Self-parking cars in Unity

Annika Richter

Cornelius Wolff

Kai Dönnebrink

1. Introduction

One of the most popular examples of practical applications in the field of artificial intelligence (AI) especially reinforcement learning (RL) is autonomous driving. However, since it is a complex problem with multiple factors to consider, e.g., speed, traffic signs, and other road users, and several challenges, e.g., understanding traffic signs and prevent accidents, it can be hardly be presented and solved in a simulation. Given the limited time and computational resources, we decided to model only a small aspect of the whole autonomous driving problem. We focused on creating a self-parking car in a limited simulation using a suitable 3D engine. This requires the agent to learn how to handle the car, i.e., accelerate, brake, and steer, and how to navigate, i.e., find a parking lot. Our idea was influenced by the YouTube video *AI Learns to Park – Deep Reinforcement Learning* from Samuel Arzt[1].

In this project we were simultaneously solved multiple problems. Firstly, we learned how to design an environment with Unity. Secondly, we wanted to investigate which effect different environment, e.g., having a fixed parking spot or different parking spots, have on the learning of the agent. Finally, we were interested in the capabilities of training a Unity environment with a Python training script instead of using the training possibilities offered by Unity.

To accomplish our aim of having a self-parking car agent we divided the task into three steps. At first, we designed an environment in Unity and tested it with manual controls. This ensured that the control of the car works, the rewards are calculated correctly, and the observations are collected. In the second step we used the *Unity Machine Learning Agents Toolkit* (ML-Agents) to train a policy for the car in the environment. By doing this, we were able to fine-tune the rewards and make sure that the environment is

[1] https://www.youtube.com/watch?v=VMp6pq6_QjI

solvable. Once we had a solvable working environment, we continued with our final step, in which we implemented the training algorithm by ourselves.

1 Background Information

1.1 Unity and ML-Agents

Unity is a game engine, that can be used to create 2D and 3D games, apps, and experiences.[2] Besides the 3D objects that can be easily created it also has built-in physics engines for 2D and 3D that ensures that the objects correctly respond to all kind of forces.[3] The 3D physics also offers a special object, i.e., the Wheel Collider, developed for vehicles with wheels such as cars.[4] For these reasons and our previous experience with Unity we decided to create the training environment for the self-parking car with Unity's 3D game engine. Furthermore, the Unity environment can be wrapped as an OpenAI Gym environment[5], where we already know how to use it.

ML-Agents is a software package developed by Unity Technologies. It enables game developers and researcher to use Unity games and simulations as an environment for training intelligent agents.[6] For game developers it provides PyTorch based implementations of a few state-of-the-art algorithms like Proximal Policy Optimization (PPO) and Soft Actor Critic (SAC). For researcher it offers a Python API to train agents using any preferred method.

Since we found a few similar projects that used PPO and achieved very good results we decided to use a PPO structure as well. In the following we used the PPO implementation for two things. First, we used it to test whether our created environment is solvable. Since the provided implementations are optimized to function with Unity this works better than using our own implementation to test solvability, because our own implementations may be buggy. Furthermore, the provided implementations are also better integrated such as the agent can use multiple environments to collect samples. This had a huge impact on the learning speed. Secondly, we used the PPO implementation to train the agent with different environments. Afterwards we used the Python API to train the agent with our own PPO implementation.

[2] https://docs.unity3d.com/2021.1/Documentation/Manual/index.html

[3] https://docs.unity3d.com/2021.1/Documentation/Manual/PhysicsSection.html

[4] https://docs.unity3d.com/2021.1/Documentation/Manual/class-WheelCollider.html

[5] https://github.com/Unity-Technologies/ml-agents/blob/main/gym-unity/README.md

[6] https://github.com/Unity-Technologies/ml-agents

1.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an Actor Critic Style Deep Reinforcement Learning Approach using a clipped loss. PPO is a highly successful policy gradient algorithm and was first introduced by John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov of Open AI in 2017.

Making use of a clipped loss, it prevents the policy from being updated to drastically from the original policy. This creates a trust region, that defines how far the loss can be altered from the initial one. The predefined minimum and maximum determine the smallest and largest update that is allowed. This technique allows the user to perform batch updates on its networks. PPO is updated using policy gradient and has an Actor and a Critic style network. The Actor network is our Policy network and updates the policy distribution. The Critic network is our Q-Network. Taking the Observation of the Actor Network as an input it judges the quality of the actions that the current policy creates.

Since PPO makes use of Policy Gradients it allows us to have a continuous action space. Implementing the movements of our agent, it would be possible to have a discrete action space as well. Nonetheless it allows us to have a more realistic control over the movement of the car using a continuous action space or an action space with exceedingly small discrete actions, why we decided to define continuous actions.

