FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Kamil Závorka

Machine Learning for Simulated Military Vehicles

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot Ph. D.

Study programme: Computer science

Specialization: Artificial Inteligence

Prague 2021

I would like to express my thanks to Ing. Juraj Blaho for his technical lead and help with the implementation. I would also like to thank my thesis advisor, Mgr. Jakub Gemrot, PhD., for his advice and help with the thesis and especially with the text part of the thesis.

Title: Machine Learning for Simulated Military Vehicles

Author: Kamil Závorka

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph. D.

Abstract: Recent research in the field of neural networks has shown that this is a very promising area of artificial intelligence. Results of the research indicate that neural networks are currently able to at least match humans in many areas. One of the intensively researched sectors is the driving of autonomous vehicles. Although most people focus on autonomous vehicles in the real world, this new artificial intelligence can also be beneficial for driving in the digital world. As more and more activities and experiments are being moved from real environments to simulated environments, the demands on the quality of artificial intelligence found in digital environments are also increasing. The aim of this work was to explore the possibilities of artificial intelligence based on deep feedback learning in the field of parking simulated vehicles. Based on this research, we created a prototype neural network and evaluated this prototype during parking in a simulated environment.

Keywords: Artificial Intelligence, Machine Learning, Navigation, Simulation

Contents

1. Introduction 3
  1.1. Goal 3
  1.2. Approach 4
2. Background and related work 5
  2.1. Reinforcement learning 5
  2.2. Policy 6
  2.3. State-value function 6
  2.4. Action-value function 7
  2.5. Bellman equations 8
  2.6. Q-learning 9
  2.7. Deep reinforcement learning 9
  2.8. DDPG 10
  2.9. TD3 11
  2.10. Demonstration buffer 13
  2.11. Self-driving cars 14
3. System implementation 16
  3.1. Director 16
    3.1.1. Ongoing evaluation 16
    3.1.2. Experience replay buffer 17
  3.2. Unity Environment 17
    3.2.1. Episode 18
    3.2.2. Simulation threads 19
    3.2.3. Episodes exporting 20
  3.3. Learning algorithm 20
  3.4. Communication 21
4. Development 22
  4.1. Parking to position 22
    4.1.1. Networks inputs 22
    4.1.2. Network outputs 23
    4.1.3. Rewarders and done states 24
    4.1.4. Network architecture 25
    4.1.5. Results 27

4.2. Parking with orientation 28
4.2.1. Network inputs 29
4.2.2. Network outputs 30
4.2.3. Rewarders and done states 30
4.2.3.1. Stepped rewarder 31
4.2.3.2. Updated stepped rewarder 32
4.2.4. Network architecture 32
4.2.5. Results 33
4.3. Parking between obstacles 35
4.3.1. Network inputs 36
4.3.2. Network outputs 38
4.3.3. Rewarders and done states 38
4.3.4. Network architecture 39
4.3.5. Results 41
5. Final solution evaluation 44
Conclusion and future work 47
Bibliography 48
List of Figures 50
List of Tables 51

1. Introduction

Self-driving cars are currently an intensively researched topic. As the capabilities of AI grows and AI outperforms humans in many areas, people started to ask questions whether self-driven cars could replace human drivers. For these purposes are massively used neural networks with deep learning methods. There are also many frameworks and related works that are covering this topic, but the real world is not the only place where self-driving cars could be used. There is also a sphere of computer simulations and games which also contains cars that sometimes have to be self-driven. For having such cars in the software there are many existing AI approaches on how to run them. The complexity of those approaches depends on the demanded quality of the product. For high quality simulations, there is a requirement that the virtual cars should have capable AI drivers which should resemble human capabilities. This requirement opens the question whether some of the reinforcement learning methods can be used for simulated vehicles as well to obtain AI capable of driving more like human drivers.

Even though for both environments (real and simulated one) we are trying to achieve almost similar capabilities, the characteristics of those environments are different. The main difference is in the way of obtaining inputs. While in the real world, the input could consist of camera shots or some lidar measurements, in the simulated environments we have much more detailed information about surrounding objects and terrain, like some kind of precise heightmap or information about precise shape and structure of obstacles even from spots the vehicle cannot see. Such information simplifies the input of the AI and can be processed much faster than for example images. With this faster processing comes another critical point of simulated vehicles and that is speed of obtaining AIs decisions. In simulations, there are usually multiple vehicles that are driving at the same time. Because of that, we want the AI for the vehicle to be also efficient in terms of performance so the simulation steps can evaluate as many AI drivers as possible in real time. To achieve this, the programmers sometimes have to trade complexity for efficiency. That also might be different against the real world driver AI, where reliability and safety is in the first place.

1.1. Goal

The goal of this work is to propose a learning algorithm for a problem of vehicle parking and to train a network able to solve the problem. We define our parking problem as a situation where a vehicle needs to park in a specific position with proper orientation while the area around the target position can be limited by static obstacles. The network should be able to park the vehicle in the specified position, have specified heading and avoid colliding with the obstacles.

1.2. Approach

To solve the problem, we decided to implement our own environment simulating a vehicle using the Unity engine. In the environment there will be defined all needed things like the goal representation, the obstacles and functionality allowing a neural network to control the vehicle and get feedback.

For the reinforcement learning we decided to create an implementation of the TD3 algorithm. The implementation will be done in Python, using the tensorflow platform and the python code will communicate with the Unity environment to transfer information needed for learning.

For the process of the network development, we chose to start with a less difficult instance of a problem and if we are able to solve this easier instance, we iteratively raise difficulty up to the final problem instance.

2. Background and related work

2.1. Reinforcement learning

Reinforcement learning is one of the methods of machine learning. It is based on direct interaction between agents and the environment. The learning algorithm starts only with a set of actions it can perform and an observation of the environment state. The learning process is finding out which actions should be taken in actual state to maximize numerical reward for its behavior. At the beginning the learner does not know anything about the actions it should take so the process of learning is also the process of discovering which action brings the biggest reward in the current situation.

In the world of machine learning one of the most known methods is supervised learning. “Supervised learning is learning from a training set of labeled examples provided by a knowledgable external supervisor.” (Sutton, Barto, 2018, p. 2). That means that supervised learning requires some kind of dataset containing labelled data. The neural network is then using these training data to find some features describing what label should be given to the example. The goal of this learning is to generalize labeling of examples, so it can correctly label even inputs that are not directly presented in the training dataset, but which somehow contain similar features. For the absence of a training dataset, it is clear that reinforcement learning is different from supervised learning.

Even though reinforcement learning does not use any labeled training data, it also differs from unsupervised learning, “which is typically about finding structure hidden in collections of unlabeled data.” (Sutton, Barto, 2018, p. 2). Instead of that, reinforcement learning is trying to maximize its reward obtained from the environment so in terms of definitions, reinforcement learning is a different type of learning than supervised learning and unsupervised learning.

As mentioned, this method does not require any training dataset but instead the agent is directly reacting within the environment and based on the reaction it gets feedback saying how successful his reaction was. Based on this feedback, the agent learns himself, so in the next run in the environment he can try to behave differently to obtain better feedback or behave the way that brought him the best feedback in the past.

The reward signal maximization implies other things typical for reinforcement learning. The feature that also distinguishes this kind of learning from others is that reinforcement learning has to deal with balancing exploration and exploitation. These two parts of the learning process are in a ratio and it has an impact on the effectiveness of the training process. “To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward.” (Sutton, Barto, 2018, p. 3). Using these effective actions is called exploitation – the agent behaves the best way he knows to maximize reward. However, one cannot say that this behavior is the best possible way, what if there are actions which would bring even bigger rewards?

The only way to let the agent find truly the best possible action for specific situations is to let him try other actions even though we don't know their real reward income yet. For this purpose, reinforcement learning also involves some kind of exploration which sometimes forces the agent to not use his currently best action for certain situations. Instead, the agent uses an action which is not expected to bring such a big reward, but later it might show up, that the action is more effective.

There are many ways of implementing exploration in reinforcement learning (using random actions to explore, adding some noise to all actions, ensuring that actions that were not used in current state yet will be used etc.) and also many ratios between exploration and exploitation. "The exploration–exploitation dilemma has been intensively studied by mathematicians for many decades yet remains unresolved." (Sutton, Barto, 2018, p. 3).

2.2. Policy

The core of the reinforcement learning and resulting algorithm is a policy. The policy is a function transferring the state of the world to the action. "Roughly speaking, the policy is a mapping from perceived states of the environment to actions to be taken when in those states." (Sutton, Barto, 2018, p. 6). Policy can be implemented in many ways, and it strongly depends on the environment and format of actions, e.g., different policies are chosen for continuous actions and for discrete actions. The policy is the main output of learning process. In application we want to have trained policy that would give us actions performed by the agent.

2.3. State-value function

To learn a policy to perform the best possible action we need to have some kind of feedback telling us whether an action is good or bad. In scope of single action we have a reward signal from the environment. However, for the optimal way we also have to take into account rewards that will be obtained later while achieving a goal. For this purpose, we use a state-value function. It is a function that takes a state of an agent as an input and returns a reward expected to be accumulated during the run. This function is useful for policy updates because it considerates runs as a whole. Some action that has high immediate reward gain now can be found as less efficient than action that has lower immediate reward gain but leads to states with high reward gain.

Each run of the agent can be described as a sequence of steps. Each step t is composed of initial state S_t, action that is used in the state A_t, received reward for the action usage R_{t+1} and new state that agent reached S_{t+1}. For the run we can define state-value function for state s and policy π in step t as:

$$v_\pi(s) = \mathrm{E}_\pi[G_t|S_t = s] = \mathrm{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right]$$

Where Gt is a sum of rewards collected from the current state till the end of the run.

In the equation we can also see γ which is the so-called discount factor. This factor is a value from the range of [0,1] and it is used to specify how much the agent should care for the reward in the distant future. The lower the factor is, the more the agent adjusts its behavior to reward gain in a small time period. If the discount factor has value of 1 the agent does not have any motivation to finish his task as soon as possible, because the expected reward he collects will be the same. The other reason to keep the discount factor lower than 1 could be to prevent the reward sum being infinite for potentially endless runs.

2.4. Action-value function

Another important part of reinforcement learning is an action-value function which estimates the effectiveness of an action used in a current state. This function takes a pair of a state and an action applied in the state as an input and returns expected collectible reward after applying the action. This function is typically unknown for us, because otherwise we would be able to create an optimal policy by just taking actions with the biggest action-value function return for current state.

The estimation accuracy should be also improved during the run of the learning algorithm. The policy uses this estimation to improve its action prediction and the action-value function then uses rewards collected to improve its estimates, the more action-state pairs were explored the more can be improved estimation. This basically means that the policy can be good only in case we have a good estimation of the action-value function, especially for states that are promising in a way of reaching the goal. Vice versa as the policy is improving, it should tend to reach more promising states more often than non perspective ones and thus the action-value function estimates should have more explorations to learn from.

The important feature of action-value function is that the estimated reward does not estimates only the reward for the following transition. It is rather the estimate of the biggest possible sum of rewards that can be gained during movement to the goal position from the state starting with usage of action.

Like state-value function we can define action-value function for state s, policy π and action a in step t as:

$$q_\pi(s,a) = \mathrm{E}_\pi[G_t|S_t = s, A_t = a] = \mathrm{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right]$$

2.5. Bellman equations

During the process of learning, we want to improve the policy and the value functions. Improving means that the policy is capable of gaining bigger reward than before and the value functions do more precise estimations. Thus, for finite-horizon tasks or for discount factors lower than 1, we can define the optimal policy with the optimal value functions. The value functions are always unique, but there can be more optimal policies.

The optimal state-value function and action-value function can be defined as:

$$v_*(s) = \max_\pi v_\pi(s)$$

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

With this definition, we can define the optimal policy π_* as any policy that satisfies:

$$v_{\pi_*} = v_*$$

Using the optimal action-value function, the optimal policy can also be defined as:

$$\pi_*(s) = arg \max_a q_*(s,a)$$

For the value functions there also exist some relationships between the estimate of a state and the estimates of its successor states. These relationships can be described by the Bellman equation. The Bellman equation comes from the dynamic programming, and it is defined as recursion:

$$v_*(s) = \max_a \mathrm{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$

If this equation is valid for every state of the environment, the resulting value function is optimal. If we want to calculate the optimal value function for the environment, we can use the assignment described by the Bellman equation.

Let us say that one iteration will execute the assignment described by the Bellman equation for all possible states. If we define starting values as:

$$v(s) = \begin{cases} 0 & for\ s\ being\ terminal\ state \\ -\infty & for\ s\ being\ non-terminal\ state \end{cases}$$

Then after performing a finite number of iterations the value of each state will stabilize and the resulting value function will fulfill the Bellman equation and thus will be optimal.

This algorithm for getting optimal state-value function however requires us to have a fully observable environment and to have information about all types of actions. This information is not typically available and thus there are algorithms which uses the idea of Bellman equation for partially observed environments.

2.6. Q-learning

In reinforcement learning we distinguish two main groups for learning methods. The groups are called on-policy and off-policy methods.

The on-policy methods evaluate and improve the same policy. The policy is used for controlling the agent while exploring and at the same time it is being updated according to state-value function.

On the other side the off-policy methods separate these two processes. There are two policies, the behaviour policy which is controlling the agent while exploring and the target policy which is being evaluated and improved. Those two policies can be unrelated. This off-policy separation allows using the deterministic target policy which is focused only on obtaining the best possible rewards and the behaviour policy with some kind of exploration.

“One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning (Watkins, 1989)” (Sutton, Barto, 2018, p. 131). Q-learning is one of the algorithms that are adapted to a partially observed environment. The main idea is that the Q-learning is doing the action-value function updates while the agent is running and exploring the environment.

The action-value function value for Q-learning method (also called Q-function) is defined as:

$$q(S_t, A_t) = q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a q(S_{t+1}, a) - q(S_t, A_t) \right]$$

This formula estimates the action-value function for a given state-action pair no matter where the action comes from. From the formula it is noticeable that the estimated value for the actual state-action pair depends on the best estimate of the succeeding state. This estimation does not have to be precise, but as the agent is exploring the environment more and more those estimations are being improved.

