SELF DRIVING CAR IN UNITY

Techniques of artificial intelligence
INFO H410

Authors
Bertoldo Damiano, 000568813
Guija Valiente Tomás, 000568600

Contents

1 Introduction

Self-driving technology is currently being widely studied. In this work, we implemented, tested and compared two different methods for developing this kind of technology in a very simple simulation environment created in Unity: Deep Reinforcement Learning and Genetic Algorithms.
We will test both of these methods in different circuits, and run several experiments to examinate their performance, learning speed and adaptability under different circumstances.

1.1 Requirements

As the project is done in Unity and the scripts used, especially the definition of the agents, are written in C#, it is needed to have these two components intalled.

Also, in the Unity project itself is needed to have the *MLAgents* package installed. This package can be installed directly from the packet manager of the Unity IDE.
This package is needed in order to be accesible from the python library *ml-agents*, used for training the DeepRL agent, and for have all the dependency needed for creating the DeepRL agent.

2 Implementation

In our implementation we tried to compare two types of algorithms.

- Deep Reinforcement Learning
- Genetic Algorithms

The reason why we choose these two algorithms resides on the fact that DeepRL is the most common type used for this application and we wanted also to experiment and compare with other algorithms. For this reason we choose Generic algorithm, as it is simple to implement.

2.1 Deep reinforcement learning

The DeepRL algorithm used is Proximal Policy Optimization (PPO). The reason we chose the PPO algorithm is because we used the *ml-agents* python library, created by Unity, that has already this metodology implemented. The library helps to streamline the process of constructing an efficient environment or project, thus allowing us to concentrate more on these areas rather than on the algorithm itself.

2.2 Genetic algorithm

For Genetic Algorithms, we manually implemented the Neural Network, as well as the Genetic Manager, due to the simple nature of this algorithm.

2.3 Neural Network

Initially, we developed a neural network with the same structure as the one previously used in DeepRL for a fair comparison. However, after running the algorithm for many generations, we found out that this structure was too complex for such a simple method, and that it would not lead to good results. These kind of procedure relies heavily on probability, and a Neural Network with too many hidden layers and neurons will present an exponentially increasing number of configurations. Finding the correct one and letting it endure and improve along the time proved almost impossible.
We have therefore opted for a much simpler structure that demonstrated to give much better results. This one consisted of two hidden layers of ten neurons each. The inputs fed to the Neural Network, as well as its outputs did remain the same.

Input layer The input layer of the neural network are 12 features. The features correspond to:

- **Raytracing rays**: 7 rays equally distaciated that goes from the left side of the car to the right side of the car. An example can be seen in Figure 1
- **Current speed**: the current speed at which the car is going
- **Current acceleration**: the current acceleration of the car
- **Distance traveled since last checkpoint**: the total distance traveled since last passed checkpoint
- **Distance to next gate**: the distance to the next gate to cross
- **Distance to previous gate**: the distance to the previous gate already crossed

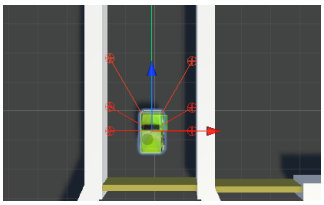


Figure 1: Raytracing lasers of the car

Output layer The output layer consists of 2 neurons. One is used for accelerating/breaking and the other for turning. For accelerating the values are:

- **Value** > 0: Accelerate
- **Value** $= 0$: Nothing
- **Value** < 0: Decelerate/Brake

For stearing the values are:

- **Value** > 0: Right turn

- **Value = 0**: Nothing
- **Value < 0**: Left turn

3 Environment

In order to test our self-driving car, we created a simple environment consisting in a single closed circuit. This circuit is bounded by walls, which the car will need to avoid at every moment. The walls are white, as in Figure 2.
Apart from the walls, there are also some checkpoints that indicate the direction of the course. The car will need to drive around the circuit always in the correct direction, following the order of the chekpoints. The chekpoints are yellow, as in Figure 2.