2 Environment Design

The environment we created consists of a car park on the left side with two free parking spaces, whereby the car park marked in red is the currently selected target location. Furthermore, there is a road on the right side, which also serves as the spawn area for the agent. These two areas are separated by several trees and a street lamp. The lamp, the trees and the outer boundaries of the environment have a tag called "Obstacle". This will be needed later for the clear classification of possible collisions with the agent.

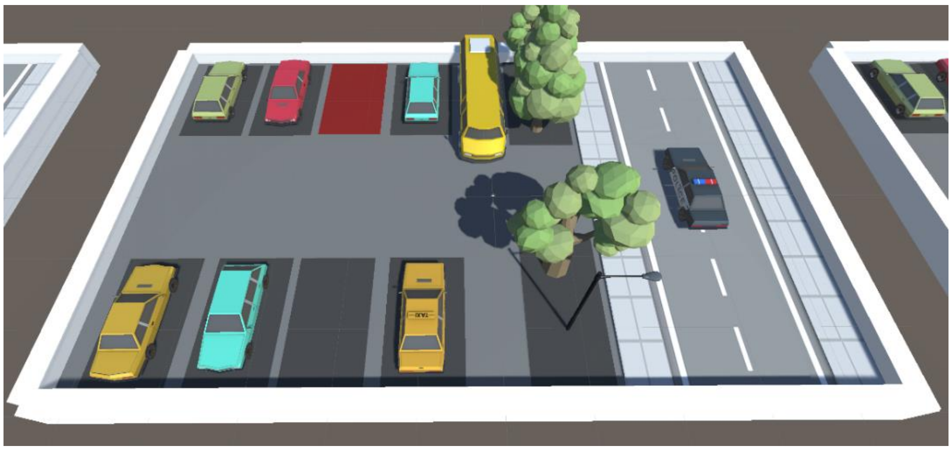

Figure 1: An example of an environment in the Unity editor.

In the further course of our work, the arrangement of the parking spaces and the corresponding cars has been changed several times resulting in that the agent learns to deal with different situations. The impact this had on the required training time and on the performance can be seen in chapter 4.1.2.

We obtained the various assets free of charge from Unity's Asset Store, the sources of which are linked below.

2.1 Agent Script

The environment script is responsible for giving the observations from the Unity environment to the agent and for executing the actions selected by the agent and returning a corresponding reward. The script also resets and spawns the agent. As this is directly coded in Unity, we used C# as the programming language.

Collecting observations is done with the help of so-called ray casts. In simple terms, these are distance sensors that can also provide some information about the object to which the ray cast is pointing. In our case, we use 8 ray casts that are evenly distributed. Furthermore, the environment is given the agents absolute orientation and its position relative to the target parking space. This is all the sufficient information the agent needs to find its target.

In the beginning, we decided to use a discrete action space. This means that the agent was not able to choose any percentage values as an action, but can only choose between a number of different predefined speed and turning increments. However, we decided to develop a very similar script for continuous action space, as this made it much easier for us to develop a corresponding model in TensorFlow. However, both scripts have in common that the maximum steering angle was set to 30 degrees, and the break force was set to 18000 and motor force to 9000. Additionally, the car is able to drive forwards much faster than backwards.

The latter factor is particularly important for the agent in the context of the returned rewards, as the car receives a negative reward for each timestep required. This leads to the car learning to drive forward when trying to park as quickly as possible. However, we also had to fix the sparse reward problem that would arise if the agent would only get a positive reward when reaching the destination. This was achieved using a function that gives a positive reward when the agent approaches the goal and conversely gives an equal negative reward when it moves away. This paid off heavily in training, as the agent learned relatively quickly to approach the target until at some point it happened to park correctly. At the beginning of each episode, the car was spawned within a predefined area on the road with a random orientation.

3 Training

3.1 Setup

While using Unity's integrated MLAgents add on, we placed a total of 9 instances of the environment created in chapter 5 in our level. This allowed the agent to learn much faster, since it can gain experience in nine different environments in parallel and thus optimize itself. However, the nine instances had some variety in our second training run. We did this to prevent overfitting as much as possible and to check what influence a more diverse environment had on the training time and performance (for results see chapter 4.1.2). For example, the free parking spots are in different positions and the position of the bus and the arrangement of the trees were also changed.

