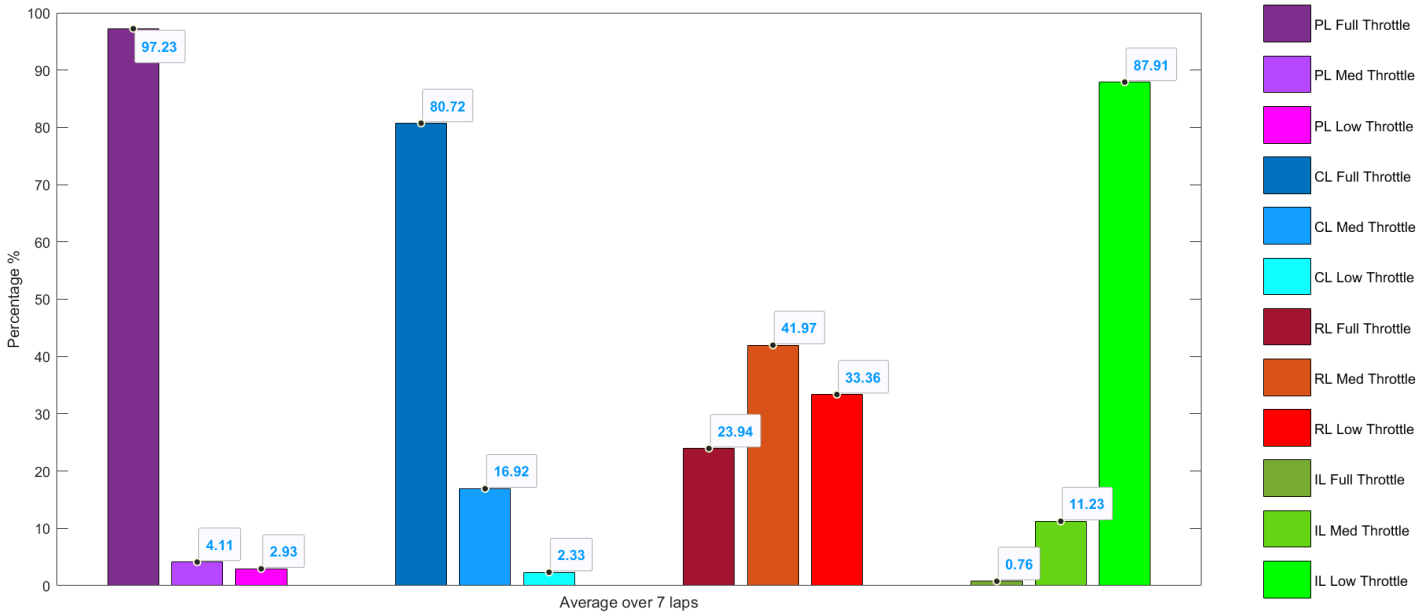
If SD is Standard Deviation and CV is Coefficient of Variation we have

For all data gathered we have $1.10 < SD < 1.48$; $1.4*10^{-3} < CV < 1.8*10^{-3}$



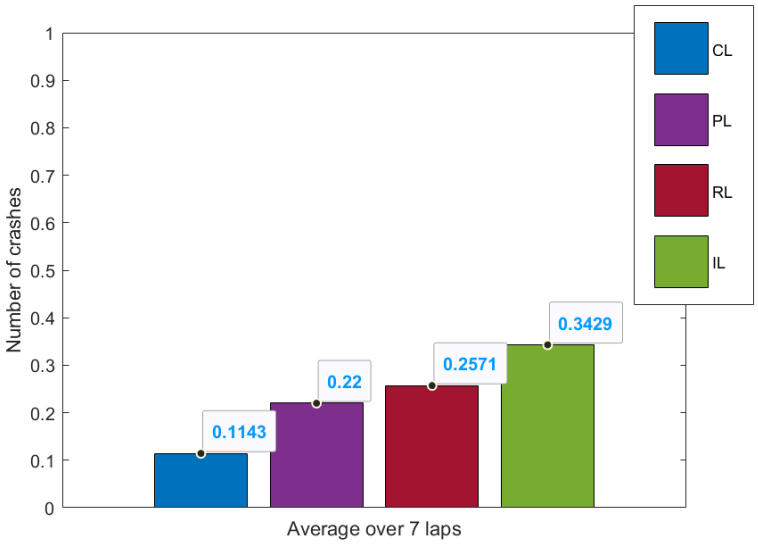
Metric2 Percentage

For all data gathered we have $0.08 < SD < 1.57$; $7.3*10^{-3} < CV < 515.4*10^{-3}$



Metric3 Number of crashes

For all data gathered we have $0.11 < SD < 0.5$; $0.8*10^{-3} < CV < 2.27*10^{-3}$

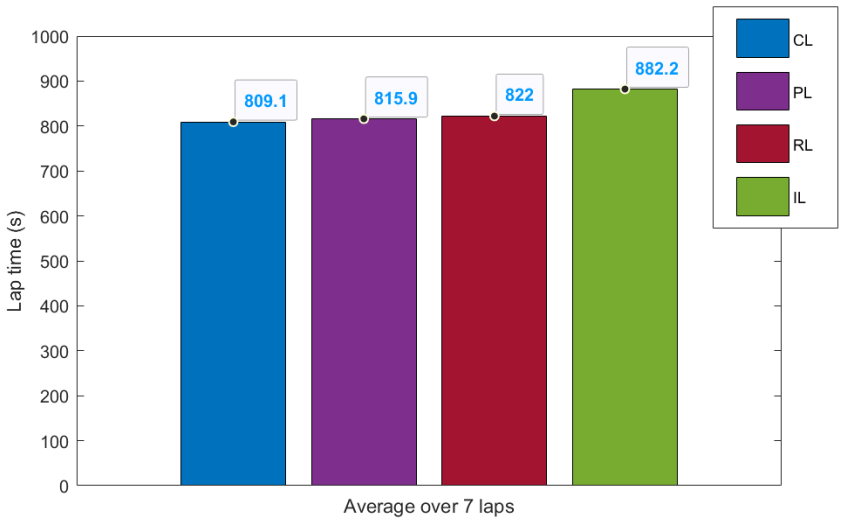


We can see how all SD are < 1.57 this means that data are clustered around the mean. CV are < 0.23% for metric 1 and 3, this means that there is no variability around this data. Meanwhile CV in metric 2 is much higher, CV are < 51.5%, this means that we have more variability in this dataset.