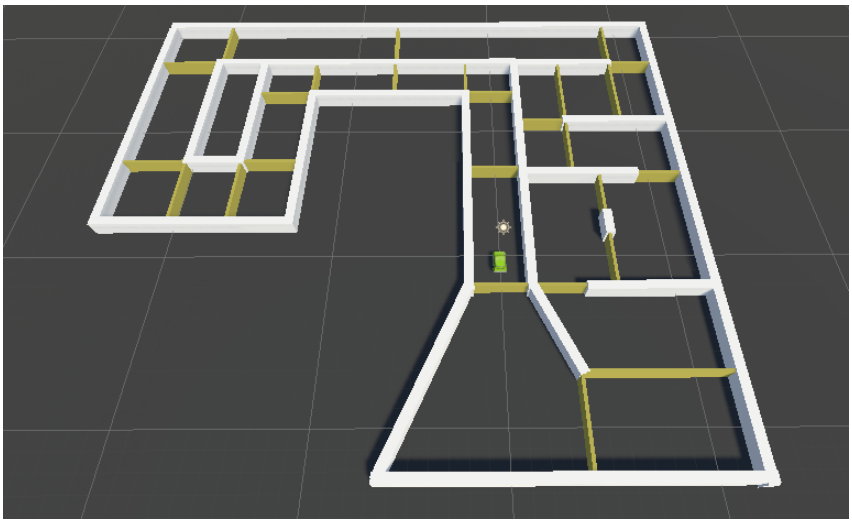


Figure 2: The training enviromnent created

This specific track was created in order to simulate various scanarios. For example long straight, a lot of turns or strange section in which our agent had to learn where to go. For this reason we had put different spawn point acrosss the track, some of them also facing the wrong direction.

3.1 Fitness and reward system

For both methods, we used the same fitness and reward system. On the one hand, we want to encourage the car travelling as much distance as possible around the circuit in the correct direction. On the other hand, we also want to reward speed. The overall score the car will obtain throughout one single run will be calculated according to the following criteria:

- To avoid the car going in circles from the beginning of the experiment and encouraging the car to face the good direction, we assigned the reward to the agent that is the distance traveled to the next gate. That could be positive if it gets closer or negative in the other case. Since the time at which Unity refreshes its status is fixed, then the faster the car goes the higher the reward.

- We will reward the distance travelled between gates based also on the time used to get there. In this case, since the distance between the checkpoints is fixed, the variation on the reward is dictated by the elapsed time. Hence, the faster it goes the higher the reward.
- Finally, we want to punish the car going backwards on the track. As the car drives through gates in the wrong direction, the punishment received will grow exponentially.

4 Training

There are two configuration for train the agents. One is used for DeepRL and the other for Genetic algorithm.

4.1 PPO

In order to train the DeepRL agent, is needed to load the scene, usually *RLMultiTrain*. If, for some reason, is needed the customization for fitness values, then is advised to open the prefab *RLEnviroment* as this component is the reference of every copy inside the scene.

After this, is needed to run the command of the python library:

```
$ mlagents-learn config.yaml --run-id=name
```

Inside config.yaml there are all the specification of the hyperparameters for the algorithm, included the neural network composition.

In order to start the training is needed to press the play button.

4.2 Genetic algorithm

In order to train the genetic agent, is needed to load the scene, usually *GAMultiTrain*. If, for some reason, is needed the customization for fitness values or neural network composition, then is advised to open the prefab *GAEnviroment* as this component is the reference of every copy inside the scene.

If is needed to modify values from the genetic manager, then in the scene *GAMultiTrain* there is the component *_manager* in which are present all the parameters.

In order to start the training is necesary to press the play button. Given that this operation does not automatically terminate, to terminate it is necessary to press the stop button.

5 Results

As expected, in the competition for the best self-driving car implementation we have a clear winner. DeepRL is by far the most suitable method for this task. The final result after five million steps (around 40 minutes of training) would be able to do several runs without crashing with fairly good speed. However, the Genetic Algorithms did actually prove to have some good properties as well. The car would quickly learn how to do the first part of the course, before getting stuck on the more complicated parts, as we can notice on Figure 3(a) it reaches a limit value after 50 generation. This achievement is quite impressive, giving

the fact that it happens in minutes of training. Is impressive also because if we compare at which time the DeepRL agent reached that specific reward, is almost at half of the training, as seen in Figure 3(b).

I Figure 3(b) we can notice different reward over time, as these different run were made with different spawns point and Neural Networks.