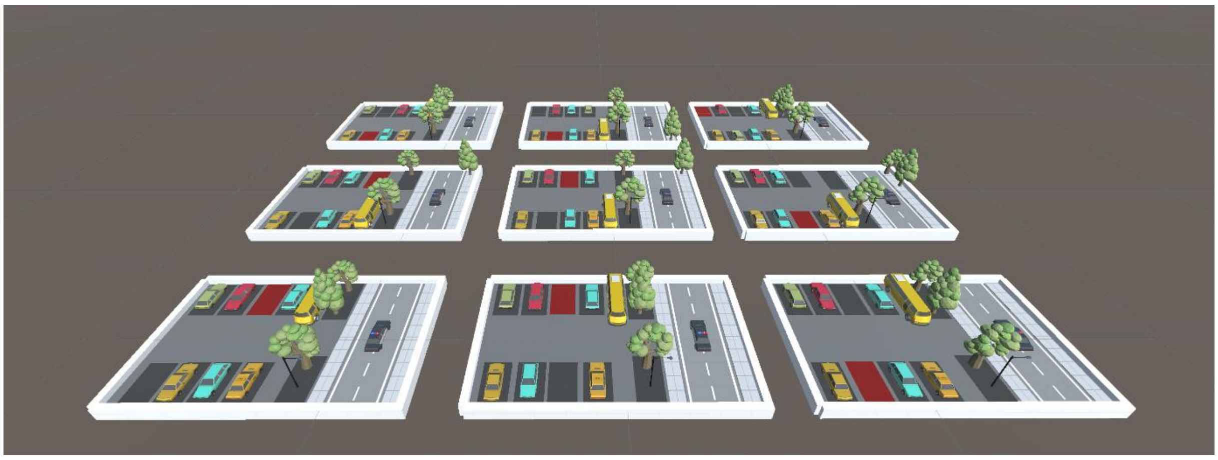

Figure 2: We used different parking setups to prevent overfitting.

However, when training with our own implementation of the PPO agent, we could only place one instance of our environment in our level. For this reason, we decided to always leave the target parking space in the same place, as otherwise the training time would have been unreasonably long.

3.2 MLAgents

The team behind MLAgents implemented PPO with the help of PyTorch. The developers give you the option of configuring the agent yourself in most respects via config files. You can set the batch size, learning rate, buffer size, beta, epsilon, lambda and number of epochs yourself. Furthermore, you can decide whether the learning rate should decrease over time and whether the inputs should be normalized, which can be a relevant factor especially for continuous control.

As hyperparameters we used a batch size of 512, a buffer size of 20,480 and a learning rate of 0.0003, which was linearly reduced over time. Furthermore, we defined Beta as 0.2, Epsilon as 0.2 and Lambda as 0.95. The actor model consists of three layers with 256 units each. Finally, we specified the maximum number of timesteps for the entire training with 100 million, while the time horizon has been set to 512.

As described in chapter 6.1, we ran the training of our PPO agent twice. Once with only one fixed target parking space and another time with different parking spaces that differed in the arrangement of the vehicles and the target parking spot.

3.3 Own Implementation

After using MLAgents at the beginning to test our environment and getting first results, we exported the environment from Unity and implemented PPO ourselves in Python using TensorFlow.

3.3.1 Code

We could not use the ReAlly framework presented in this course because it is incompatible with the exported Unity environment. Therefore, we developed a corresponding agent on our own.

In contrast to our implementation in MLAgents, we decided to rely on a continuous action space in our own version of PPO. We did this because it is much easier to implement three continuous output neurons than to define several different output layers for discrete actions. Besides, we ensured that our model is relatively flexible and can be used for a whole range of different continuous environments. Accordingly, we have successfully tested it with the ContinuousLunarLander environment.

For more details concerning our code, we refer to the readme file in our Github repository and the corresponding comments in our code.

3.3.2 Hyperparameters

As explained in chapter 6.1 we were able to only use one instance of our environment with our own PPO implementation, as it is not possible to use several instances at once in exported Unity environments. In our training, both Critic and Actor have a learning rate of 0.0001. As far as the networks are concerned, both have three dense layers with 256 units each. We use ReLu as the activation function and Adam as the optimizer. The batch size of 4096 is considerably larger than in our MLAgents implementation, as this resulted in better learning behaviour in this instance. The buffer size, on the other hand, is also twice as large as with MLAgents, at 40960. We used 0.2 as the clipping value for the loss.

4 Results

Subsequently, we will present the results of our trainings. We will first present the results achieved with one fixed parking spot and then show what impact having a more diverse environment has on training and performance.

