# Navigating in a Simulated Environment with Curriculum-based Reinforcement Learning

**Master's Thesis**

**Jevgenij Martinkevič**

Aalborg University Copenhagen

A. C. Meyers Vænge 15

2450 Copenhagen SV

AALBORG UNIVERSITY

STUDENT REPORT

**Medialogy**
Aalborg University
http://www.aau.dk

**Title:**

Navigating in a Simulated Environment with Curriculum-based Reinforcement Learning

**Theme:**

Master's Thesis

**Project Period:**

Spring 2018

**Participant(s):**

Jevgenij Martinkevič

**Supervisor(s):**

Hendrik Purwins

**Copies:**

1

**Page Numbers:**

46

**Date of Completion:**

August 31, 2018

**Abstract:**

It has been recently demonstrated how Reinforcement Learning method can been used to solve a range of different locomotion, navigation and robotics tasks as well as reach exceptional performance in a number of games of a varying complexity. However, it has also been proven to be an extremely complicated and time consuming task to convert the goal of the problem into a reward signal. In this thesis, a Curriculum-based Reinforcement Learning approach is investigated and applied to solve a navigation task using only sparse reward signal. The training process is performed in a simulated learning environment built within Unity Engine, while the agents are trained with Proximal Policy Optimization method implemented with Unity Machine Learning Agents Toolkit. The results show that the target task could not be solved by the typical reinforcement learning agent using only sparse reward signal within the given time. However, the agent trained with an environment-centered curriculum, where the task is deconstructed and introduced to the agent in lessons of increasing difficulty, managed to solve the target task and reach success rate of 99%. Furthermore, a combined application of curriculum learning and reward shaping is investigated. It is observed, that this can negatively affect the training process if the two approaches are overlapping by encouraging the same type of behavior.

# Content

# I. Introduction

## 1.1 Motivation

In the past few years, reinforcement learning has not only been successfully applied but also reached remarkable performance in different Atari 2600 games and the game of Go (Mnih et al., 2013, 2015; Silver et al., 2016). It was used to learn locomotion as well as solve different robotics tasks (Heess et al., 2017; Lillicrap et al., 2015; W. Yu, Turk, & Liu, 2018). Moreover, some applications were extended even further, by transferring the learned knowledge from simulations to the real-world environment. Sadeghi & Levine (2016) showed how an autonomous quadrotor robot, or a multirotor helicopter, can be trained from just simulated visual data using reinforcement learning approach. The task was based on collision avoidance and the learned control policy, which encapsulates the knowledge acquired from simulated training, was successfully deployed to a real-world robot. A similar approach was used to train a real-world robotic hand to solve a task, which involves changing the orientation of the object held (OpenAI, 2018). Recently, reinforcement learning has been used to train an agent to drive a car in a real-world environment (Kendall et al., 2018). Nevertheless, the initial training sessions were conducted in the simulated environment. After numerous experiments were conducted, part of the acquired knowledge was used during the implementation of a similar setup on a real-world car.

All of these examples are united by the fact that the training process, or at least part of it, was conducted in simulated environments. This approach has been widely adopted as it provides a number of benefits. Firstly, the training process can go through millions of episodes allowing proper environment exploration (Amodei et al., 2016). Secondly, the implementations can be tested in a safe environment (Sutton & Barto, 2018). Environment simulations are especially heavily used in the development of autonomous cars. NVIDIA has developed a photorealistic simulation platform specifically designed for testing autonomous cars, while Waymo, an independent company developing its own self-driving car technology, has managed to drive autonomously for over 2.7 billion miles just in simulation (NVIDIA, 2018; Waymo, 2018).