In the equation there is also a parameter of α which is called a step size. The value of the step size is from the range of (0,1] and it denotes how much the old value is replaced by the new one. In the fully deterministic environment we can use a value of 1 as the new value is stable, but for stochastic environment it makes sense to do only a partial step towards the new value as the resulting value for the same state-value pair could not be stable.

2.7. Deep reinforcement learning

Deep neural networks are widely used for nonlinear function approximation. Because of their impressive results in machine learning systems, they are also widely used for reinforcement learning systems for nonlinear functions approximation. The advantage is also good generalization capability. The same deep learning methods that were discovered for supervised deep learning can be used for these networks.

The typical usage is for the approximation of the policy function and the state-value function which are the base of the reinforcement learning.

2.8. DDPG

Deep deterministic policy gradient (DDPG) is an algorithm based on the Q-learning algorithm. The algorithm is based on concurrent learning of policy function and Q-function at the same time. It uses the Bellman equation to train the Q-function. In the reinforcement learning terminology, the policy function is called actor and the Q-function is called critic. That is why these systems can be also known as actor-critic algorithms.

“This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $q_*(s,a)$ then in any given state, the optimal action $\pi_*(s)$ can be found by solving:” (OpenAI: DDPG, 2018)

$$\pi_*(s) = arg\max_a q_*(s,a)$$

Important aspect of the DDPG algorithm is that it is designed to be used for environments with continuous action space. This feature comes from a way how we compute maximum over actions from action-value function estimation of the given state. For discrete actions this is easy, because we can just go through all actions that can be used by the actor at a given state. For continuous action space this is impossible, because we can not effort evaluate the whole action space at every step of the agent and the optimization problem is also too expensive to be done in each step.

To be able to work with continuous action, the DDPG has to introduce a specific way how to train the actor and the critic network.

The first thing commonly used by deep reinforcement learning algorithms is an experience replay buffer D. It is a data structure that holds previous experiences. It is filled by the agent while moving through the environment. Instead of directly using and forgetting observed situations, we store them in a buffer, so the learning algorithm can use them also later. The data are stored as transitions, which are composed of 5 elements (s,a,r,s',d) where s is an original state, a stands for an action used in the original state, r is a reward obtained for the action usage, s' is a new state after action being used and d says whether the new state is terminate or not. The value of d is equal to 1 if the state is terminal and 0 otherwise.

To be able to use continuous action, we must update the way the action value function is updated. The basis for the learning is still the Bellman equation. Our goal is to learn our Q-function network having weights θ, so the Bellman equation is fulfilled for each state. To achieve that the DDPG uses a function calculating a mean square Bellman error:

$$L(\theta, D) = E_{(s,a,r,s',d)\in D}\left[\left(q_\theta(s,a) - (r + \gamma(1-d)\max_{a'} q_\theta(s',a'))\right)^2\right]$$

This function defines how far are the actual estimates of the action-value function from satisfying the Bellman equation. Minimizing this error is the goal of the learning process. Here we can use deep learning methods for computing gradient update out of the loss function. The potential issue that might be a source of unstability is that during the gradient update we are training the same network parameters θ which are used for the prediction of a target we want to reach.

To resolve this issue, the DDPG creates so-called target networks. The target networks are networks that are created as copies of the standard actor and critic networks. The purpose of the target networks is to have networks $(q_{\theta'}, \pi_{\mu'})$ that have parameters close to the parameters of the main networks (q_{θ}, π_{μ}), so it can be used for the loss function computation, but not the same so there will not arise any unstability. The target networks are thus a sort of time delayed images of the main networks. To keep them close to the main networks, the DDPG algorithm updates them using the polyak averaging:

$$\theta' = p * \theta' + (1 - p) * \theta$$

$$\mu' = p * \mu' + (1 - p) * \mu$$

Where p is a constant from the range of (0, 1) defining how much is the target networks parameters moved to the main ones.

Because finding a $\max\limits_{a} q_{\theta}(s, a)$ is a problematic for continuous spaces, the DDPG gets around that by using the target actor to predict action a, that approximately maximimize the $q_{\theta'}(s, a)$ as:

$$\max_{a} q(s, a) \approx q(s, \pi(s))$$

With having the target networks, we can define the mean square Bellman error as:

$$L(\theta, D) = E_{(s,a,r,s',d) \in D}\left[\left(q_{\theta}(s, a) - (r + \gamma(1 - d)\, q_{\theta'}(s', \pi_{\mu'}))\right)^2\right]$$

2.9. TD3

The DDPG algorithm itself has some limitations that could restrict its capabilities. To solve these limitations the TD3 (Twin delayed DDPG) algorithm was proposed. It comes out of DDPG principles, but few updates are involved.

One of the limitations of the DDPG algorithm is that the Q-function estimation often overestimates the real Q value. The overestimation error grows and it can lead to breaking the action estimation function as it strongly depends on the Q-function estimation.

To address this issue the TD3 algorithm adds 3 features that should reduce the overestimation phenomenon.

1) Creating a pair of the critic networks that estimates Q-function independently. The resulting Q value is taken as the minimum of their outcomes.

2) The actors are updated less frequently than the Q-functions. In the original DDPG there is an update of the actor for each update of the Q-function which could lead to the less stable action predictions. The original paper (Fujimoto, van Hoof, Meger, 2018) proposes using one actor update for two Q-function updates. This actor delayed update helps to keep the actor network more stable because from the poor Q-function estimation comes poor policy. If the policy update is delayed the Q-function has more updates to reduce the error and make the estimation stable before the policy is updated according to it.

3) Smoothing of target policy. This method basically adds noise to the actions that are predicted by the policy. This noise is supposed to prevent the actor from exploiting actions that can be the result of the Q-function error. If there is an overestimation for the specific action in the current state, the noise will reduce the exploitation of this action by using a random action from a small area around the target action in the action space. The noise is clipped so the resulting random action is still close to the target one.

With usage of these three updates the resulting algorithm can be described by a pseudocode shown in the Figure 2.1.

Algorithm 1 Twin Delayed DDPG

```
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$, $\phi_2$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
3: repeat
4:    Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:    Execute $a$ in the environment
6:    Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:    Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:    If $s'$ is terminal, reset environment state.
9:    if it's time to update then
10:      for $j$ in range(however many updates) do
11:         Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:         Compute target actions
               $$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \quad \epsilon \sim \mathcal{N}(0, \sigma)$$
13:         Compute targets
               $$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$
14:         Update Q-functions by one step of gradient descent using
               $$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$
15:         if $j$ mod policy_delay $= 0$ then
16:            Update policy by one step of gradient ascent using
                  $$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$
17:            Update target networks with
                  $$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$
                  $$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$
18:         end if
19:      end for
20:   end if
21: until convergence
```

Figure 2.1: TD3 algorithm pseudocode (OpenAI: TD3, 2018).

2.10. Demonstration buffer

In addition to the classic TD3 algorithm there also exist some techniques for improving the learning process. In our thesis we decided to use demonstrations to speed up the training. The basic learning process of the TD3 algorithm is to run the agent in the environment and let him explore environment states and actions. This process can be improved by creating some limited training data that are repeatedly presented to the learning algorithm. These training data can be obtained by different algorithms for the problem or even by the human user. Thus, the demonstration buffer can be a good starting point for the algorithm which points him to a possible way to the goal from the beginning and from which the algorithm can start exploring and improving in a good way and not randomly.

2.11. Self-driving cars

In the field of self-driving cars there are several papers. However, the field of car driving is wide and many aspects of it can be researched.

One of the existing work that is focused on self-driving cars in the simulated environment is the work of [Wang, Jia, Weng, 2019]. In this paper the authors learn simulated vehicles to drive in a simulated environment on a racetrack. For us the interesting part is that the authors used a similar approach with reinforcement learning as we decided to utilize for the parking. The goal of their work however was to let vehicles learn how to follow the road, where the trajectory was said to be in the middle of a racing track. The episodes also include the need of avoiding collisions, in their case with other cars on a track.

From the area of vehicle parking, there is for example an existing paper by [Junzuo, Qiang, 2021]. In this paper the authors are solving the problem of vehicle parking, where a vehicle is experiencing a typical car park situation. The vehicle is coming from the side and the goal is to park on a free parking spot surrounded by two parked cars. The authors in the paper describe in detail the kinematic model of the vehicle and the output of the network is directly interpreted as the parameters for the model. These actions are steering of the wheels and the change of car's displacement. Also the environment is strict in terms of respecting parking lines.

This case is really specific. In our work we decided to solve a bit more generalized situation, where we do not have any defined parking lanes as our use case is not based on a car park, but on an offroad environment with randomized obstacles. Also one of our intentions was to learn the network to work with the engine thrust so the speeding up or slowing down has more realistic control.

Other interesting work focused on parking was done by [Zhang, Xiong, Yu, Fang, Yan, Yao, Zhou, 2019]. The authors are also trying to park their car in a lined parking spot. However, this work also includes a method for the parking spot detection so it can be used in real world problems. The approach author used for the parking algorithm itself is also similar to the approach we chose to utilize. The reinforcement learning algorithm for driving is built upon parking spots detection algorithm which results in coordinates of parking spot corners. Based on this input, the parking algorithm is supposed to return the desired steering angle which the car should take.

This paper uses real life methods and sensors to observe the surrounding environment, which is the step we do not have to take as we are interacting with simulated environments where all data can be directly available. The parking algorithm itself is a bit similar to the approach we took, however for our use case we also want a network to control the speed of the car, so it can speed up and slow down to find a way to park itself.

There are also works that are solving parking problems based on path planning, one of them is done by [Zhang, Chen, Song, Hu, 2020]. In the paper there is also a parking problem on a street that the authors are solving. Unlike the other mentioned approaches their network is not used for car control but for the planning of the sequence of actions to park the car properly.

The process starts with the spot detection algorithm, that creates and information about the empty parking spot. The whole algorithm uses a motion model of the vehicle to predict states of the environment during the process of planning. Then there is a neural network which takes the state of the environment and returns the probability distribution of different steering wheel angles. For the speed there is also a separate policy that defines speed of the car based on three speed phases (acceleration, stable movement and deceleration).

For the parking spot there is created a plan of parking using the P-MCTS algorithm. The search tree is composed of nodes where each node has edges for all possible actions leading to other nodes and some statistics filled by the search. Based on these statistics the tree search keeps choosing the most promising actions until it results in a state where the vehicle is parked on the parking spot.

This work solves the problem of parking in a different way by using the planning phase to create a chain of actions leading to the goal. It is another approach to the problem, which is different from the approach we choose to investigate. Also the resulting algorithm is meant to be used on real life vehicles vehicles.

3. System implementation

The whole project is composed of two key parts communicating with each other, an environment and a deep reinforcement learning algorithm. The simulation of the environment is done in Unity engine and the learning algorithm is written in Python. For communication there are also some intermediary structures. We can express the composed system as a diagram shown in the Figure 3.1.

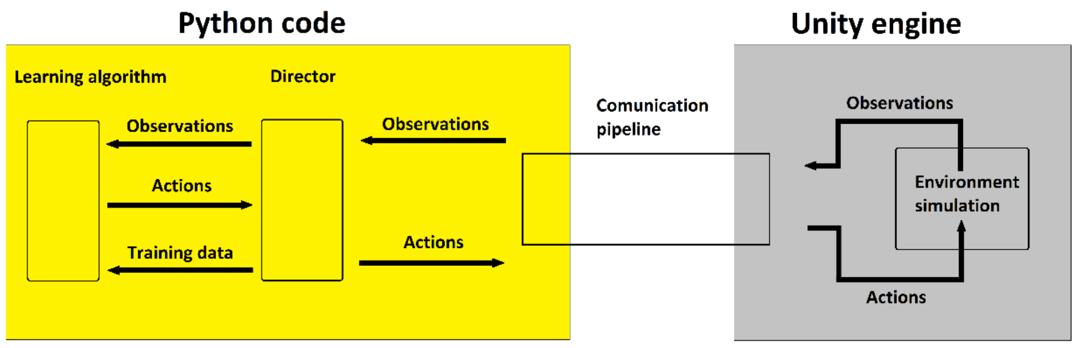

Figure 3.1: Scheme of the proposed learning system

3.1. Director

The Director is the part between the simulation and learning algorithm. It is written in Python and it ensures the initialization and run of the whole process. Each run starts with running the director code which initializes the learning algorithm class with all its neural networks and inner data. Then it runs the Unity application with simulation and starts to communicate with it. Once everything is initialized it takes care of receiving the observations and reward from simulation and sending the actions obtained from the actor network back to it. Besides that it also ensures creating batches with training data and sending them to the learning algorithm.

3.1.1. Ongoing evaluation

Because we want to get ongoing information about how well our network is doing during the learning process the director also ensures periodical evaluation of the network. This ongoing evaluation can help us to find an issue with learning as soon as possible so we don’t waste our time waiting until the learning is finished.

The evaluation is always done once after a specified amount of training episodes. We tried to choose this amount according to the difficulty of the problem, so the data gives us an image of how the network's success rate is growing. To provide general information about the network capabilities the evaluation itself is composed of a hundred separated episodes and the result is averaged from results of the episodes. The evaluations give us information about average total reward obtained by the vehicle, the average distance to waypoint from the very last state of each episode and the success rate of the network in achieving the defined goal.

The last output of the evaluation is the difference between the Q-function estimation of an initial state of an episode and the real total reward accumulated during the episode. This information does not really say anything about the network success rate and it gives just a rough idea about how well the critic is doing in estimating but it is useful when it comes to detecting some problems in learning, because we can easily see if the critic starts to do some massive overestimating.

3.1.2. Experience replay buffer

As a connection between the simulation and the learning algorithm the director maintains the replay buffer structure. It is a data structure holding transitions that the agent went through during the process of training. Each transition is a structure containing a previous state, an action and a new state obtained by using the action in the previous state. Also there is a reward obtained by the environment for using the action and information whether the new state is final or not. These transitions are inserted into the experience replay buffer by the director as the agent keeps exploring various states. These buffer data are being sampled into batches of training data that are passed to the algorithm for training.

Originally we implemented our replay buffer as an potentially infinite list, however it turned out to be a performance issue and after long runs even a memory issue. That is why we decided to implement the replay buffer as a large array which is being filled with transitions and after hitting its maximum capacity it starts to override the oldest transitions records. This turned out to be a good performance improvement and it prevents memory problems after runs for tens of thousands of episodes.