TRACK1 UNSEEN

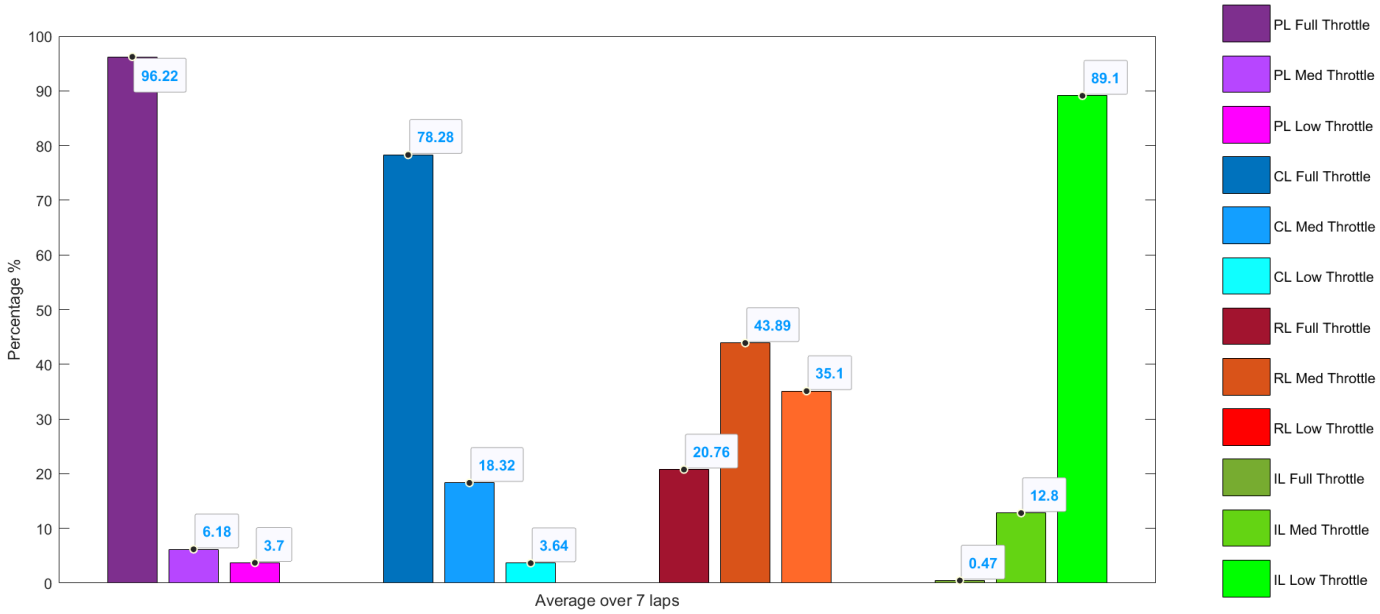
Metric1 Lap time

For all data gathered we have $0.54 < SD < 1.11$; $1.1*10^{-3} < CV < 1.5*10^{-3}$



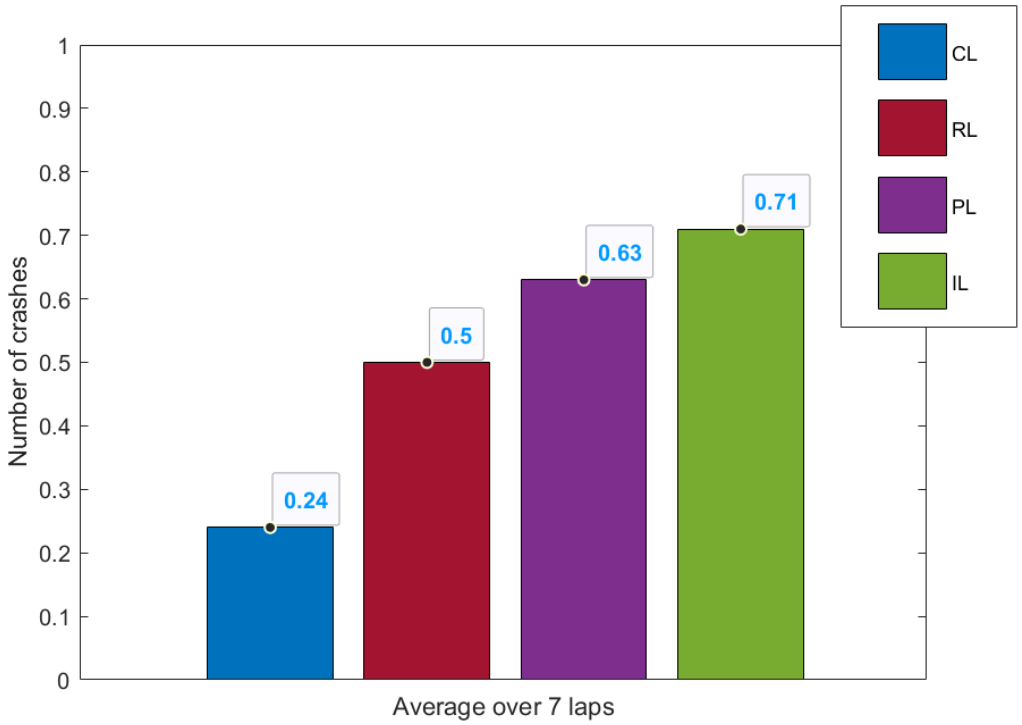
Metric2 Percentage

For all data gathered we have $0.63 < SD < 1.23$; $8.8*10^{-3} < CV < 482.8*10^{-3}$



Metric3 Number of crashes

For all data gathered we have $0.26 < SD < 0.7$; $0.48*10^{-3} < CV < 1.11*10^{-3}$



We can see how all SD are < 1.23 this means that data are clustered around the mean. CV are < 0.15% for metric 1 and 3, this means that there is no variability around this data. Meanwhile CV in metric 2 is much higher, CV are < 48.2%, this means that we have more variability in this dataset.