- **Correct direction spawn**: in this case the spawn points present were always facing the correct direction. In fact we can see that is the first one to reach almost 1000 on reward.
- **Wrong direction spawn**: in this case the spawn points present were the same as before, but with the addition of some of them facing the wrong direction. In this case we can notice that it struggles to learn first, but then it was able to recover.
- **4 Layer NN**: this case is the same the previous one with the difference that the neural network was istanciated with 4 layers insthead of 2. Also we increased the hyperparameter *learning rate* to be double the value of the previous settings. We can notice that this run was performing normally during the middle part, but it excelled at the end, surpassing the other two.

As for adaptability, we made ourselves the following question: did the car learn to drive, or did it learn to go through a specific cirtuit? The answer was found after building a new track, where the DeepRL car would indeed manage to drive through the first part, failing after a couple of turns. Probably, the original circuit was designed so the features given to the agent would let to overfit the proposed problem, failing to face completely new situations.

As for the Genetic algorithm, as stated before, it would manage to be able to drive only if the circuit build for training and test is quite simple, like a circle or oval shape.

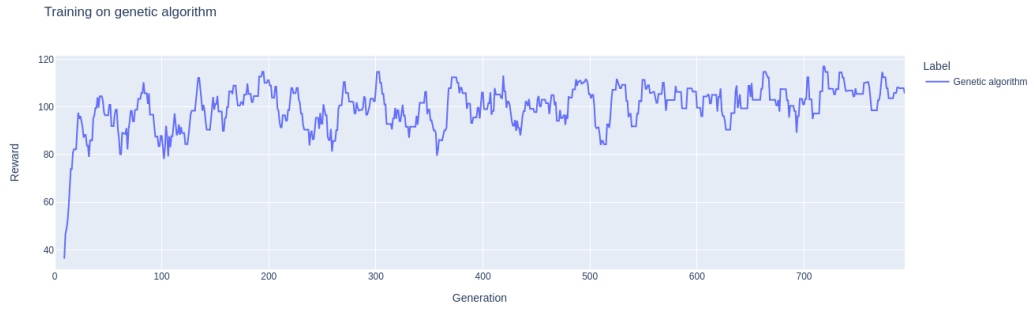
6 Conclusion

Overall, the results of this project were quite satisfactory. However, we learnt that, in order to build an AI capable of adapting to different situations and environments, it will need a more diverse training phase. In our case, the way the track was built, as well as the reward system, were too focused on obtaining and agent able to drive through our hand-built circuit.

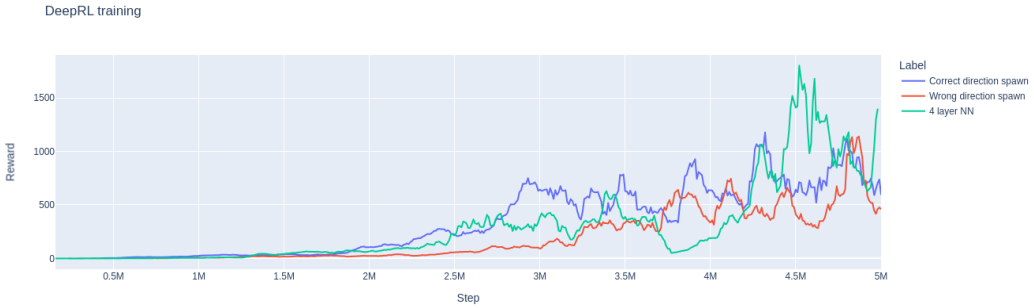
As we have tested our DeepRL agent on a different track with similar features, we saw that the agent was unable to drive in a good way. It would recognize some turns, but on the long run it wouldn't be able to complete even half of the track.

We supposed that this is because of the reward system was too biased towards the train environment track. Hence the agent mastered the training track thanks to the environment variables in its possesion.

To avoid this behaviour we could have trained the car purely on the distance traveled on the correct direction, without using gates, as this would have allowed the raycast sensor to have a bigger influence over the car's decisions, rather than considering where the next and previous gates lie. This would have probably lead to the car beign able to drive in different environments.



(a) Genetic algorithm training



(b) DeepRL training

Figure 3: Training results