The replaye buffer is not really the only transition structure that our director is managing. The director also holds a demonstration buffer described in the chapter 2.5 which is a kind of similar structure but holding transitions that were pre-generated by different ways and loaded to the algorithm before the learning started. In our program we created those pre-generated episodes manually and stored them in a data file. Because there are two structures with transitions the director has to use each of them for composing a batch of training data which is then being sent to the network for training. In our system we use a constant ratio 1/3 for demonstration buffer records and 2/3 for experience replay buffer records. However this ratio is also a variable that can be tuned.

3.2. Unity Environment

The environment in which we want to train our networks is simulated by using Unity Engine. It is shown in the Figure 3.2 and it contains a car, its waypoint with orientation and some obstacles. In variants of this scene, we want to train our networks.

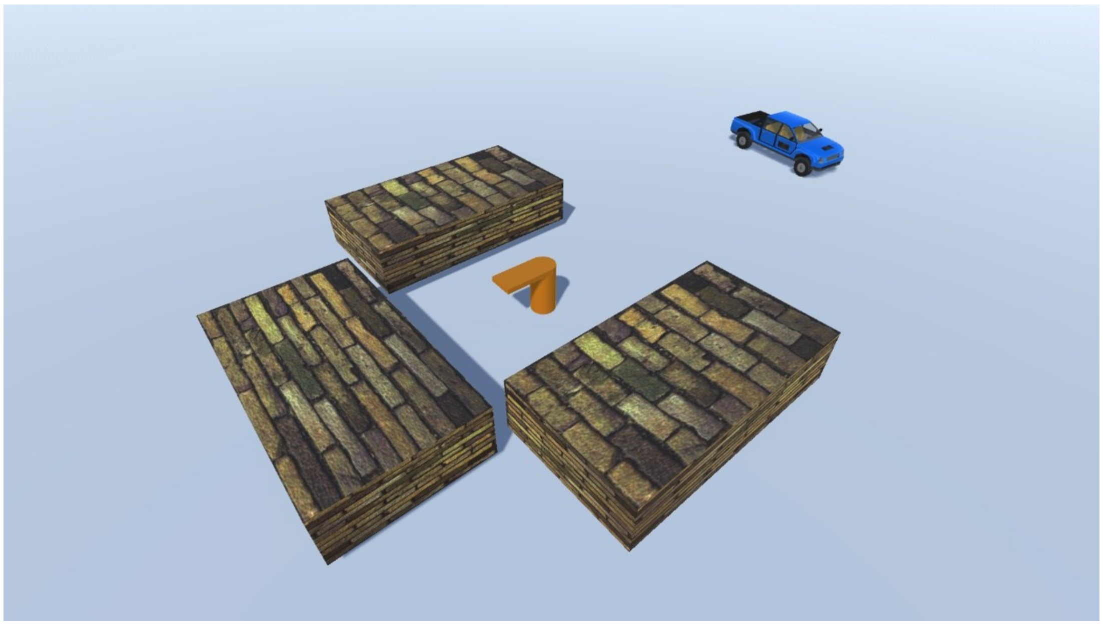

Figure 3.2: Screenshot of the environment instance

The vehicle simulation is based on a slightly modified MS free vehicle system asset [MS Vehicle System free version, 2018]. The asset contains a basic model of a vehicle and a code implementing vehicle physics in the environment. The modifications that we added to the code were related to the removing first person view cameras and some functionalities related to the vehicle manual control. We also removed some unnecessary functions like steering wheel rotations and skidding marks and added some functions needed for correct usage of the reinforcement learning network outputs for the car.

In Unity we also created a system for obtaining observations and rewards for the agent based on the simulation data that can be obtained from the Unity engine (distances, raycasts etc.). These functions have different variants implemented for different problem definitions and also to try different approaches during the learning.

3.2.1. Episode

As mentioned before, the environment contains a car, obstacles and a waypoint which marks the target, where the car is supposed to park. Because parking can be considered as related to a specific location, we decided that the car should be operating in a circle of 30 metres around the waypoint. There was no reason to consider parking for longer distances as the car should always get closer before starting some advanced maneuvers.

The simulation episode starts with spawning. Because we decided to limit our problem to a 30 m radius around the waypoint the vehicle is spawned within this radius. Another restriction for spawning is, that we don't want to spawn the vehicle directly on the waypoint, so the network has to provide at least 1 action in each episode. As we also do not want to spawn the vehicle in a position where it would be colliding with some obstacle, we decided to do the spawning in a safe distance from the waypoint (15-25 metres). This distance ensures that the episode is not instantly ended by collision and it makes sure that the action sequence needed for the vehicle to get to the waypoint is not trivial.

When the spawn is done the episode starts to run. The episode itself can be divided into steps. Every step starts with a vehicle being in a specific state. This state can be described by a vector of numbers which we call observation. This observation is being sent to the network and while the python code is computing the actions, the simulation is waiting. When actions are received back from the network it is applied by the vehicle. When actions are set the Unity will continue in environment simulation of acceleration, collision checks etc. After a certain time of simulation run, the simulation is paused and the new observations are sent back to the component together with reward obtained for the last action usage. This one iteration of getting actions and doing simulation is called a step.

Because we didn’t want any vehicle to get stuck, we decided to limit each episode with the maximum possible number of steps. It means that if a certain step count is reached, the simulation is stopped and the network will get the information about the episode being ended. If the episode is ended the director can restart the environment using a function call to Unity. After receiving the call, the simulation will restart the environment with a new random position for the car and with a restarted step counter.

3.2.2. Simulation threads

To use the simulation environment effectively and to save some learning time, we decided to establish more cars in the environment learning to park at the same time. Practically it means that in the environment there are multiple cars, each having its own waypoint and its own obstacles. We call these cloned situations simulation threads and their main characteristic is, that they are completely separated from each other. None of the vehicle is able to find out in which simulation thread it is located, nor how many simulation threads are present in the environment. All simulation threads are simulated in parallel and one of the important features is that their step counters are synchronized and thus the episode starts and ends for each simulation thread at the same time.

The python director is adjusted to manage all simulation threads at the same time. That means it is able to keep observations for each vehicle separated and send back proper action for each vehicle. Even though the states of each vehicle are separated, obtained transitions are stored in the same replay buffer as they are not dependent on a particular simulation thread.

The important point is that regardless of how many simulation threads are running, the training of the network is always called only once for each step. This means that we don’t speed up training by adding simulations threads (the training speed can be increased by increasing the size of the batch sampled for training), but it helps us to populate the experience replay buffer faster and with more various explorations

However, having more simulation threads does not help only for exploration. Even as it does not speed up the training, it speeds up the evaluation, because evaluation episodes can be run on all simulation threads parallely. It means that if we have 4 simulation threads, we can run evaluation episodes for four cars at the same time. That lowers the time needed to complete 100 evaluation episodes to ¼ compared to the case where all episodes are done by a single car.

3.2.3. Episodes exporting

For getting demonstration episodes to speed up training we decided to use manually driven episodes. To obtain such data the simulation code must provide the ability to allow the user control the vehicle in the simulation by his own keyboard and drive in the episodes himself. In our simulation we created this functionality by creating a fake vehicle control agent, which is taking data from the keyboard input instead of the communication pipeline and also sending observation data to the data file instead of sending them back to the pipeline. Using the same functions ensures that the data that are stored in the data file are equal to data that would be sent to the python and thus it can be used during the training without any misinterpretation. While creating the demonstration episode, the user controls one vehicle and the other simulation threads are not running.

3.3. Learning algorithm

The reinforcement learning algorithm is implemented in Python next to the director. The director is responsible for creating, maintaining and calling learning algorithm functionalities. The algorithm we use is the TD3 algorithm described in the Chapter 2.4. The algorithm provides a simple interface for initialization of networks, running training iterations and obtaining actor and critic networks outputs.

Besides that the network is capable of saving itself into files as well as loading from those files again. This functionality is useful in several cases. It is definitely handy in the case that we run some long-lasting training and some error occurs like power outage, computer freeze or if we want for some reason just pause the run and continue later. For such cases we decided to create backup saves of all networks after each evaluation process (that means that after each few hundred episodes we always create a backup of the current state). Another case is that we cannot say for sure how many training episodes would be needed for satisfying results so we can either continue with training beyond the original estimate of episodes, or cancel the training earlier and still have all networks stored for later examination or practical usage.

3.4. Communication

The last important part of the environment is the communication pipeline between simulation made in Unity and the learning algorithm run in Python. For this purpose, we used the ML-agents library [Juliani et al., 2020].

The library is an open-source library that allows users to connect AI training algorithms to games and use them as training environments. The library itself provides several AI learning algorithms that developers can use for training their own AI. It provides several API for connecting users' own AI training algorithm to the environments. One of the supported environments is Unity.

For connecting to Unity, the library contains two packages – the Unity package and the Python package that communicate with each other by using a channel at a specified port.

The unity part of the ml-agents library defines a reinforcement learning interface through which we get regular requests for observations, rewards and done state values. The passed data is then sent to the Python package where our python code can read the data.

The library has several configurable options regarding the communication. One of the important parameters is a variable saying how often is the neural network requested to provide actions for the environment. If the delay between two consecutive actions is huge, the vehicle could have bad reaction time. But asking too often can be an unnecessary waste of computation time. This variable can be tuned up to perfect balance between performance and reaction time, however in our work we went with a fixed value of one action request per five simulation updates done by Unity.

4. Development

During our process of creating a solution, we decided to continue iteratively. At the beginning we started with a definition of a simple problem, far easier than the definition of our final use case. On this simple problem we created and tested our environment, learning algorithm and other functionalities. After being able to solve this situation, we continued with more complicated ones. While solving different difficulties of the problem we were also testing various approaches to learning and comparing how well they are doing.

4.1. Parking to position

The first instance of a problem we tried to solve was parking to a certain position. The target position is specified by a waypoint object. The car is supposed to drive to the position and stop there. The environment is shown in the Figure 4.2.

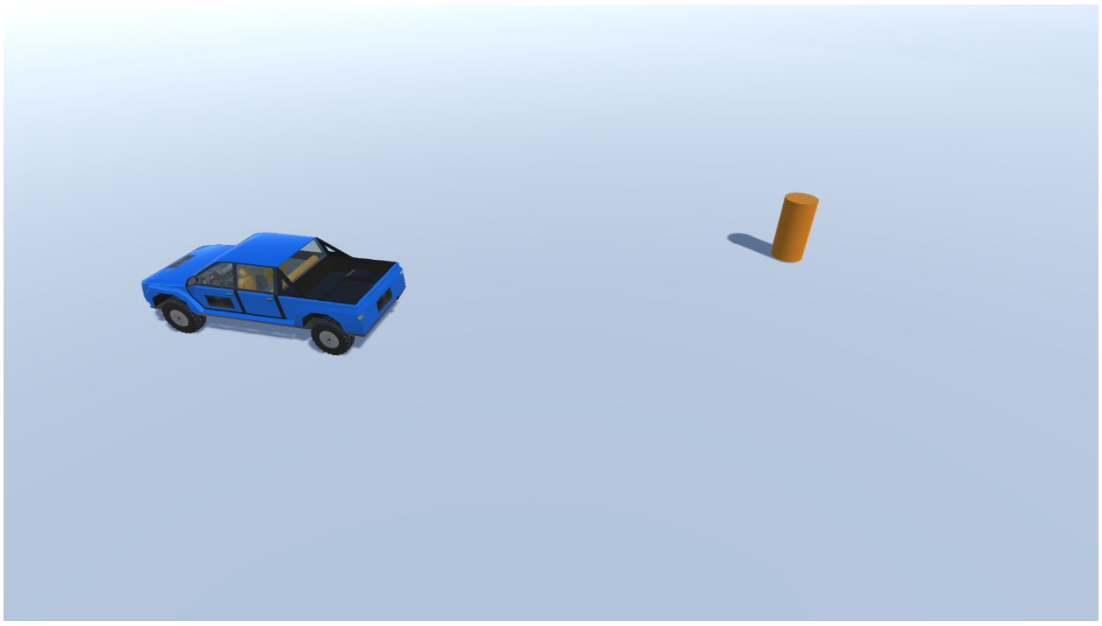
Figure 4.1: Environment instance of the parking to position problem

4.1.1. Networks inputs

In each episode the car is spawned in a random position which is further from the waypoint than 15 metres. Also the initial orientation of the car is random. To be able to localize itself and find a way to the waypoint the network gets three inputs. All inputs are normalized so they fit into the interval of [-1.0, 1.0]. This normalization ensures that there are not any huge values passing through the network. First observation is an angle between the vehicle's forward vector and the direction to the waypoint. This input basically says how much is the vehicle faced to the waypoint. The angle value is signed and normalized so the value of 0 means the vehicle is perfectly faced to the waypoint, thus values close to –1 or +1 means that vehicle is faced away from the waypoint.

The second input provides information about distance to the waypoint. The distance is a bit tricky because it cannot be negative, so the normalized value is also always higher or equal to zero. But also, the distance can be potentially infinite which is problematic for normalization. Because we do not expect parking to be performed in an infinite space we decided to establish 30 metres from the waypoint as a maximal distance and the normalization is done with respect to this value. That means that distance input is a number in the range of [0,1.0] where a value of 0 is equal to 0 metres and a value of 1 to 30 metres.

The third input describes the speed of a vehicle. If the car is supposed to stop on the waypoint it has to have some information describing how fast it drives. Based on this information the decision whether breaking is required or not can be made. Similar to distance even for a speed we can discuss what should be the upper bound for normalization. In our simulation we decided to establish an upper bound as 50 km/h. Obviously we can work with vehicles that can easily drive faster than this and there are also vehicles with lower max speed. For these cases the input normalization can be adjusted to reflect true capabilities. Speed as a physical quantity is always non-negative so it would be natural to provide normalized speed as a value inside [0,1.0]. However we also need to distinguish between driving forward and backward, because both movements distinguish different situations. Driving backward while facing the waypoint means that vehicle is getting further from the waypoint, but moving forward gets the vehicle closer. To distinguish between moving forward and backward we decided to give the speed a sign. That means that the speed parameter is actually a value from the range of [-1.0,1.0] where value of –1.0 represents vehicle driving backward with speed of 50 km/h, value of 1.0 represents vehicle driving forward with speed of 50 km/h and value of 0 means that vehicle is not moving at all.