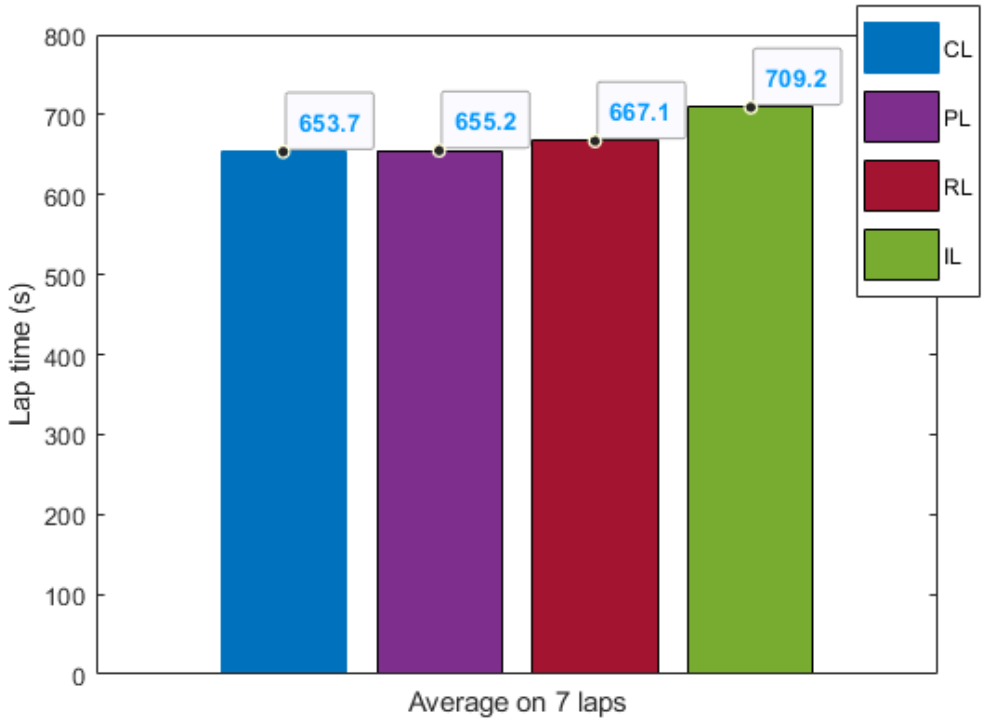
TRACK 2

Figure 35: Track2

TRACK2 SEEN

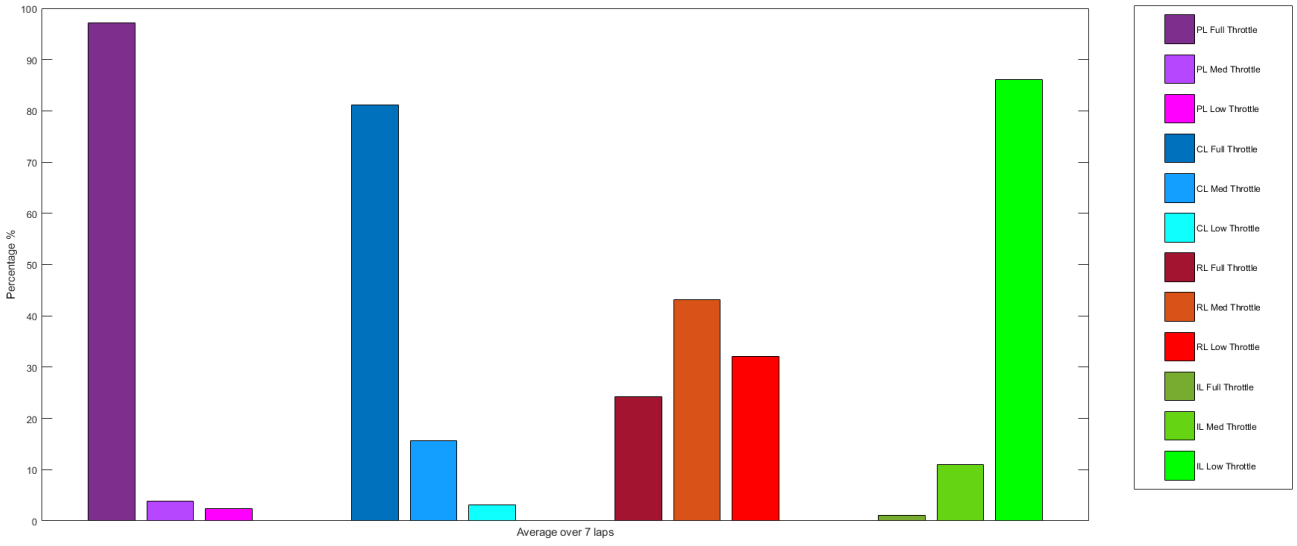
Metric1 Lap time

For all data gathered we have $1.21 < SD < 1.38$; $1.35*10^{-3} < CV < 1.7*10^{-3}$