Even though simulated learning environments can be used for extensive testing and training, some reinforcement learning specific issues can still persist. One of the most crucial parts of reinforcement learning is defining the goal (Sutton & Barto, 2018). The goal of the task in question is interpreted through rewards, which are used to either encourage certain behavior or, in fact, to penalize it. Therefore, in most of the mentioned cases, a custom, goal specific, reward function is used to calculate such reward. In the case of the object manipulation with robotic hand, the current and the target object rotation angles define the received reward. Additionally,

the robot receives a reward of -20 as a penalty when it drops the object, while a reward of +5 is acquired for solving the task. On the other hand, in the autonomous quadrotor robot example, the received reward is calculated based on: radius of the quadrotor, distance to the nearest obstacle and defined distance threshold. The received reward is lower when the robot is flying closer to the obstacles. In both cases, the reward function is encouraging certain problem specific behavior. The issue of choosing appropriate rewards may occur when the solution to the reinforcement learning problem is not clearly defined. Nevertheless, a number of approaches have been proposed, which can address this issue.

## 1.2 Purpose

In this thesis, the application of the curriculum-based reinforcement learning approach is presented. One specific reinforcement learning challenge will be introduced and discussed in detail, and the researched solutions will be described briefly with examples. One of these solutions, the curriculum learning approach, and its variations will be presented. With this approach in mind, the goal of this thesis is to study whether a reinforcement learning agent can learn to navigate in an environment filled with obstacles using only sparse reward signals. Additional reward signals and their impact on the training process and the curriculum will be investigated. Both, training and evaluation of the learned policies, will be conducted in a simulated learning environment.

## 1.3 Scope

The learning environment will be created using Unity Engine with the aim to construct training scenarios in the context of navigation and collision avoidance in a simulated environment, which otherwise would be time consuming to reproduce in real world. The agents will be trained using a Proximal Policy Optimization method implemented with Unity Machine Learning Agents Toolkit (Juliani et al., 2018). The method will be presented briefly, however, it will not be discussed in detail or evaluated as that falls outside of scope of this thesis.

The target task will involve navigating a car on a flat track filled with obstacles. An aspect of randomization will be added to the training in the form of obstacle positioning. To test the application of the curriculum learning approach, the target task will be used as the final lesson of the curriculum. Additional lessons, which can be viewed as simplified versions of the target task, will be constructed and presented to the agent during training as a form of curriculum. Moreover, additional reward signals and their combinations will be tested.

The training will be conducted using a single machine with a mobile variant of the 4-core Intel Core i7 6700HQ processor due to the fact that the reinforcement learning algorithm implementation, that is part of Unity Machine Learning Agents Toolkit, is not optimized for

graphics processing units (Intel, 2018; Juliani et al., 2018). Due to the limitation of the available processing power, the car control scheme will be limited to two specific actions: acceleration and steering. These actions will be represented using a discrete action space, rather than a continuous one. The agent will be trained using a continuous observation space, however, vision-based data will not be used.

# II. Background

In this chapter, the concept of Reinforcement Learning method and one of its main challenges are introduced. Furthermore, few researched solutions are presented, followed by detailed introduction to Curriculum Learning approach. Finally, multiple Curriculum Learning variations and related work with this approach applied are studied.

## 2.1 Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning, where a certain agent learns to solve a specific problem by interacting with the surrounding environment (Sutton & Barto, 2018). Through this interaction the agent collects observations, which are used to determine the subsequent interaction, or more specifically, an action. The collected observations define current state of the learning environment, which the agent can change using a defined set of actions. The agent in this case is a certain entity that, for example, can be an autonomous car. The available actions can be acceleration, steering, braking, etc., while the state observation can contain information about the car's location as well as the destination.