The variance in a vehicle's capabilities and properties is also problematic if we want to use the same network for various vehicle types. In real world and thus in proper simulation, different vehicle types have different capabilities like acceleration, deceleration, max steering angle, mass, engine power etc. These capabilities have a direct influence on vehicles' physical behaviour while driving. If the network does not have inputs describing such properties, it probably won't be really efficient e.g. to let the network trained with a tractor to drive a racing car. To address this issue it could help either to use a separate network for different vehicle types or to let the network work also with vehicle capabilities and train with different types of vehicles.

4.1.2. Network outputs

The action we expect from the network to return are two elements. As we are normalizing inputs of the network, we also want to normalize output of the network so all outputs have values in the interval of [-1.0,1.0].

The first of the numbers we want to receive from the network is steering. This value represents how the wheels of a vehicle should be rotated. The value is in the normalized range of [-1.0,1.0] where positive values represent turning to the left while the negative value represents turning to the right. The absolute value represents how much the wheel should be rotated. In our experiments we interpret the steering as an absolute value and wheels get immediately desired rotation. This direct usage of steering can be a bit uncontinuous in simulated environments. In the real world we are not able to rotate the wheel immediately, instead it is rather a process of turning that we execute by rotating with the steering wheel on a vehicle.

If one wants to achieve this level of reality, some kind of PID controller can be established on the simulation side to simulate continuous change of a steering angle. Another way to go could be to add steering angle from the previous state as a network input and decrease reward of the network if the difference between the old steering angle and new one is large. Another possible approach could be to interpret steering output of the network not as an absolute value of a steering but as a steering angle difference that will be added to previous steering angle in which case the network also needs previous steering angle as an input.

The second expected output of a network is a thrust of the engine. This value also falls to the range of [-1.0, 1.0] where positive values are increasing velocity in the forward direction while negative values are increasing velocity in a backward direction. Of course the velocity cannot be set directly so setting a thrust is just an input for the process of continuous changing of vehicle velocity.

In the first version we also tried to use a third action output of a network representing the breaking of a vehicle. This action would better match to real world vehicle driving, however if we are interested only in vehicle capabilities, it turns out that the brake input is not necessary. For slowing down and stopping the vehicle it is sufficient to use reverse engine thrust and learning a network to give us just two actions is easier than learning a network for three actions.

4.1.3. Rewarders and done states

For the training we must define some states that are considered as final so we can end the episode and start a new one. The obvious final state is a state where the vehicle reaches the waypoint. After getting to the desired position there is no reason to drive further. However vehicles do not reach the waypoint all the time. Sometimes it might get stuck or to some states that are not perspective in terms of reaching a waypoint and restarting the episode helps the vehicle to return. For those situations we added counting of simulation steps and the episode is terminated after 250 steps as well.

During our experiments we found out that untrained actors might drive with vehicles far away from the waypoint. Even though we might say that states met at the distance are valid and there is a way back to the waypoints, exploring those areas is not effective. We want to properly explore mainly the area in a close proximity to the waypoint where all maneuvers will take place and we don’t need to waste our training computations for those unpromising states at all. For those reasons we decided to limit movement of agents and terminate the episode immediately if the vehicle drives further than 30 metres.

Important part of the network performance is a way how we compute rewards given to the network during the process of training. The reward function should be providing feedback about how well the actor works. In the first instance of a problem we decided to try two approaches to the rewarder.

First we used a simple rewarder based on a distance to the target. The reward between two states is calculated as a difference between distance to waypoint in previous state and distance to waypoint in current state. By moving toward the waypoint the vehicle lowers the distance and collects reward for that. The maximum reward the vehicle can collect is equal to original distance to the waypoint. Even though this reward function is simple, it requires some thoughts about how the function looks. What is a possible collectible reward and whether this reward in total is in reasonable bounds.

These thoughts brought us to an idea if we can define an easier reward function. That's why we tried a second approach, that gives reward only as a big one time reward in a final state after reaching the waypoint position. This function does not require any analysis of the accumulated reward – the only reward vehicle can collect is the reward in the end so there is no danger of accumulating reward out of some bounds. This kind of rewarding is much simpler, because instead of thinking of rewarding criteria, it just requires to properly describe the goal state.

Another motivation behind this need of simplicity could also be allowing creation of training environments to people that do not have much experience with programming. For those people it would only require the ability to describe the desired goal state to be able to train networks for some environment instead of creating some complicated function of simulation variables.

4.1.4. Network architecture

For the problem we used actor networks having 2 hidden layers of 25 neurons. The same layer width was used for critic networks. The networks schemes can than be described as in the Figure 4.2 and the Figure 4.3:

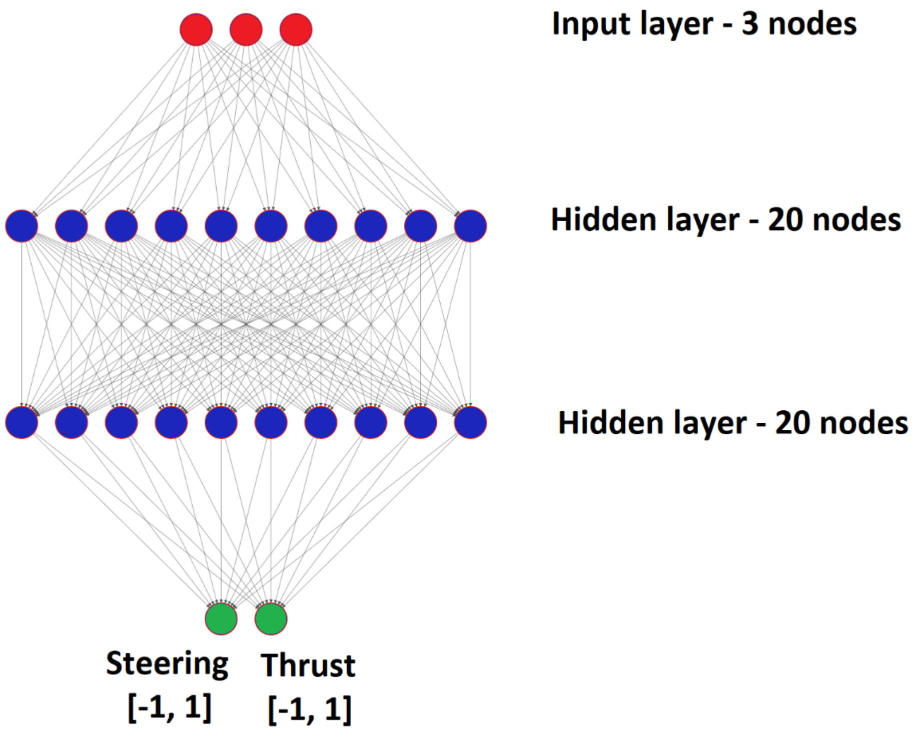

Figure 4.2: Scheme of the actor network for Parking on position problem

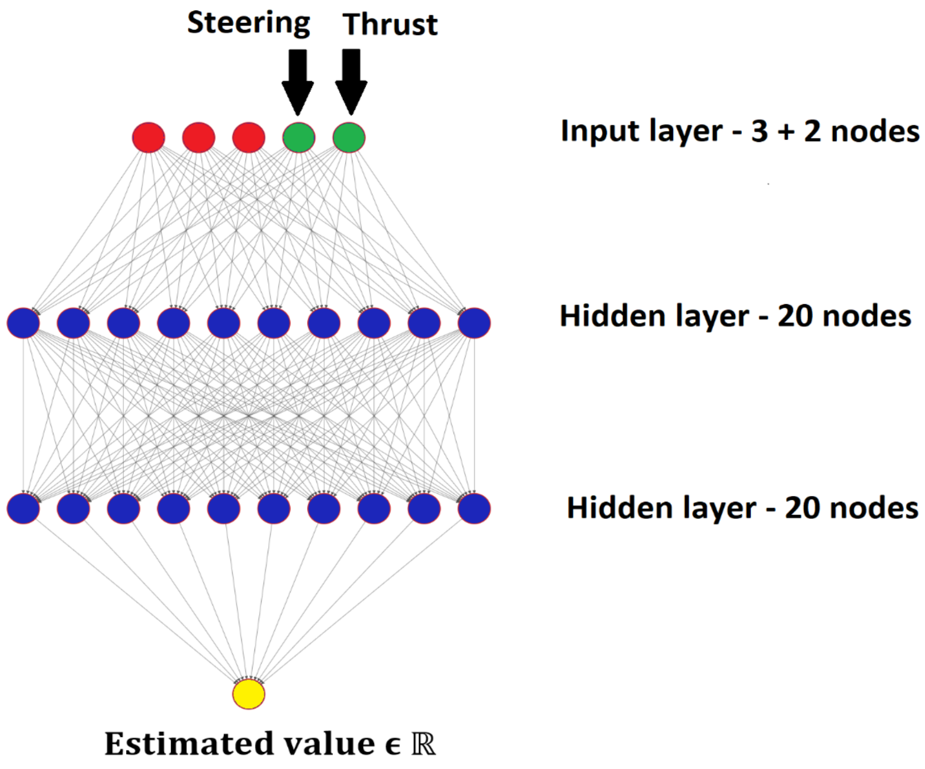
Steering
Thrust
Input layer - 3 + 2 nodes
Hidden layer - 20 nodes
Hidden layer - 20 nodes
Estimated value $\epsilon \mathbb{R}$

Figure 4.3: Scheme of the critic network for Parking on position problem

For the networks we used values of parameters that are listed in the Table 4.1. The parameters have the same naming as in pseudocode in the Chapter 2.9.

Parameter name	Parameter value
Replay buffer size (D)	1 000 000
Discount factor (γ)	0.99
Polyak (ρ)	0.001
Actor learning rate	0.001
Critic learning rate	0.001
Batch size (B)	200
Noise sigma (σ)	0.4
Noice clip (c)	0.2
Policy delay	2

Table 4.1: Parameters of the network for the parking to position problem

4.1.5. Results

The learning process of our network can be described using the evaluation episodes outcomes. The values we want to compare are the success rate of the network and the average remaining distance to waypoint position when the episode ends.

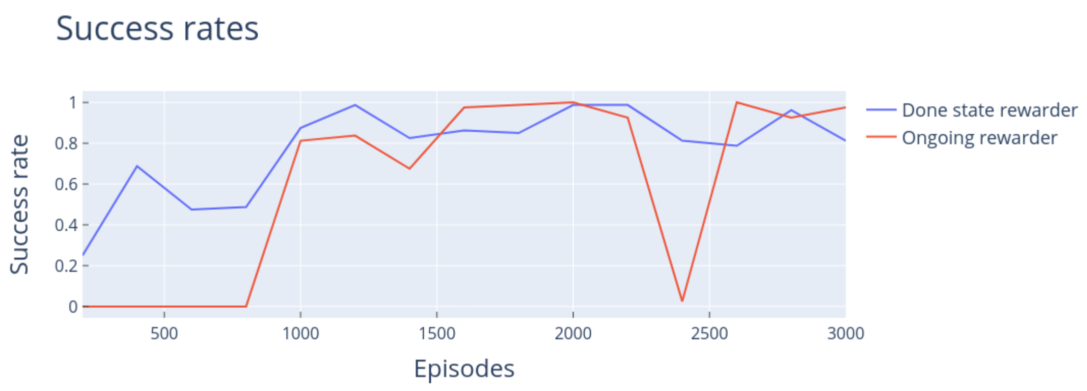

Figure 4.4: Success rates of the networks by the time of learning

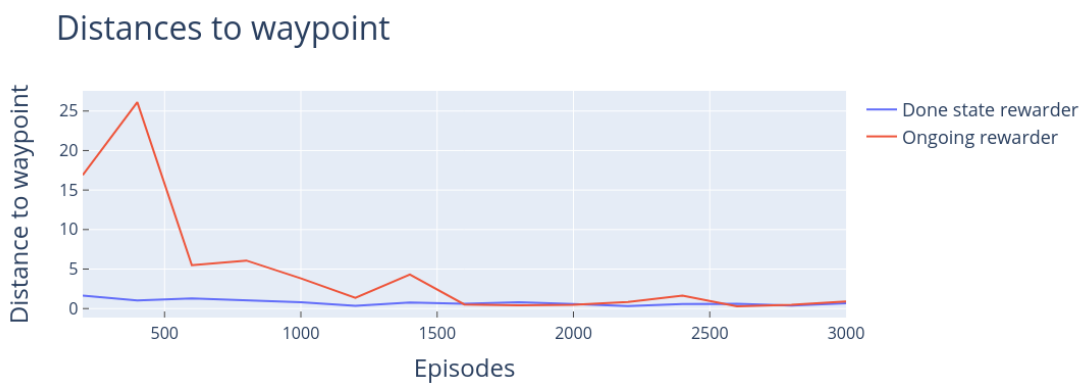

Figure 4.5: Final distances of the vehicles by the time of learning

In the Figure 4.4 and 4.5 we can see values measured during the evaluation comparing two used rewarders for the problem. After the 3000 episodes of training, both networks seem to have pretty similar outcomes, so both can be used for problem solving.

In both graphs we can see that the Done state rewarder is a bit worse at the beginning. This is probably caused by slow propagation of the reward information into starting states. The only source of reward defining the proper direction and the goal is to the network provided only in the final state. Thus, it takes time until the critic connects this information through the sequence of states leading to the goal position. For that reason the distance to waypoint is for the Done state rewarder in the first 500 episodes at values that could correspond to spawn position – the actor does not have information in which direction he should explore.

On the other hand, the Ongoing rewarder provides information about correct direction from the beginning, so the actor almost immediately starts to move to the waypoint. The actor may not know how to stop at the position yet – the success rate is not so high in the first 500 episodes, but the low distance to waypoint indicates that he is moving around the good position and explores how to stop there to gain the best reward.