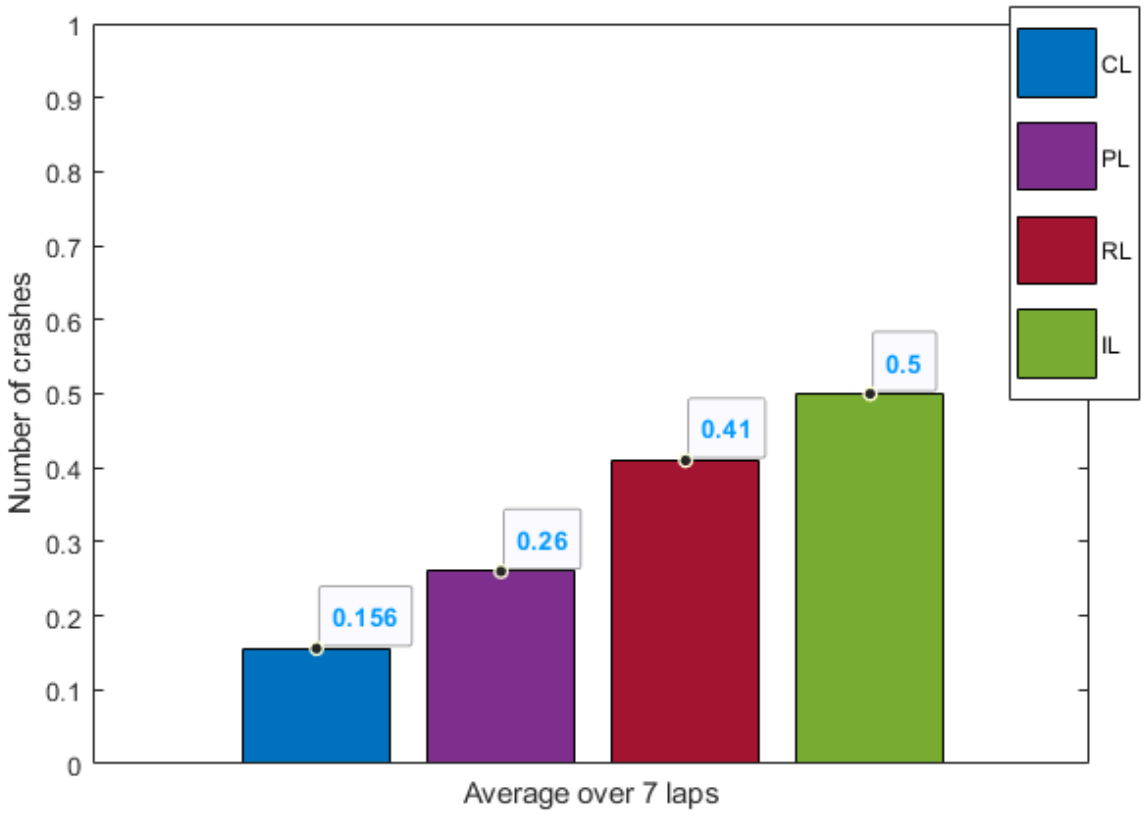
Metric2 Percentage

For all data gathered we have $0.02 < SD < 1.59$; $7.9*10^{-3} < CV < 502.4*10^{-3}$



Metric3 Number of crashes

For all data gathered we have $0.23 < SD < 0.61$; $0.7*10^{-3} < CV < 2.02*10^{-3}$

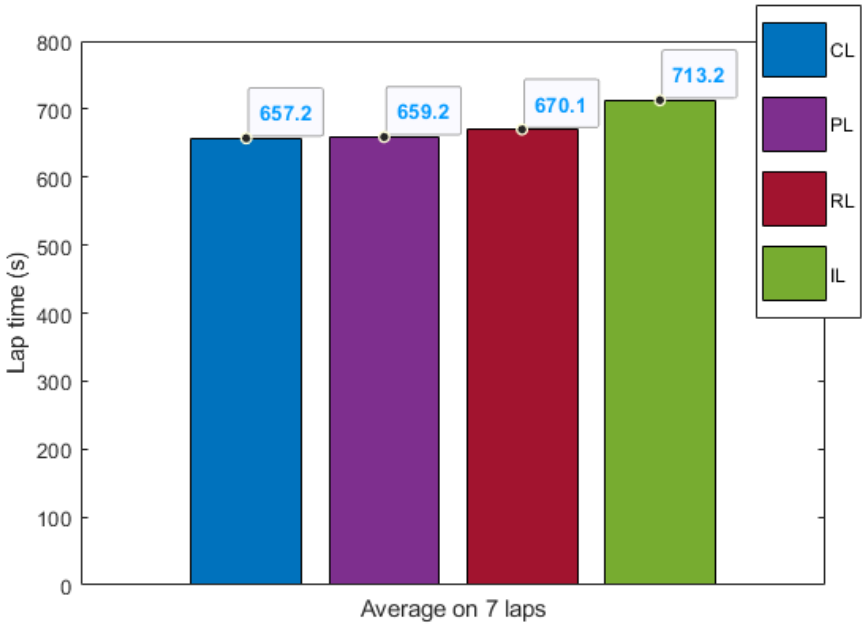


We can see how all SD are < 1.21 this means that data are clustered around the mean. CV are < 0.20% for metric 1 and 3, this means that there is no variability around this data. Meanwhile CV in metric 2 is much higher, CV are < 50.2%, this means that we have more variability in this dataset.

TRACK2 UNSEEN

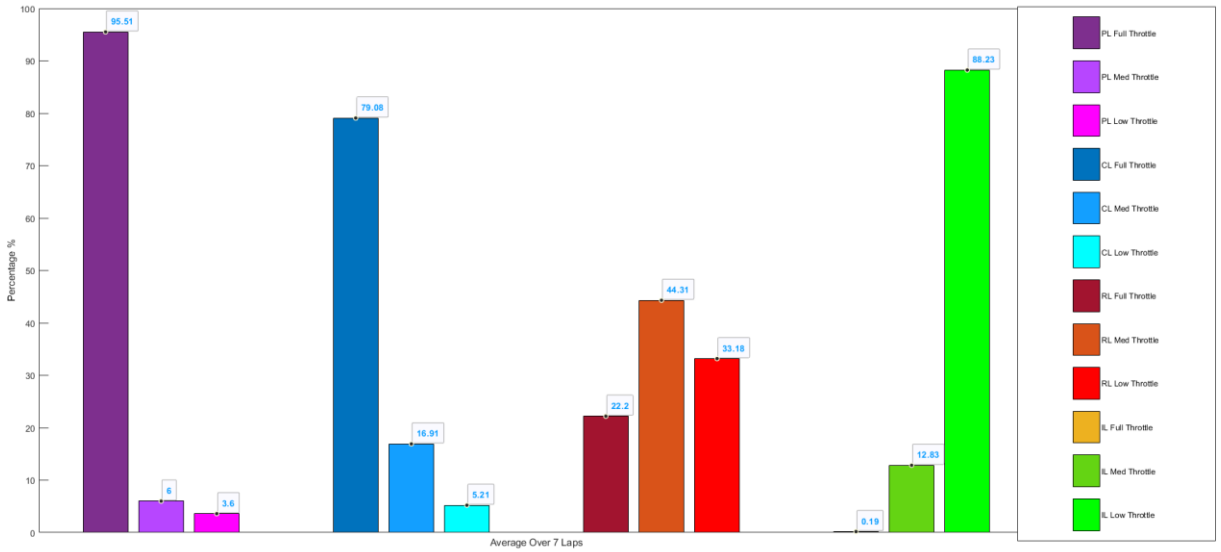
Metric1 Lap time

For all data gathered we have $1.19 < SD < 1.32$; $1.31*10^{-3} < CV < 1.59*10^{-3}$