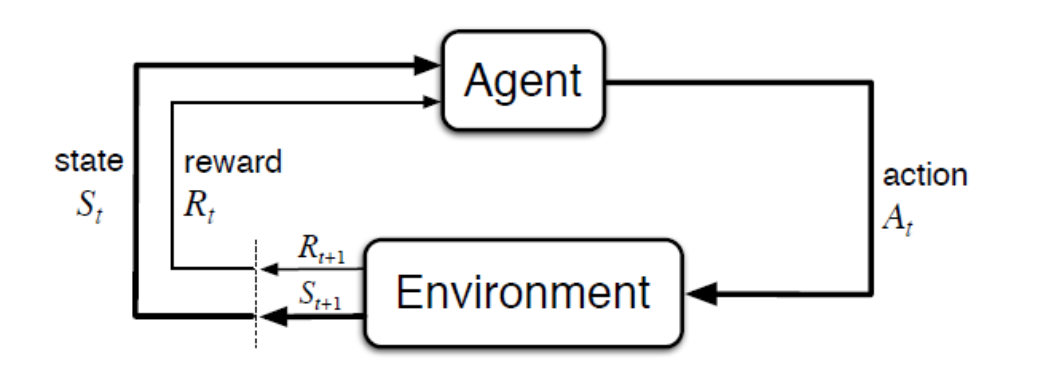



***Figure 2.1**: The agent-environment interaction in a Markov decision process (Sutton & Barto, 2018).*

Sutton & Barto (2018) state that generally reinforcement learning problems are formalized as Markov decision processes (MDPs) that can mathematically define a model of sequential decision making. The idea behind the MDP framework is that a goal-oriented problem, based on the process of learning from interaction, can be minimized to three specific signals that have to be passed from the learning environment to the agent and vice versa. These signals, as presented on Figure 2.1, are:

- The agent's actions.
- The representation of the states, based on which the appropriate actions are chosen.
- The reward signal.

A distinctive part of reinforcement learning is a reward signal, which is the only feedback the agent receives from the learning environment, and it is used to interpret the goal of the RL problem. This signal can be both, negative and positive, and the agent receives it during each step of the learning environment. As the agent's goal is to receive the highest possible cumulative reward, it has to learn to avoid receiving low and, especially, negative reward signals. At the very beginning of the training process, the agent has no information on what actions it has to take, thus, has no notion of how to receive a reward. Therefore, the agent has to take random actions at first to learn in what state what action results with a reward. This approach is called trial-and-error search. Moreover, in some cases, actions might have a more long lasting impact on the environment, thus, affecting the subsequent states and potential future rewards. This effect is known as *a delayed reward*. All these characteristics differentiate reinforcement learning from other types of machine learning approaches.

At the beginning of the training process, as shown on Figure 2.1, the observation data, which represents the initial state of the learning environment, is collected and sent to the agent. Afterwards, the agent chooses an appropriate action based on its knowledge and the received state information. That action is sent back to the learning environment, where a certain entity, controlled by the agent, executes it. Finally, the learning environment collects observation data of the new state and sends it to the agent along with the reward that is calculated based on the previous state-action pair. The agent evaluates the received reward and updates its knowledge. In the next step, the agent choses a new action based on the new state observation and sends it back to the learning environment. This process continues until the point when a terminal state is reached. At that point a single episode is considered to be over.

Sutton & Barto (2018) explain that the reinforcement learning system, apart from the agent and the learning environment, can also contain the following components:

- *A policy* - determines the agent's behavior. The policy can be as simple as a lookup table, where states are mapped to specific actions.
- *A value function* - defines the value of the current state, which is based on the expected reward that the agent can accumulate not only from the current state, but also from the future states. Sutton & Barto (2018) explain that "*Whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the long-term desirability of states after taking into account the states that are likely to follow, and the rewards available in those states.*" (p. 6). Furthermore, the state values are only estimations, as they are not provided by the learning environment, like reward signals.
- *A model* - defines the behavior of the learning environment. The models are used for planning, which can determine the subsequent states and their rewards based on the current state-action pair.

As presented on the Figure 2.2, the RL agents can fall into different categories depending on which of the components are used.

- *Value-based* - only contains a value function.
- *Policy-based* - only contains a policy.
- *Actor-Critic* - contains both, a policy and a value function.
- *Model-free* - contains a policy and/or a value function.
- *Model-based* - contains a policy and/or a value function as well as a model.