4.2. Parking with orientation

Having a solution for the first step, parking to the position, we decided to increase difficulty. For the parking problem it is typical that we want to specify the state we want the vehicle to end in. This state could be specified by position and orientation. For that purpose, we added in this difficulty the orientation for the waypoint which specifies desired final orientation of the vehicle. The goal of the episode is thus for the vehicle to stop at the position of the waypoint with its absolute orientation being as close to waypoint orientation as possible. The screenshot of the environment can be seen in the Figure 4.6.

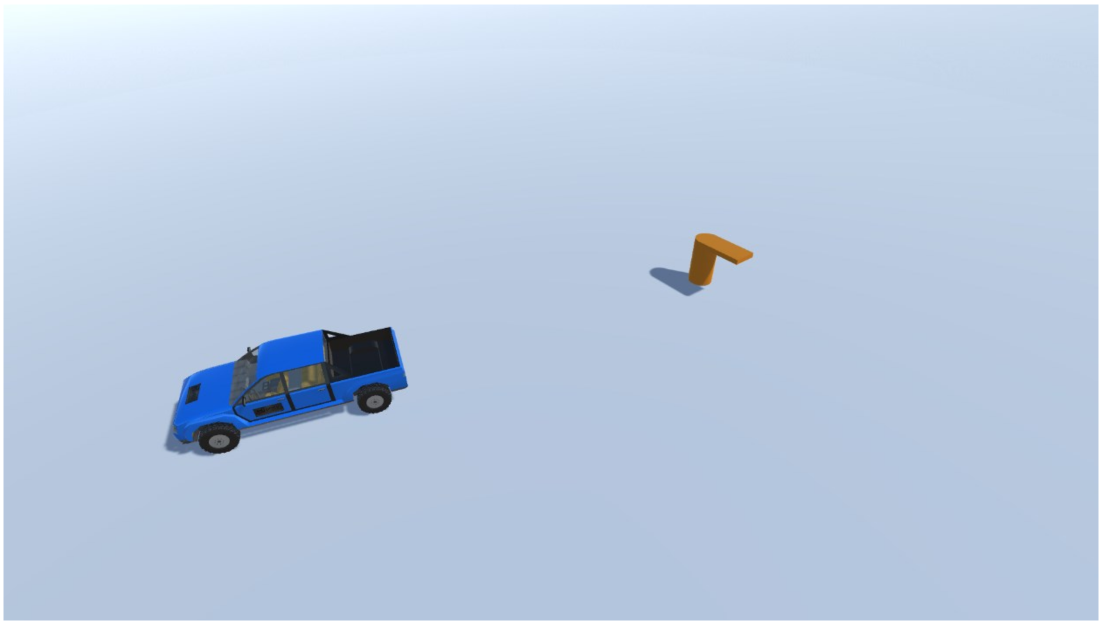

Figure 4.6: Environment instance of the parking with orientation problem

4.2.1. Network inputs

As this situation is a bit expanded instantiation of the environment from experiment in chapter 4.1 which we resolved with parameters specified in section 4.1.1, we adopted this part of the network.

For inputs we also want all values to be normalized in range of [-1.0, 1.0]. The first three inputs are adapted from the network described in the Chapter 4.1. The first input represents the angle between vehicle orientation and the direction to the waypoint. Its value is normalized to the range of [-1.0, 1.0] where 0 corresponds to zero angle and values approaching -1.0 respectively 1.0 corresponds to angles approaching -180 degrees respectively 180 degrees.

The second input gives information about distance between the vehicle's position and the waypoint’s position. This value is normalized to range [0,1.0] where 0 implies that both positions are identical and the value of 1.0 corresponds to distance of 30 metres which was chosen as a reasonable operational distance for the maneuver.

The third input is the value of the vehicle's speed. Even though speed can be only positive, the value is in the range of [-1.0, 1.0] where a value of 0 means that the vehicle is not moving at all. The values -1.0 and 1.0 correspond to the speed of 50 km/h which was chosen as a reasonable cap speed. We cannot say whether the vehicle won’t overcome this value, but it is not likely for the problem of parking. Even though the negative and positive value of the input represents the same absolute speed, there is a difference between them. The negative values are used for speed which is reached while driving backward, while positive values represent speed in forward direction. Thus with this sign the network gets information whether the vehicle is driving moving forward or backward.

These inputs were just taken from the inputs described for the network in the Chapter 4.1. However, for this problem we must introduce a new input signal, because the network has to get some information about the orientation in which it is supposed to park. This new input value is also normalized in the range of [-1.0, 1.0] and it represents the angle difference between the vehicle's orientation and the waypont's orientation. The value is received the same way as the first input and normalized as well. The value of 0 means that the orientation of the vehicle is aligned with the orientation of the waypoint. The values approaching -1.0 or 1.0 means that the vehicle is faced almost precisely to the other side.

4.2.2. Network outputs

Output of the network remains the same as we don't need any other parameter for vehicle driving. The first output is the desired steering of the vehicle in the range of [-1.0, 1.0] where values of -1.0 and 1.0 represent the maximal possible steering of the vehicle either clockwise or anticlockwise. The second output represents the throttle of the engine. The value is also in the range of [-1.0, 1.0] where values of -1.0 and 1.0 represent maximal possible engine throttle in backward direction or forward direction.

4.2.3. Rewarders and done states

Because we now want our vehicle to also meet desired orientation, the definition of episode being done is a bit changed. As for previous difficulty there are still done states related to count of steps in the episode – the episode also has limit of 250 steps and after meeting this limit the vehicle driving is stopped. Also the limitation for the distance that prevents vehicle to drive away from the space we reserved for the maneuver is valid. If the vehicle reaches a distance of 30 metres from the waypoint, the episode is ended so we don't explore space further and we don't train our critics for states that are not relevant for effectively reaching the goal.

The part that actually changes is the definition of whether the waypoint reaching is finished or not. In previous difficulty level the waypoint reach was defined as stopping in position really close to the waypoint position. Now if we want the vehicle to also reach the waypoint's heading, the definition of done has to involve orientation as well. For that reason we added a third condition that has to be met to actually declare run as successful and that is, that the absolute orientation of vehicle and the desired orientation of the waypoint do not have angle difference bigger than 30 degrees. That means that if the vehicle just reaches the position but with the bad orientation, it can still try to adjust the heading and make it as close to the requirement as possible. This change should be beneficial, because it is a chance for the actor to collect a bit more reward during the run.

In our experiments we also tried to run some training where the definition of done was unchanged and the orientation was not a part of the condition for ending the episode. For the vehicle was then only the reward function a source of information saying something about its orientation correctness. During the run it could have accumulated a bigger reward if also proper orientation was met. This approach however did not turn out to be so efficient. The success rate function of the learning was growing much slower than for training with definition of done involving orientation. Also, it restricted the vehicle a bit as the episode ended just after reaching waypoint position and the actor could not adjust the orientation by some small movements.

What also has to be changed for this increased difficulty is the reward function. As for the previous setup we tried some of them and compared their influence on the function of learning. The first reward function we tried is the simple reward function returning a big reward when the waypoint is successfully finished, otherwise it returns 0. For this reward function we had to create demonstrations so the learning algorithm has some good data to start with, otherwise the learning would not be efficient as it would take a really long time until the exploration drives the vehicle to the finish state returning some reward.

4.2.3.1. Stepped rewarder

The second approach we tried is the classic reward function giving some reward for every transition. This reward function is based on the distance as the reward function tested in experiment in the Chapter 4.1. However as we now operate with the final orientation, this reward function has to also take the orientation into account, so the learner is able to identify that lowering the difference between own orientation and the result orientation is a way to go for increasing reward income. Based on that we proposed a Stepped reward function which is based on evaluating individual states. The transition is than rewarded as a difference between the value of previous state and the following state so moving from less valued states to more valued states is a way how to gain positive reward income.

The evaluation of the states from which we return the difference is composed of two parts – the distance score part and the orientation score parts. Both parts have the same value and create a half of the resulting state value. The distance score of the state is the normalized and inverted distance of the state position to the waypoint position. The maximum distance score is received for the state with 0 distance to the waypoint. The orientation score is the normalized angle difference between the orientation of the vehicle in the state and target orientation of the waypoint.

Having such values for the state is obvious that the most optimal state – exact position and orientation of the waypoint, has the maximum value, which is the upper bound of both scores (distance and orientation score). However this value is not the maximal obtainable value for the actor. Because the vehicle starts in various states which can be closer or further from the waypoint and also rotation is randomized. Because the reward is the difference between values of two following states, the maximum obtainable reward in the episode corresponds to the difference between the most optimal state value and the episode’s initial state value. Thus this property does not allow us to compare networks in a term of gained reward and we must rather compare the success rates or the distances to the waypoint position in a final state of the episodes.

4.2.3.2. Updated stepped rewarder

The rewarder giving a reward as a difference between steps showed up to work well during the learning. For that reason, we decided to create a bit more complex variation which should enforce vehicles to park more precisely. The idea of the updated stepped rewarder is, that in basic stepped rewarder all states are evaluated the same way so the actor is not motivated to improve its precision in the end more than at the beginning. We tried to change that, because in our opinion the final phase precision is more important than precision at the beginning. For example at the beginning we do not insist on having vehicles and waypoint’s orientation precisely aligned, however at the end we want them to have minimal difference.

For the implementation we decided to add a third score to the calculation of the reward function. This third score has non-zero value only in a radius of two metres around the waypoint and this value is added to basic state value estimation as it was done for the basic variant. This third score is also composed by the distance and rotation scores of the states.

4.2.4. Network architecture

For the problem we used actor networks having 2 hidden layers of 40 neurons. The same layer width was used for critic networks. The networks schemes can than be described as following:

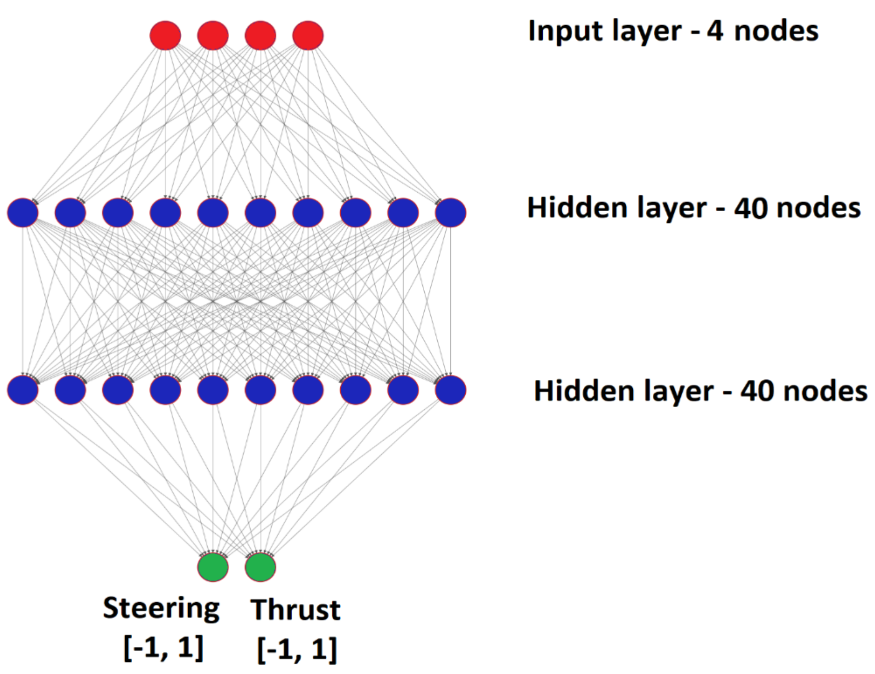

Figure 4.7: Scheme of the actor network for Parking with orientation problem

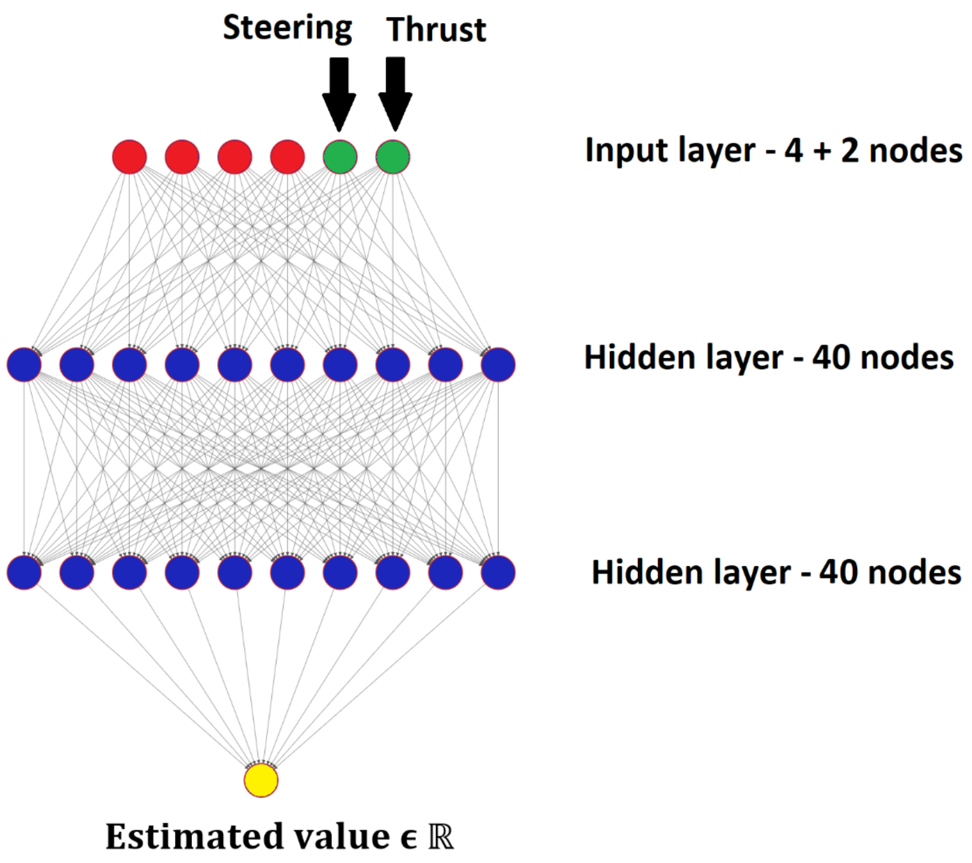

Figure 4.8: Scheme of the critic network for Parking with orientation problem

For the networks we used values of parameters that are listed in the Table 4.2. The parameters have the same naming as in pseudocode in the Chapter 2.9.