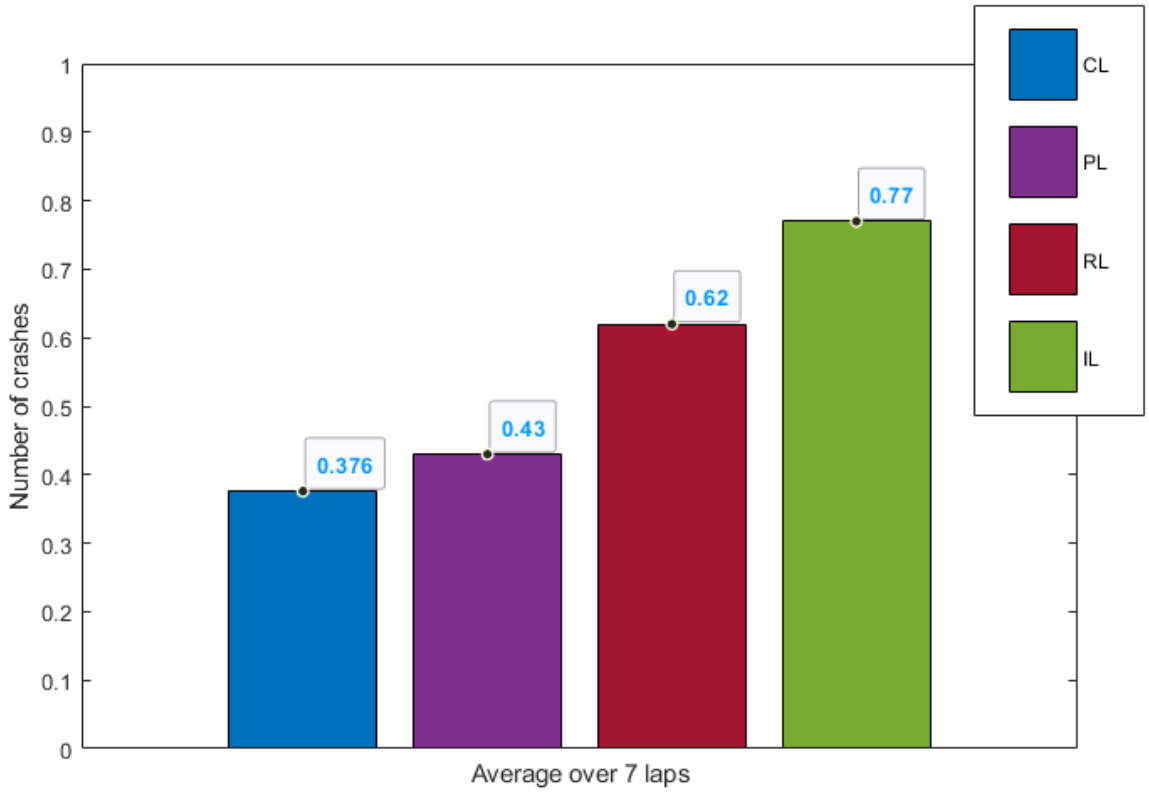
Metric2 Percentage

For all data gathered we have $0.021 < SD < 1.67$; $7.2*10^{-3} < CV < 501.2*10^{-3}$



Metric3 Number of crashes

For all data gathered we have $0.13 < SD < 0.41$; $0.62*10^{-3} < CV < 1.92*10^{-3}$



We can see how all SD are < 1.67 this means that data are clustered around the mean. CV are < 0.19% for metric 1 and 3, this means that there is no variability around this data. Meanwhile CV in metric 2 is much higher, CV are < 50.1%, this means that we have more variability in this dataset.