4.1 MLAgents

4.1.1 One fixed environment

The training with only one fixed environment took a total of 17 hours and 22 minutes. When looking at figure 2, which is showing the achieved average reward over time, one can see a relatively long stretch at the beginning, in which it looks as if the agent is hardly making any progress for the first 6 million timesteps. However, the representation is somewhat deceptive at this point, as an observer could clearly see when visually observing the agent in the environment that the car has understood over time that it gets a positive reward when it approaches the target parking spot. However, the agent still collided so often during the time and never ended the episode by successfully parking, so the negative reward of the timesteps here largely overshadows the small successes in learning. However, as soon as the agent has learned to reach the target parking spot at least sometimes, the average reward increases strongly. This can also be seen in the fact that the average length of each episode starts to drop at the same time.

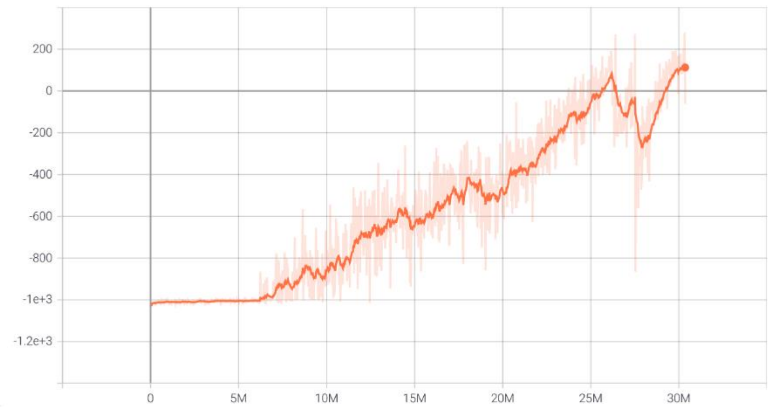

Figure 3: Cumulative reward over time in a fixed parking space.

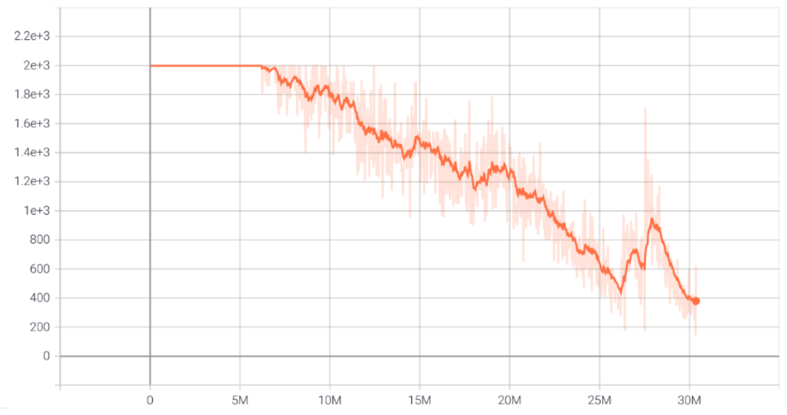

Figure 4: Average episode length over time.

At this point, one might be wondering about why the average reward around timestep 27 million has dropped so significantly. The simple reason is that we paused the training at this point in order to adjust the reward function a little bit. We noticed that the agent very often accepts collisions in order to get into the target spot quickly. Accordingly, we have increased the negative reward that the agent receives when it crashes into an object. Therefore, the average reward has dropped significantly for a short time. However, the agent learned this new fact relatively quickly and adjusted its policy accordingly.

4.1.2 Diverse environments

Compared to the training with one fixed parking space, the training with a wider variaty of free parking spaces took significantly longer. In total, it took almost 90 million timesteps and thus close to 50 hours for the average reward to reach a comparable level. This strongly suggests that the agent in the previous setup is somewhat inclined to overfitting since the parking procedure was always the same after the correction of the initial angle after spawning. Here, however, the agent had to learn how to park in different parking spaces, which made the approach angle and the necessary corrections much more complex.