Parameter name	Parameter value
Replay buffer size (D)	1 000 000
Discount factor (γ)	0.99
Polyak (ρ)	0.001
Actor learning rate	0.001
Critic learning rate	0.001
Batch size (B)	200
Noise sigma (σ)	0.4
Noice clip (c)	0.2
Policy delay	2

Table 4.2: Parameters of the network for the parking with orientation problem

4.2.5. Results

The learning of networks can be described by values obtained during periodical evaluation. The values we use for quality measurements are the success rate of the network and the average distance to target position when the episode ends.

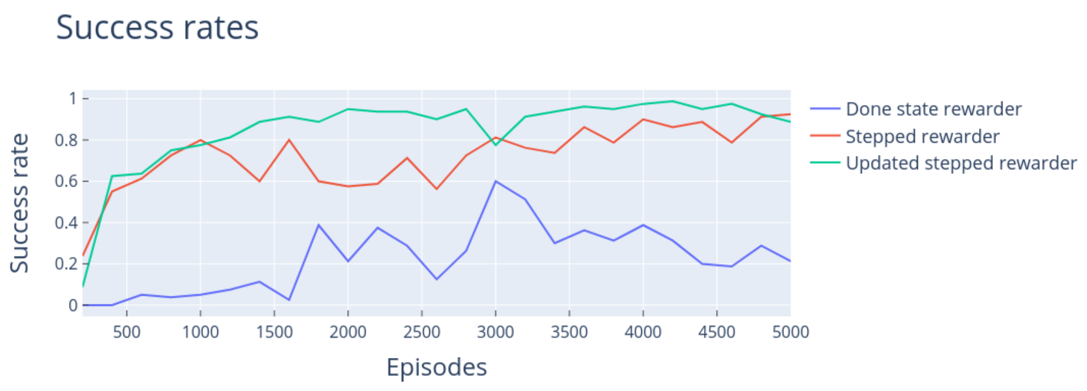

Figure 4.9: Success rates of the network by the time of learning

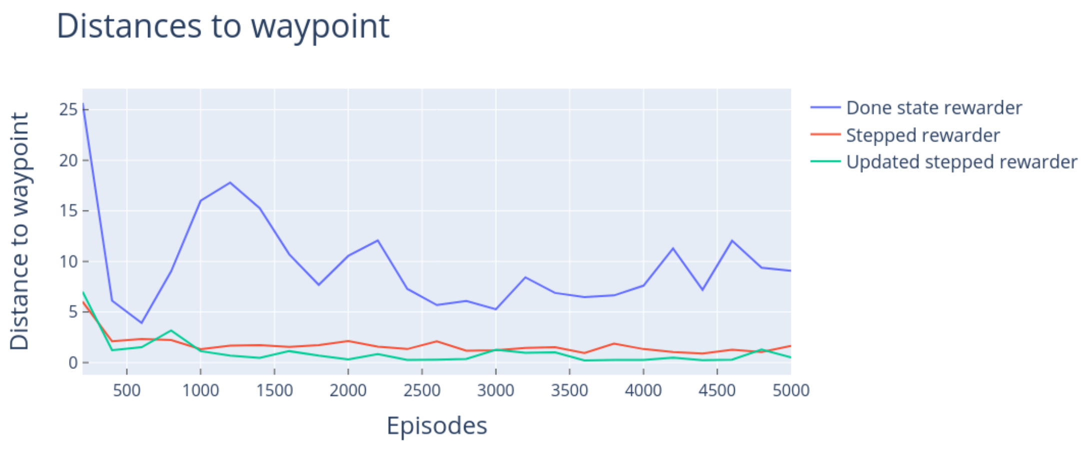

Figure 4.10: Average distances to the goal position after episode ends.

In the Figures 4.9 and 4.10 we can see resulting values of the evaluation for three rewarders described in 4.2.3. The first observation is that the Done state rewarder is not much suitable for the problem. Apparently, its outcomes are unstable and even though the success rate might tend to rise a bit it does not seem to be comparable with other two kinds of rewarder. For that reason, we decided to not continue with experiments with this kind of rewarder for the more difficult problem instances.

The two stepped rewarder seems to be working for the problem instance. Based on the feedback the vehicle is able to learn how to drive to the waypoint and stop there with proper orientation. We can see that at the beginning of both stepped rewarders functions have almost the same start. Later the updated stepped rewarder seems to be a bit more efficient. We think that as the average final distance is lower and even the success rate is bigger the actor's precision in the close area around the waypoint was supported and it leads to better reward gains.

Based on these results we decided to implement an updated stepped rewarder for the final use case.

4.3. Parking between obstacles

The last difficulty level and our final use case was parking to the position with desired orientation while the position is surrounded by the obstacles. Vehicle has a goal to park at the waypoint with the given orientation, but the new condition is that it must not collide with any obstacle. In the real world it can be compared to a situation where a car needs to park in a parking slot surrounded by other cars. The other use case from the military environment could be the case where we have a bridge builder vehicle that is supposed to build the bridge. Typical procedure for this case starts with trucks which deploy bridge parts at specific positions and then the building crane is supposed to park between these parts with specific orientation in which he is supposed to build the bridge.

In our simulation we created this environment using some generic objects that are creating the obstacles around the waypoint. The screenshot of the environment can be seen in the Figure 4.11.

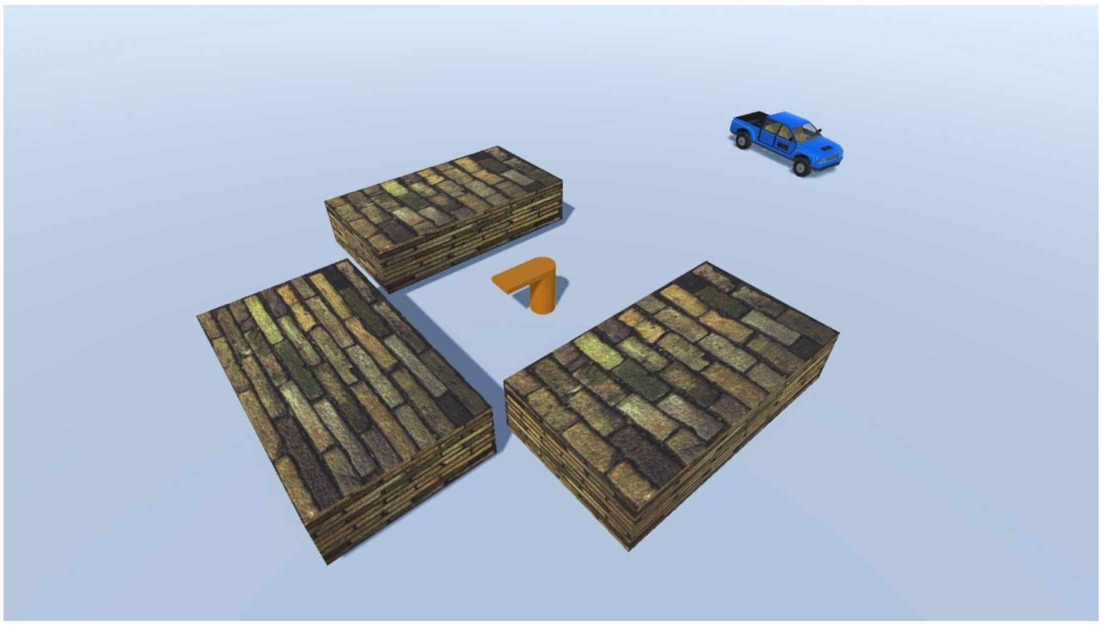
Figure 4.11: Environment instance of the parking between obstacles problem

The environment is not constant. After each reset of the environment we randomly hide some obstacles, so it can happen that all 3 obstacles are active in the episode, but it can also happen that none of the obstacles will be present.

With introducing obstacles, we also have to adjust the spawning of the vehicle. Because our system does not involve any path planning, we must make sure that the waypoint can be seen from the spawn position of the vehicle. Otherwise if the vehicle is spawned behind some obstacle, the network will not be able to determine how to go around the obstacle so it can reach the waypoint. This issue limits our environment, however in a complete system we could assume that there is some kind of path planning with path following AI. The path following AI can then follow the path until the waypoint is seen and then pass control to the parking network.

Because of this limitation we adjusted our spawning of the vehicle. We can divide the area around the waypoint into four quadrants and each quadrant can be selected for the vehicle spawning only in case that the obstacle in the quadrant is not active. The instance of an environment with marked quadrants can be seen in the Figure 4.12.

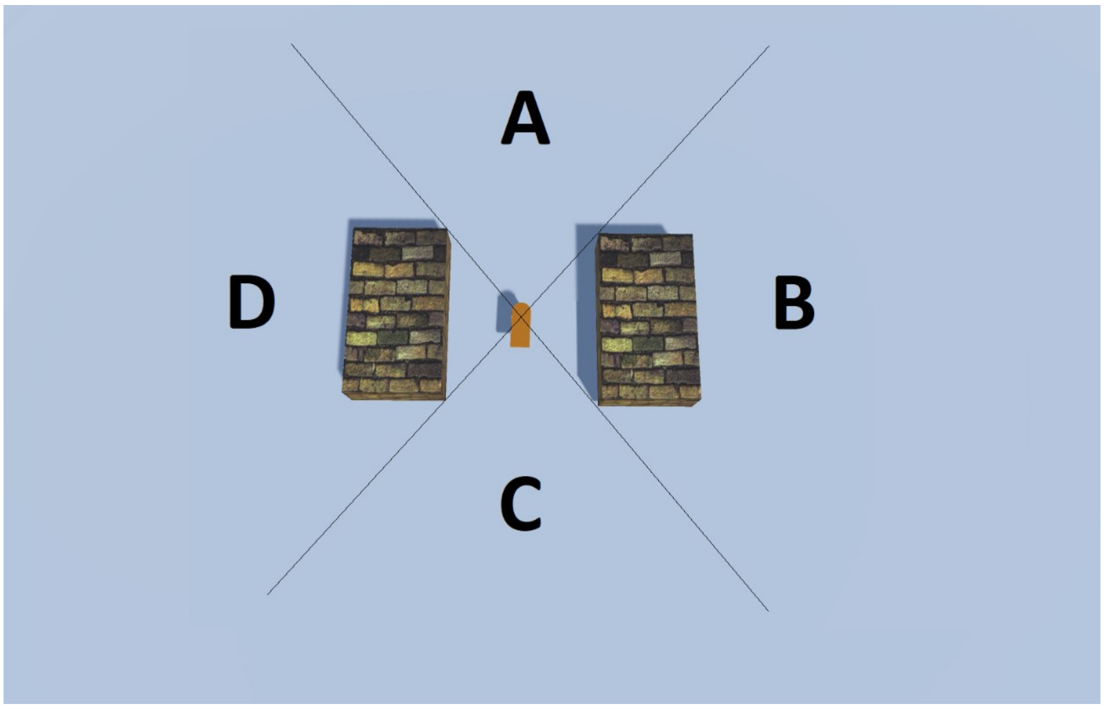

Figure 4.12: Example of the environment with obstacles and marked quadrants for spawning

For the situation in the figure is true, that we can spawn the vehicle either in the quadrant A or quadrant C. The quadrants B and D cannot be selected for vehicle spawn as they are hidden behind active obstacles.

4.3.1. Network inputs

For this problem we utilized inputs defined in previous difficulties as these inputs turned out to be sufficient for the solution. Of course we had to expand the input so the networks could get information about the obstacles positions. The first three inputs were adopted from the network described in the Chapter 4.1 while the fourth one is the one input added in the network from the Chapter 4.2.

The first input value defines the angle difference between the vehicle's absolute orientation and the direction from the vehicle position to the waypoint. The value is taken from the range of [-1.0, 1.0] where value of 0 means the angle difference is 0, while the values approaching borders of the interval represent angles close to -180° respectively 180°.

The second input passes information about the distance between the vehicle position and the waypoint position. The distance is normalized to range of [0, 1.0] where value of 0 means that the current position of the vehicle and the position of the waypoint are identical, while the value of 1 says that the distance between vehicle’s position and waypoint’s position is 30 metres.

The third input is a speed value. The value is also in the range of [-1.0, 1.0]. The value of 0 represents speed of 0 km/h and the maximum and minimum values of the interval represent speed of 50 km/h. The sign of the value distinguishes between speed of driving in forward direction and speed of driving while reversing.

The fourth input is the angle difference between the vehicle's absolute orientation and the wayponit’s orientation, which is the heading we want the vehicle to have after the parking maneuver is done. This value is normalized to the interval of [-1.0, 1.0] similar to the first input.

These inputs are sufficient to give to the actor information about the vehicle's position relative to the waypoint and also information about the desired vehicle state. With added obstacles we also need to pass some kind of information about the shape of obstacles and their position so the vehicle can avoid them. For that purpose we decided to use raycasts.

The raycasts have origin in the vehicle's center point and the cast distance we set is 15 metres. The main purpose of the cast is to detect obstacles for the network so the network could avoid collision based on the raycast result. Thus, the cast distance has to be set to the distance in which the vehicle is capable of stopping. That means that the cast distance could be different for different vehicle types, as some vehicles can decelerate quickly, but e.g. the vehicles with large mass could have bigger braking distance.

The vehicle can collide with obstacles in many ways – basically it can collide in all directions, not only during the head-on crash. For that reason, we have to provide raycasts information from any direction. In our simulation we created a uniform raycast system, that is sending raycasts circularly and evenly to all directions. The raycast count is 36 and each raycast is rotated by 10° from the previous one. The example of the environment with visualized raycasts can be seen in the Figure 4.13.