TRACK 3

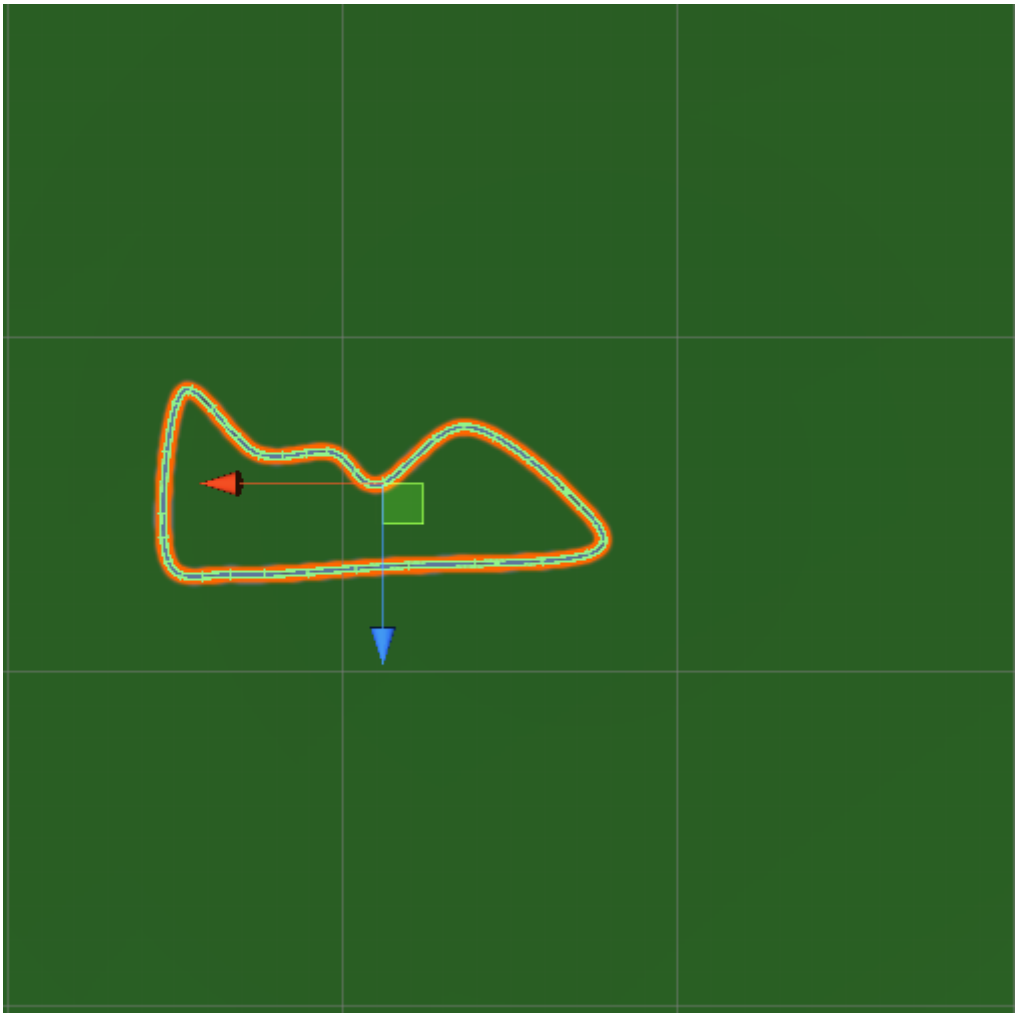
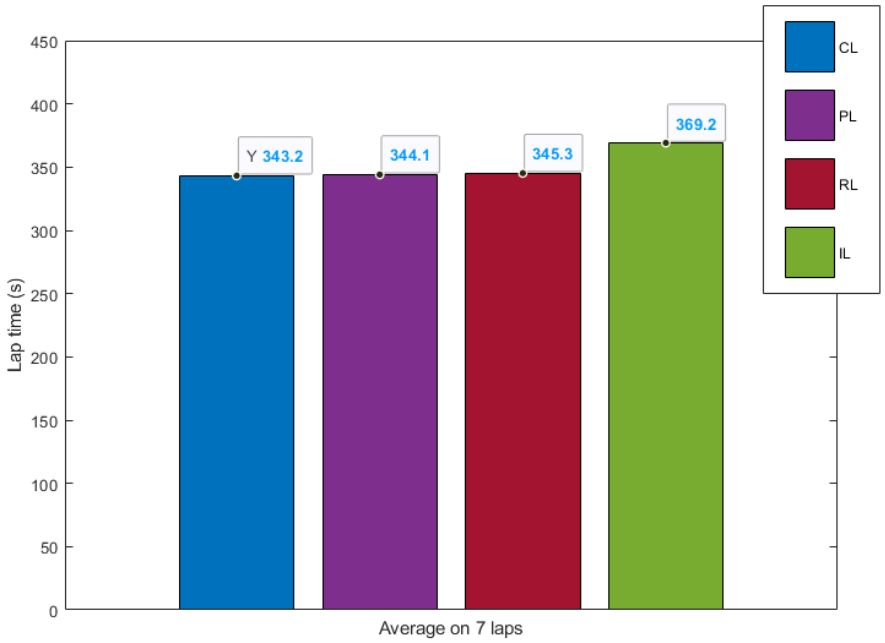


Figure 36: Track3

TRACK3 SEEN

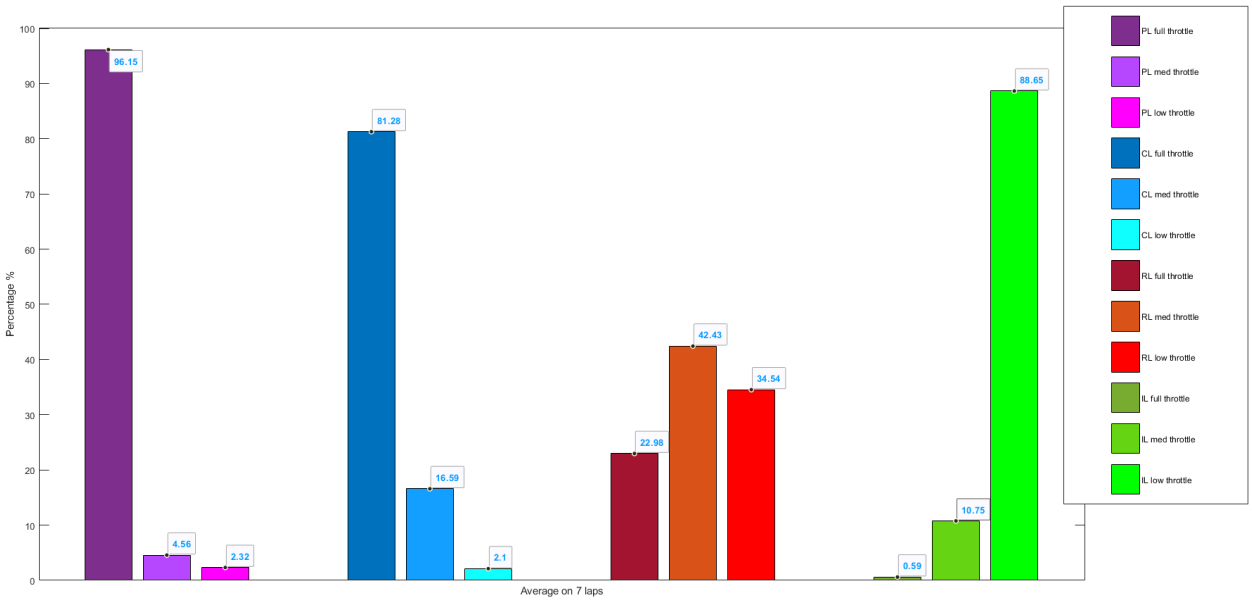
Metric1 Lap time

For all data gathered we have $1.19 < SD < 1.27$; $1.39*10^{-3} < CV < 1.68*10^{-3}$