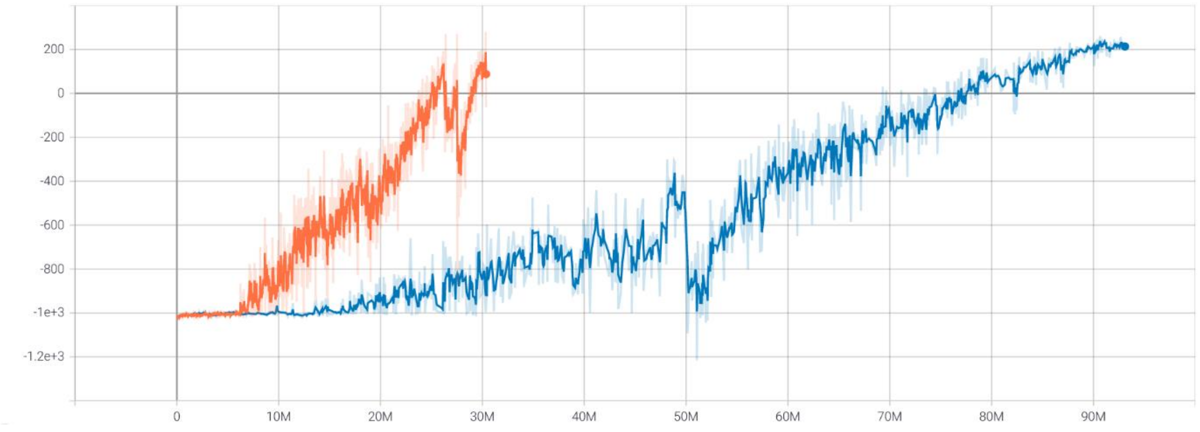

Figure 5: Average reward of training with diverse environments (blue) and fixed parking space (orange).

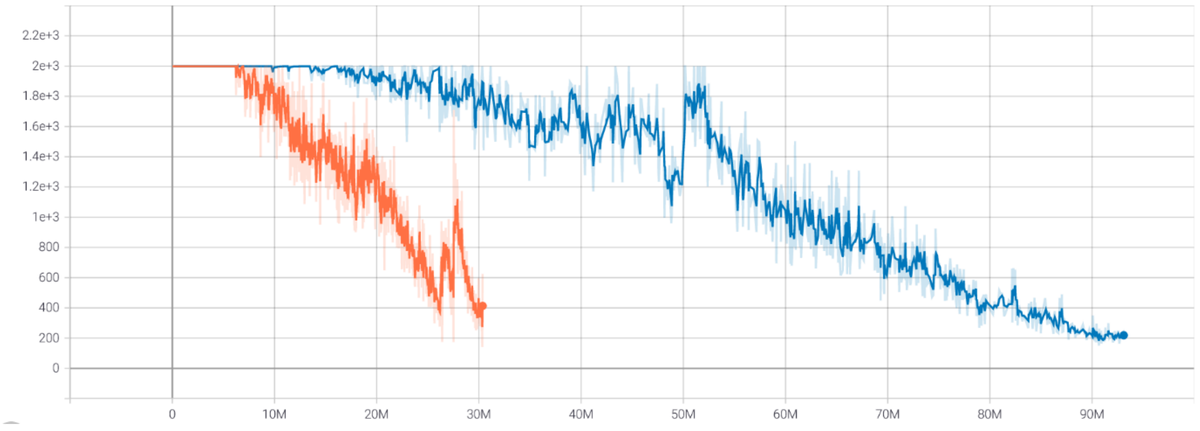

Figure 6: Average length of episode during training with diverse environments (blue) and fixed parking space (orange).

At this point, the drop in the average reward around the 50 million timestep could be noticed by some. This is again due to the fact that we have increased the negative reward for crashing, as the agent has hardly taken this into account before. However, it recovered from this relatively quickly, even if this recovery took a little longer than in the previous setup.

4.2 Own Implementation

Training with our own implementation of PPO took much longer than with MLAgents for several reasons. First of all, we could only train with one instance of our environment, which has already been described in previous chapters. Furthermore, due to our own limited hardware and poor optimization, the speed of the exported Unity environment could only be increased by a factor of 10, while MLAgents in Unity was significantly faster. Last but not least, our agent unfortunately resorted to parking in reverse at the beginning of the training. This suboptimal policy had to be overcome during the training, which also took some time.

In the end, due to time constraints, after 4 days of training we decided not to run the training completely until the agent found the optimal policy (perfect parking). In our case, we stopped the training when it became obvious that the agent was learning similarly to our training runs in MLAgents. Thus, the agent learned relatively quickly that it would get a negative reward if it crashed into other objects. Then, the agent learned that it would get a greater reward if it tries to get closer to the target parking spot. When the training was stopped, the agent was in the process of learning how to change its initial position in such a way that it could easily approach the target parking space. Since all of this progress (similar to MLAgents) has little effect on the average total reward as this only starts to significantly increase once it reaches its target, we refer you to our video where you can see the agent's progress.

5 Conclusion

Overall, we have succeeded in reimplementing a PPO and used it to train a car in a unity environment to park itself. Furthermore, we showed how making an environment even a little bit more complex drives up the required training time massively. Though this was to be expected the degree to which this happened was surprising. Building on top of this work it might be interesting to see how training time and the required depth of the neural network increases when the environments complexity and size is increased even further.