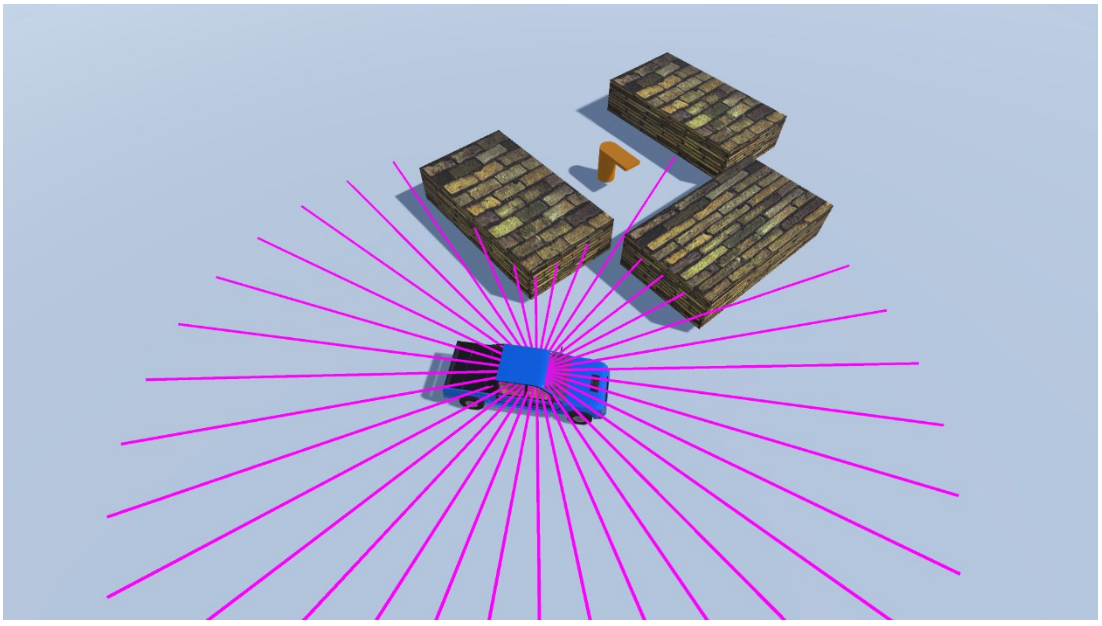

Figure 4.13: Environment instance with visualized raycasts

Raycasts results are thus other inputs of the network. Each raycast results in a number in the range of [0, 1.0] where the value of 0 means that the raycast did not hit any obstacle in its whole cast distance. The other value means that there is an obstacle in the range of the raycast and the absolute value gives information how far it is from the center of the vehicle. The value of 1.0 means that the distance between vehicle center and the obstacle is 0 (however this is not supposed to happen as the vehicle center is surrounded by the vehicle collider).

The raycasts result are the following 36 inputs of the network, so the resulting format of the input is 40 numbers.

4.3.2. Network outputs

As for previous difficulties of the environment the output is two numbers. Actions that the actor can perform are still the same. The first output is the steering of the wheels. It is a number in range of [-1.0, 1.0] where values of -1.0 and 1.0 represent the maximum steering the vehicle can take. The sign specifies whether the steering is clockwise or counterclockwise. The value of 0 means that vehicle wheels are pointer directly forward.

The second output is the desired thrust of the engine. It is also number within the range of [-1.0, 1.0]. The value of -1.0 and 1.0 represents the maximum possible engine thrust while the sign specifies whether the engine should create movement forward or backward.

4.3.3. Rewarders and done states

With the possibility of collision we have to add a new condition for ending the episode. Conditions for ending the episodes, that we used in instances of lower difficulty remains the same. The episode is ended, if the vehicle reaches the waypoint position with the desired orientation. This state is the desired one and reaching that is the criteria for considering episode as successful. We also still want to end the episode if the vehicle drive further than 30 metres away from waypoint because we don't want to explore states that are further and spend training time on them as they are not perspective for the problem solution. We also still gives a step limits for the episode. For this situation we decided to enlarge the limit to 500 steps per episode. If the vehicles reaches the limit, the episode is terminated.

Because there are now obstacles in the environment, we need to also add a new condition for ending the episode. The episode is now ended for the vehicle if the vehicle collides with the obstacle.

For the rewarder we decided to use the Updated stepped rewarder described in the Chapter 4.2.3.2. The reward gives reward based on the difference between values of two states in a transition. The value of state is composed by adding scores for distance of a state from the target position and the difference between vehicle's orientation in the state and the desired orientation of the waypoint. To increase the precision of parking there is a third score that increases the values of states within the range of 2 metres around the waypoint to make the difference between those states more relevant for the actor to enforce improving precision.

The updated stepped reward rewarder however does not penalize vehicles for reaching unwanted states. So far we did not have much of a clearly unwanted state. We could say that the ending state where vehicles drive further than 30 metres away from the waypoint is an unwanted state, but there wasn't any reason to penalize it, because when the vehicle is so far away from the waypoint, it already has to have accumulated a poor reward. However, for the case with the obstacles there is a difference. If the vehicle collides with the obstacle while being near the waypoint, the accumulated reward is already high, so there is not much of a pressure for the actor to avoid collisions while being close to the waypoint. Thus, we tried to handle this by creating a new version of rewarder, that is based on the Updated stepped rewarder (the Chapter 4.2.3.2). For the whole episode the returned rewards are the same, however for the done states, there are extra rewards to either penalize or to praise the actor. Practically if the vehicle collides, it gets penalized by high negative reward no matter how close to the waypoint the vehicle is and if the vehicle completes the waypoint correctly, it gets extra positive reward for a good job.

4.3.4. Network architecture

With the increased number of inputs we had to enlarge the network into shape that would have enough hidden neurons to process all the information. Thus, the actor and critic networks for this level of difficulty have 250 neurons in 2 hidden layers.

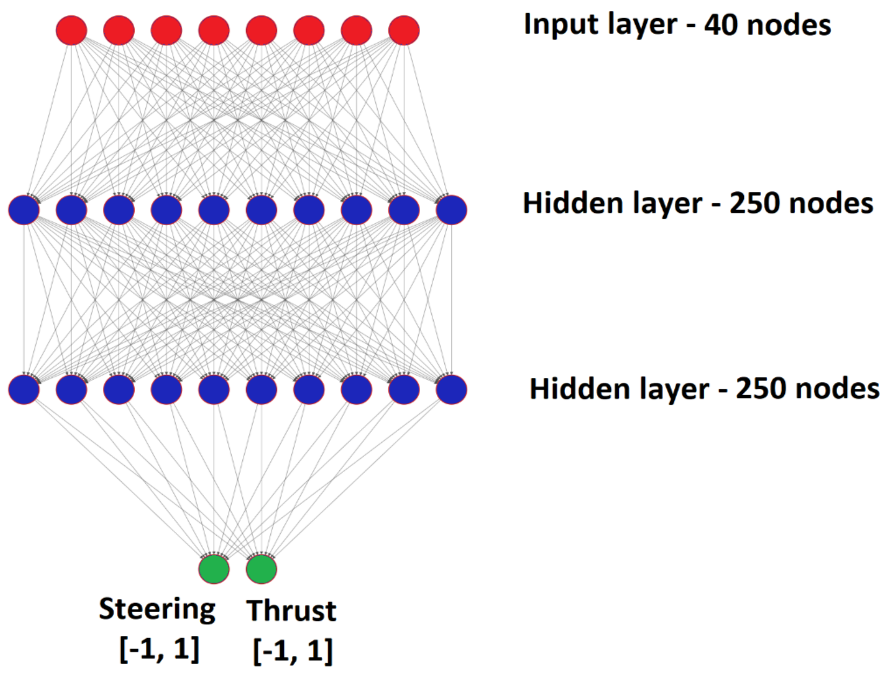

Figure 4.14: Scheme of the actor network for the Parking between obstacles problem

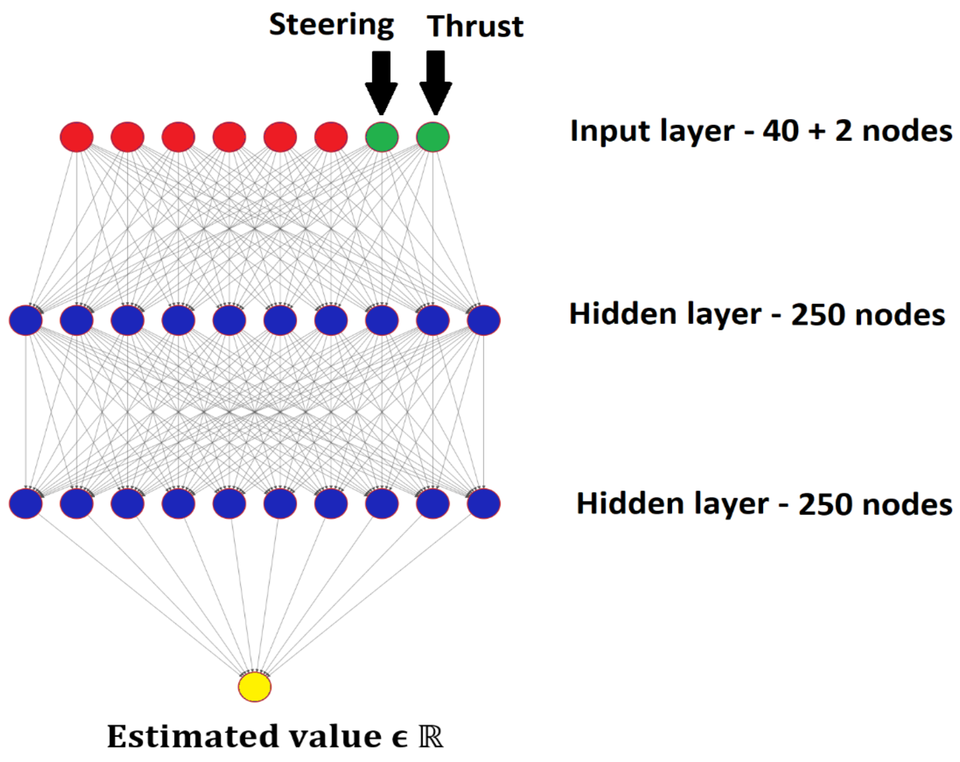

Figure 4.15: Scheme of the critic network for the Parking between obstacles problem

For the networks we used values of parameters that are listed in the Table 4.3. The parameters have the same naming as in pseudocode in chapter 2.9.

Parameter name	Parameter value
Replay buffer size (D)	1 000 000
Discount factor (γ)	0.99
Polyak (ρ)	0.001
Actor learning rate	0.001
Critic learning rate	0.001
Batch size (B)	400
Noise sigma (σ)	0.4
Noice clip (c)	0.2
Policy delay	2

Table 4.3: Parameters of the networks for the Parking to position problem

4.3.5. Results

We describe the learning process of the network using the same values as for easier problems in the Chapters 4.1 and 4.2. The values we are tracking during periodical evaluation are the success rate of the actor network and the average distance of the vehicle from the waypoint when the episode is ended.

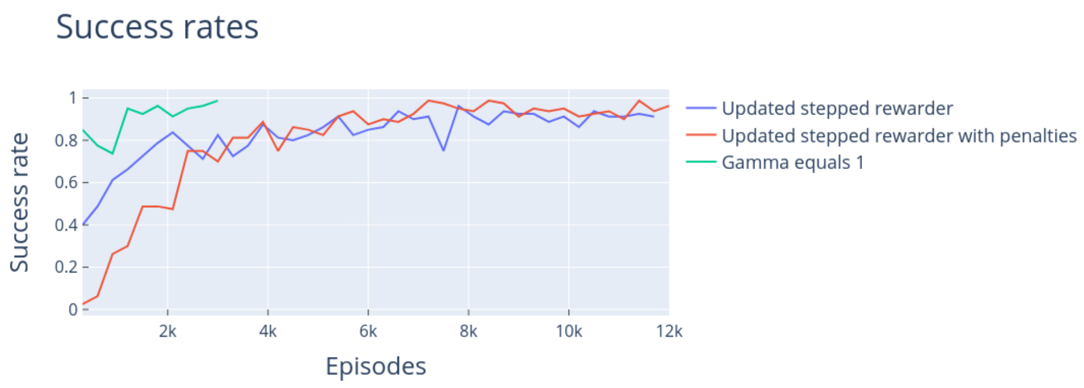

Figure 4.16: Success rate of the network for the Parking between obstacles problem

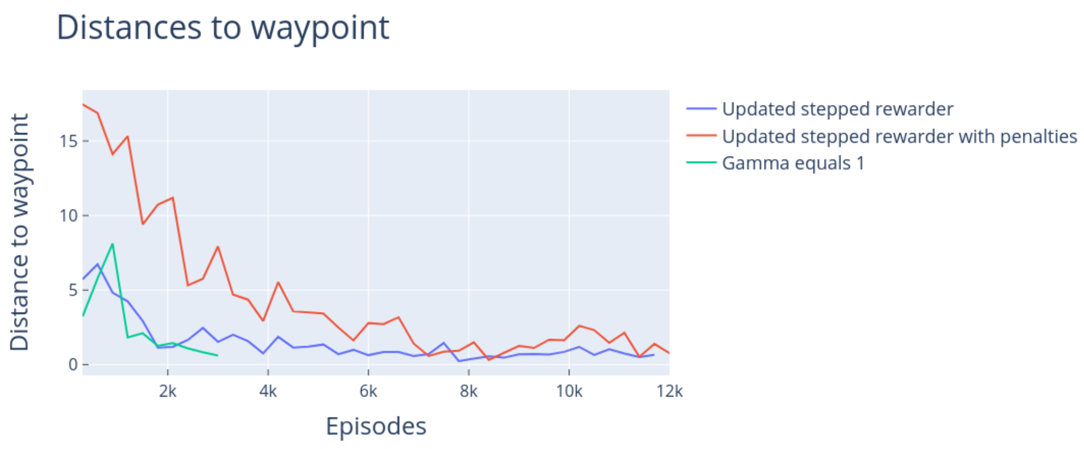

Figure 4.17: Average distances to the target positions after episode endings for Parking between obstacles problem

In the Figures 4.16 and 4.17 we can see progress of networks learning for the Parking between obstacles problem. For the problem we tried to use two instances of rewarding function, the Updates stepped rewarder which we introduced during solution of problem in the Chapter 4.2, and the Updates stepped rewarder with penalties which we introduced especially for this problem to see whether it can be an improvement.

From the resulting graph we can see that at the beginning of the learning process the version of rewarder with penalties is a bit behind the Updated stepped rewarder. The success rate is lower as well as the average distance to the goal. We think that this might be caused especially by the penalties. If the actor is not trained, the collisions are pretty common. However, the version of the rewarder with penalties gives for such collisions high negative reward which might cause the actor to be scared of the obstacles. That might result in lower determination to explore areas near the waypoint as there is a big risk of getting negative reward and it takes longer to find a safe path to the goal without collision.