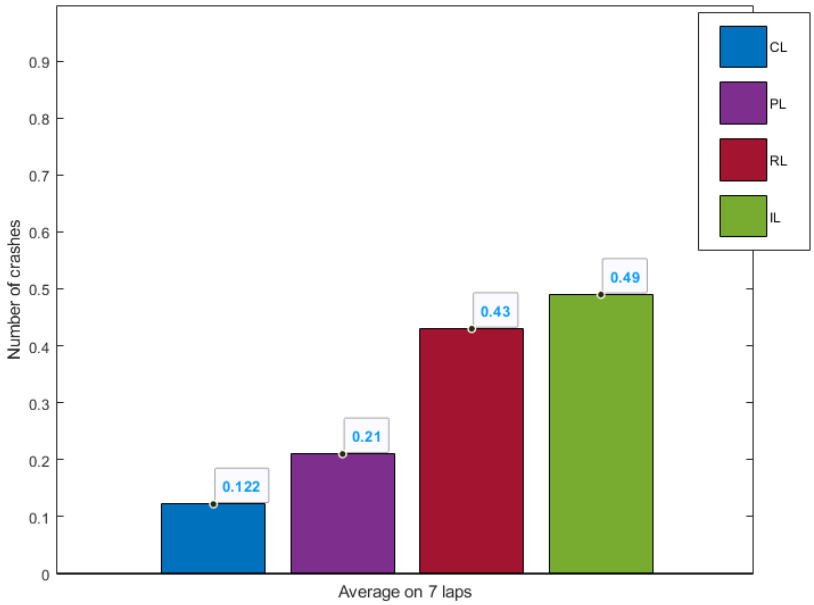
Metric2 Percentage

For all data gathered we have $0.03 < SD < 1.28$; $6.91*10^{-3} < CV < 501.3*10^{-3}$



Metric3 Number of crashes

For all data gathered we have $0.19 < SD < 0.62$; $0.68*10^{-3} < CV < 2.22*10^{-3}$

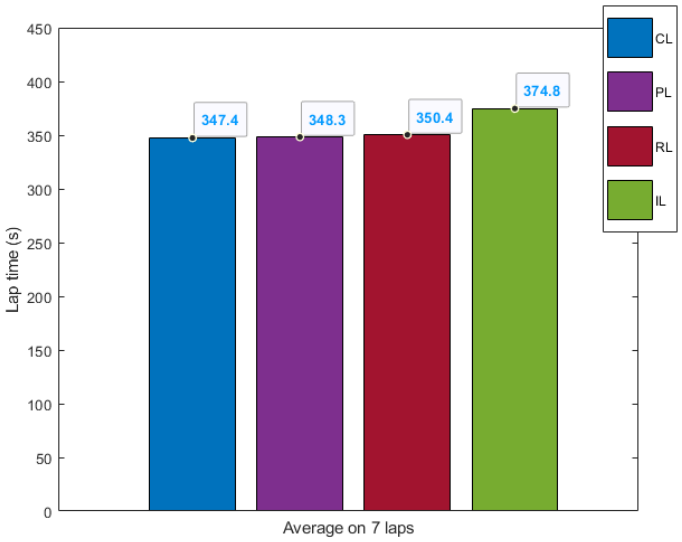


We can see how all SD are < 1.27 this means that data are clustered around the mean. CV are < 0.22% for metric 1 and 3, this means that there is no variability around this data. Meanwhile CV in metric 2 is much higher, CV are < 50.1%, this means that we have more variability in this dataset.

TRACK3 UNSEEN

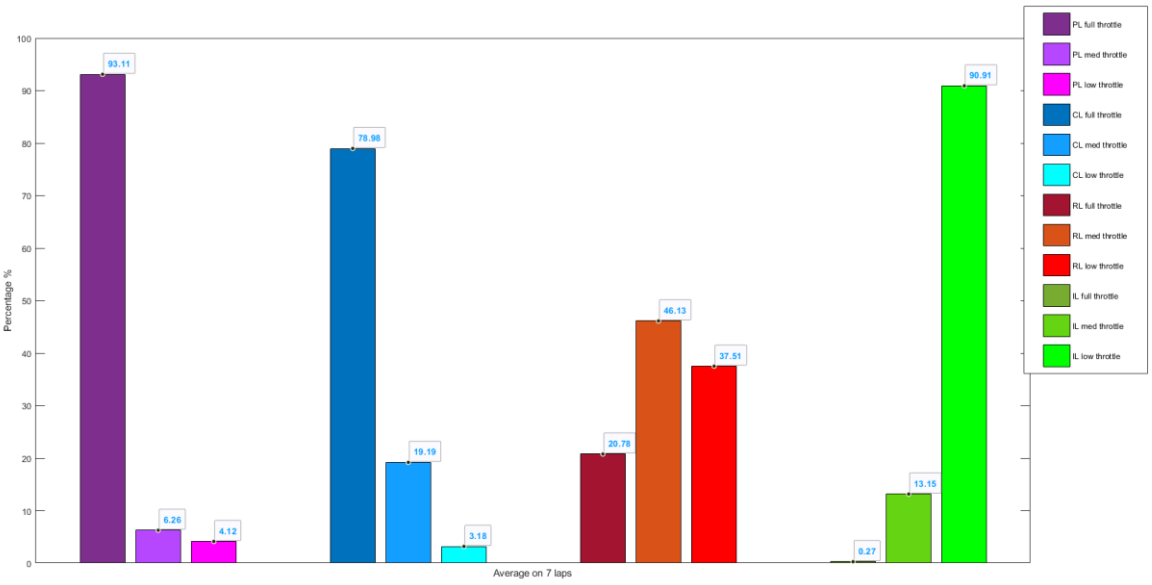
Metric1 Lap time

For all data gathered we have $1.21 < SD < 1.39$; $1.36*10^{-3} < CV < 1.72*10^{-3}$