The last function in the graph named as “Gamma equals 1” is the Updated stepped rewarder, however during the time of training we were using discount factor (Gamma) equals to 1. Even though we ended this training after 3000 episodes, it seems promising. The average distance to target is almost similar as for Updated stepped rewarder function and the success rate looks even better. Interesting thing is that when we looked at how the network drives, we found out that the network is not rushing to the waypoint and takes its time. Practically it looked like that the network found out a good way to the waypoint from the specific position and in each episode the goal of the network was to get to this specific position from which it knew how to reach the waypoint.

This approach is effective in terms of success rate. The network basically divided one complicated problem into two easier. However in terms of effectiveness this solution is not so good. There are instances where the vehicle has to drive around obstacles to the specific position even though it could just directly drive to the waypoint. Moreover, the actor does not have any motivation to make runs more effective, because the critic does not see a difference between reward collected in 150 steps and basically the same reward collected in 500 steps as the discount factor of 1 does not decrease the expected values of states in a distant future.

5. Final solution evaluation

The problem defined in the Chapter 4.3 is a final level of difficulty we decided to solve. The resulting networks showed a high success rate after being trained on 12 000 episodes. However, the evaluations we did during training are only for orientation and do not fully indicate the true capabilities of the solution. Thus, for the purpose of evaluation we decided to create a global uniform test procedure which does not use randomly generated episodes, but some equally distributed environment instances.

The idea of the test is that we run our network on a bunch of episodes which uniformly covers various starting positions of the vehicle in a maneuver space, with various rotations and with various obstacles appearance.

The episodes are generated in a way that we uniformly sample the space, where the vehicle can be spawned with the granularity of 1 metres. For each position we added 4 possible starting orientations. The initial orientation is that vehicle heading away from the waypoint and the other 3 we create by rotating the vehicle for 90°, 180° and 270°.

For each pair of the position and the initial vehicle orientation we can use different patterns of activated obstacles (as defined in problem description of 4.3). For our evaluations we used all possible combinations of activated obstacles so the environment assumption (the waypoint can be always seen from the episode initial position) is met.

Defining all these initial position parameters, we resulted with 12 800 different environment instances that we evaluated. After running the evaluation, we obtained following result which gives us the real scoring of the network efficiency:

- Average distance to the waypoint: 0.9533344650002619 metres
- Success rate: 0.957109375
- Collision rate: 0.003671875

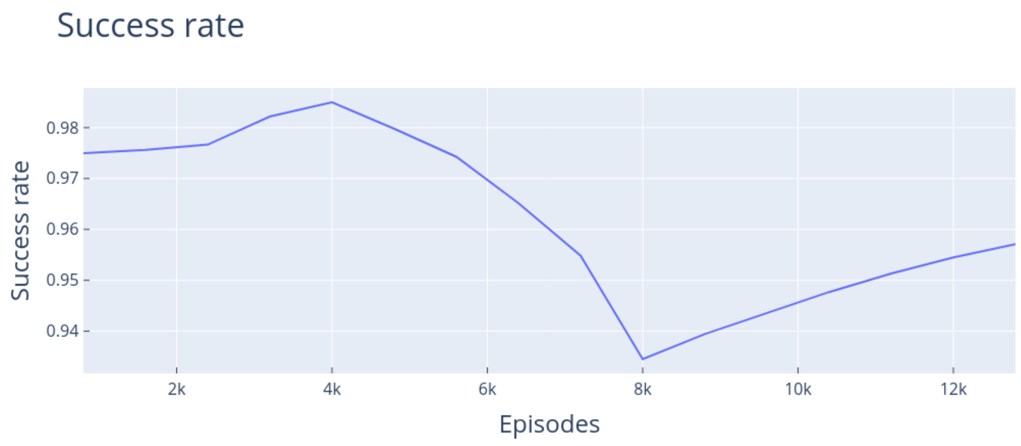

Figure 5.1: Success rate progress during the final evaluation

From the result data we can see that the trained network reaches 95.6% success rate with only 0.36% of collisions. However, the interesting part is the graph in the Figure 5.1. The graph is showing changes of the success rate at the time of evaluation. We can see that there was a major drop of the success rate in the period between the 4K episodes and the 8K episodes.

Based on our observation the problematic cases were concentrated in the area marked in the Figure 5.2 by a red circle, especially with vehicles spawned with orientation directing away from the waypoint.

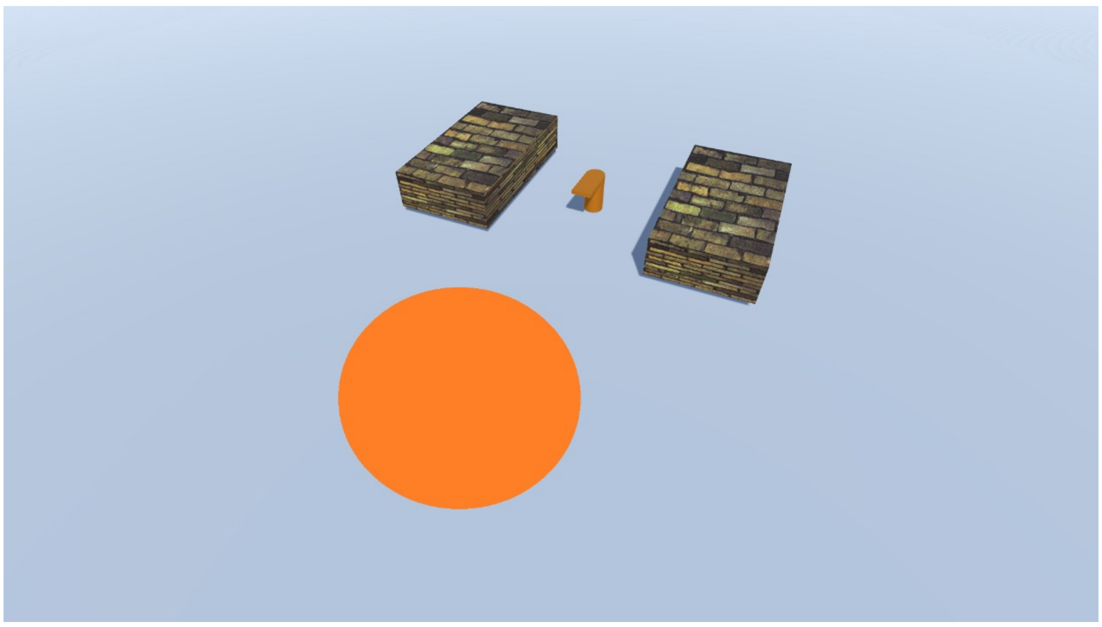

Figure 5.2: Environment screenshot with marked problematic area for the resulting network

This problematic area could be the result of a low amount of exploring being done in this area. To improve the exploration or to force the vehicle to get more focused on the problematic area, the spawning of a vehicle could be updated so the vehicle is more often spawned within the unexplored region. Another approach that could be worth trying is to find exact starting positions where the vehicle fails to drive to the goal and create demonstration episodes by manual solving of these situations. Those demonstration episodes then can be used in a demonstration replay as a part of a training data, giving to the network an idea of how the goal could be reached.

Conclusion and future work

In the thesis we created a simulated environment with a vehicle and a neural network that can solve the problem of parking between static obstacles with a success rate of 95.7 %. From the analysis of the evaluation results we identified an area which is problematic for our network to resolve. Based on this observation we proposed a method to improve the success rates and to cover these problematic instances.

Even though we created a solution for this level of difficulty, there can still be many more complex situations that can be explored. In the Chapter 4.3 we mentioned that the network cannot work with path planning. One of the future work can therefore introduce a network which is designed for following the path. In connection with our parking network it can create a more complete solution for a whole vehicle driving problem.

Also a parking problem can be expanded with adding some new features into the environment. One of the net levels of difficulty can for example include smaller obstacles like stones or trees spawned randomly in a whole maneuvering area. Also we can define a situation, where more than one vehicle is parking next to each other in the same maneuver area. These instances of a problem are defining a more complex environment and can be a point of future research.

Bibliography

Sutton, R. S., Barto, A.G. (2018). Reinforcement learning: an introduction, MIT Press. ISBN 9780262039246.

Hou, Y., Liu, L., Wei, Q., Xu, X., Chen, C. (2017). "A novel DDPG method with prioritized experience replay," *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2017, pp. 316-321, doi: 10.1109/SMC.2017.8122622.

MS Vehicle System free version (2018). Unity free asset. [Accessed: 2020-04-07]. Obtained from: https://assetstore.unity.com/packages/tools/physics/ms-vehicle-system-free-version-90214#description.

Juliani, A., Berges, V., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., Lange, D. (2020). Unity: A General Platform for Intelligent Agents. arXiv preprint arXiv:1809.02627. https://github.com/Unity-Technologies/ml-agents.

Sallab, A. EL, Abdou, M., Perot, E., Yogamani, S., (2017). Deep Reinforcement Learning framework for Autonomous Driving. Electronic Imaging, Autonomous Vehicles and Machines 2017, pp. 70-76(7). DOI: https://doi.org/10.2352/ISSN.2470-1173.2017.19.AVM-023.

Wang, S., Jia, D., Weng, X. (2019). Deep Reinforcement Learning for Autonomous Driving. arXiv preprint arXiv:1811.11329v3.

Shalev-Shwartz, S., Shammah, S., Shashua, S. (2016). Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving. arXiv preprint arXiv:1610.03295v1.

TensorFlow. [Accessed: 2020-04-02]. Available at: https://www.tensorflow.org.

OpenAI: DDPG. [Accessed 2021-05-20]. Available at: https://spinningup.openai.com/en/latest/algorithms/ddpg.html.

OpenAI: TD3. [Accessed 2020-04-02]. Available at: https://spinningup.openai.com/en/latest/algorithms/td3.html.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D. (2019). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971v6**.**

Plotly.Io. [Accessed 2021-07-18]. Available at: https://chart-studio.plotly.com/feed/#/.

Yoon, Ch. (2019). [Accessed 2021-05-20]. Deep Deterministic Policy Gradients Explained. Available at: https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b.

Adam, S., Busoniu, L., Babuska, R. (2012)."Experience Replay for Real-Time Reinforcement Learning Control," in *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 2, pp. 201-212, March 2012, doi: 10.1109/TSMCC.2011.2106494.

Byrne, D. (2019). [Accessed 2020-04-03]. TD3: Learning To Run With AI. Available at: https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93.

Fujimoto, S., van Hoof, H., Meger, D. (2018). Addressing Function Approximation Error in Actor-Critic Methods. arXiv preprint arXiv:1802.09477v3.

Francois-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., Pineau, J. (2018). An Introduction to Deep Reinforcement Learning. arXiv preprint arXiv:1811.12560v2**.**

Weng, L. (2020). [Accessed 2021-03-12]. Exploration Strategies in Deep Reinforcement Learning. Journal lilianweng.github.io/lil-log. Available at: https://lilianweng.github.io/lil-log/2020/06/07/exploration-strategies-in-deep-reinforcement-learning.html.

blackburn (2019). [Accessed 2021-05-20]. Reinforcement Learning: Bellman Equation and Optimality (Part 2). Available at: https://towardsdatascience.com/reinforcement-learning-markov-decision-process-part-2-96837c936ec3.

NN-SVG. [Accessed 2021-07-18]. Available at: https://alexlenail.me/NN-SVG/index.html.

Junzuo, L., Qiang, L. (2021). An Automatic Parking Model Based on Deep Reinforcement Learning. *J. Phys.: Conf. Ser.* **1883** 012111.

Zhang, P., Xiong, L., Yu, Z., Fang, P., Yan, S., Yao, J., Zhou, Y. (2019). Reinforcement Learning-Based End-to-End Parking for Automatic Parking System. *Sensors* **2019**, *19*, 3996. https://doi.org/10.3390/s19183996

Zhang, J., Chen, H., Song, S., Hu, F. (2020). "Reinforcement Learning-Based Motion Planning for Automatic Parking System," in *IEEE Access*, vol. 8, pp. 154485-154501, 2020, doi: 10.1109/ACCESS.2020.3017770.

Song, S., Chen, H., Sun, H., Liu, M. (2020) Data Efficient Reinforcement Learning for Integrated Lateral Planning and Control in Automated Parking System. *Sensors* **2020**, *20*, 7297. https://doi.org/10.3390/s20247297

List of Figures

Figure 2.1: TD3 algorithm pseudocode (OpenAI: TD3, 2018). 13

Figure 3.1: Scheme of the proposed learning system 16

Figure 3.2: Screenshot of the environment instance 18

Figure 4.1: Environment instance of the parking to position problem 22

Figure 4.2: Scheme of the actor network for Parking on position problem 26

Figure 4.3: Scheme of the critic network for Parking on position problem 26

Figure 4.4: Success rates of the networks by the time of learning 27

Figure 4.5: Final distances of the vehicles by the time of learning 28

Figure 4.6: Environment instance of the parking with orientation problem 29

Figure 4.7: Scheme of the actor network for Parking with orientation problem 32

Figure 4.8: Scheme of the critic network for Parking with orientation problem 33

Figure 4.9: Success rates of the network by the time of learning 34

Figure 4.10: Average distances to the goal position after episode ends. 34

Figure 4.11: Environment instance of the parking between obstacles problem 35

Figure 4.12: Example of the environment with obstacles and marked quadrants for spawning 36

Figure 4.13: Environment instance with visualized raycasts 38

Figure 4.14: Scheme of the actor network for the Parking between obstacles problem 40

Figure 4.15: Scheme of the critic network for the Parking between obstacles problem 40

Figure 4.16: Success rate of the network for the Parking between obstacles problem 41

Figure 4.17: Average distances to the target positions after episode endings for Parking between obstacles problem 42

Figure 5.1: Success rate progress during the final evaluation 45

Figure 5.2: Environment screenshot with marked problematic area for the resulting network 45

List of Tables

Table 4.1: Parameters of the network for the parking to position problem 27

Table 4.2: Parameters of the network for the parking with orientation problem 33

Table 4.3: Parameters of the networks for the Parking to position problem 41