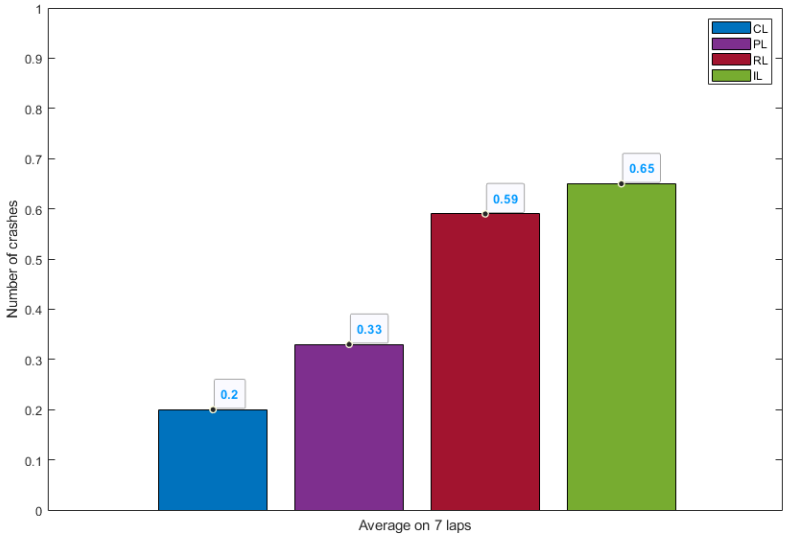
Metric2 Percentage

For all data gathered we have $0.01 < SD < 1.71$; $7.01*10^{-3} < CV < 503.2*10^{-3}$



Metric3 Number of crashes

For all data gathered we have $0.22 < SD < 0.5$; $0.55*10^{-3} < CV < 1.87*10^{-3}$



We can see how all SD are < 1.39 this means that data are clustered around the mean. CV are < 0.18% for metric 1 and 3, this means that there is no variability around this data. Meanwhile CV in metric 2 is much higher, CV are < 50.3%, this means that we have more variability in this dataset.

5 Conclusions

In this thesis, it has been studied in deep the incredible and new world of Machine Learning focusing on videogames. We have seen a focus on three techniques: Reinforcement, Imitation and Curriculum. I successfully applied these techniques onto self driving cars in a Virtual Environment of a racing game. This Virtual Environment is a great benchmark that could be used in future to test other machine learning cars.

In the end the ML cars have been tested with 3 different objective metrics. The results have shown how Curriculum is the best approach for this kind of task.

CL has the smallest Lap Time on average. The time passed at full Throttle for CL is the highest. So it has better performance. We see how CL has less crashes and so it is the safest driving car.

Real Players will pass more time at full throttle, but they have a bigger lap time. That is because they have a bigger number of crashes. So we can say that players try to complete the track going at max-speed, but they will crash more than CL having a worst performance.

To conclude I can say that Machine Learning could be a powerful tool for game developer but with some limitations: The first limitation of ML is computational power. Once the Reward function and the NN are well implemented, a train of 90 hours on a super-cluster-computer could be launched. This training will give us an AI with better performance than one trained on a 500$ laptop. This creates a gap in development where big corporation will probably have the best performing agents. Indie game could suffer from this fact.

Moreover if there is no cap in the learning process, the agent will become smarter and smarter and hard (if not impossible) to defeat. In my project this was solved by

restarting the reward at the end of the lap. In this way the reward is not a function that tends to infinity.

If we do not use take the right precautions we end up having an over powerful AI that can not be beaten. That is frustrating and not challenging (or fun) for players. In my project we can see how on average players perform worst than CL, but the AI is beatable.

The future of ML in videogame is flourishing and promising. The machine will learn to solve more and more tasks. The problem is that sometimes we see AI like an enemy to beat. AlphaGo in 2016 won against Lee Sedol in the game of GO, after some years Lee retired with this statement "*Even if I become the number one, there is an entity that cannot be defeated".* Of course it cannot be beaten, the AI was trained for years with a super-computer studying the best players' technique in the world. How can you compete against such an entity? Imagine closing a real person in a room and train him all his life to play a specific game. Imagine that his trainer will be the best player in the world. It is very probable that this person will become the best player in the world, but it is also probable that he will have suicidal tenders. Let the computers be computers and humans be humans.

We should change our point of view on AI. It is not an entity to be beaten but a tool to be used. AI could analyze our gameplay to give us hints to improve are moves. AI could be a fair competitor to play against. It is not an undefeatable totem.

6 Bibliography

[1] Amira E. Youssef, Sohaila El Missiry, Islam Nabil El-gaafary, Jailan S. ElMosalami, Khaled M. Awad, Khaled Yasser. *Building your kingdom Imitation Learning for a Custom Gameplay Using Unity ML-agents*

10.1109/IEMCON.2019.8936134

[2] Kun Shao, Yuanheng Zhu, Dongbin Zhao. *StarCraft Micromanagement with Reinforcement Learning and Curriculum Transfer Learning*

10.1109/TETCI.2018.2823329

[3] ShuQin Li, YiZhong Qi, JianBo Bo, Yao Fu. *Design and Implementation of Surakarta Game System Based on Reinforcement Learning*

10.1109/CCDC.2019.8832340

[4] Stuart Russel, Peter Norvig. *Artificial Intelligence: A modern approach 4th edition*. Prentice Hall.

[5] Mel Vecerik, Todd Hester, Jonathan Scholz, Fumin Wang Olivier Pietquin, Bilal Piot, Nicolas Heess Thomas Rothörl, Thomas Lampe, Martin Riedmiller. *Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards*

https://arxiv.org/pdf/1707.08817.pdf

[6] Jeffrey Elman. *Learning and Development in Neural Networks: The Importance of Starting Small*

DOI: 10.1016/0010-0277(93)90058-4

[7] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, Jason Weston. *Curriculum Learning*

https://ronan.collobert.com/pub/matos/2009_curriculum_icml.pdf

[8] Zhipeng Ren, Daoyi Dong, Huaxiong Li, Chunlin Chen. *Self-Paced Prioritized Curriculum Learning with Coverage Penalty in Deep Reinforcement Learning*

DOI: 10.1109/TNNLS.2018.2790981

[9] Ian Hudson. *Racetrack Generator*

https://i-hudson.github.io/projects/2019-02-02-Race-track-Generator/

[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. *Proximal Policy Optimization Algorithms*

https://arxiv.org/abs/1707.06347

[11] OpenAI Spinning Up. *Proximal Policy Optimization*
https://spinningup.openai.com/en/latest/algorithms/ppo.html

[12] Jonathan Ho, Stefano Ermon. *Generative Adversarial Imitation Learning*

https://arxiv.org/abs/1606.03476