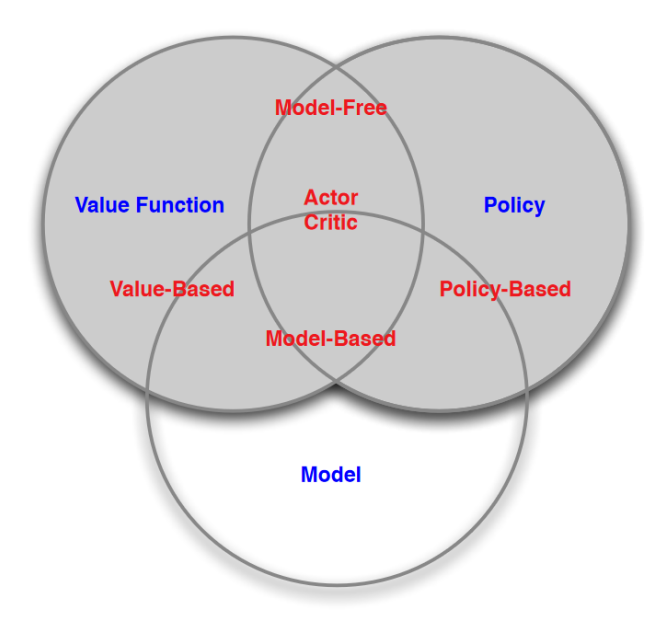



**Figure 2.2**: Reinforcement learning agent taxonomy (Silver, 2015).

## 2.2 Reinforcement Learning challenges

### 2.2.1 Reward signals

Examples, previously mentioned in Section 1.1, show that reinforcement learning can be adopted to solve a variety of problems. However, successful application of RL highly depends on the problem itself as well as how one approaches to solve it. Typically, a reinforcement learning agent adjusts its behavior based on the reward signals it receives from the learning environment. This reward is the only learning signal it collects and adjusting it has proven to be a challenge of its own (Arulkumaran, Deisenroth, Brundage, & Bharath, 2017). Solving a complex problem means that the reward function has to undergo a tuning process that would reflect the desired agent behavior. Otherwise, if the agent only receives sparse rewards, even a simple problem can become a complicated one (Sutton & Barto, 2018).

Effectiveness of the RL as well as overall results heavily depend on reward signals, which directly affect agents' behavior evolution during training. Therefore, Sutton & Barto (2018) also emphasize the fact that the design of the reward signal is crucial part of reinforcement learning. Moreover, they also state that it is rather easy to run into two different challenges during this process. First of all, it can be a complicated task to convert the goal of the problem into a reward signal. Especially, if the problem requires the agent to complete multiple tasks or to do specific sequence of actions for it to be considered as solved. Afterwards, it is likely for an agent to come across a strategy that can yield a positive reward based on the designed reward function, which

can be viewed as unwanted or even harmful to the environment, other agents or itself. Amodei et al. (2016) describe this behavior as *negative side effects*, which can occur when the agent is trained based on only one specific reward signal. The given example is of a household assistant robot, which task is to move a box from one point in the environment to another. If the agent receives rewards only based on the transportation task, then avoiding collision with other objects is not something it has to take into consideration. In this case, the learning environment has a sparse reward signal, which means that the agent will get a positive reward only when a specific state is reached, for example, when the box is delivered to a specific location. Moreover, Amodei et al. (2016) describe another type of unwanted behavior called *reward hacking*. Similarly to the previous example, a cleaning robot, that receives reward signal when it collects trash, could potentially learn how to exploit the environment to generate more trash for collection. On the other hand, if the reward is received when the environment is clean, the robot could learn to avoid areas where trash is most likely to be found.

Training an agent with sparse reward signals can be a truly challenging task. Depending on the learning environment, the chances of finding a particular state that yields a positive reward for an agent can be close to zero. For example, the learning environment might have several starting states, such scenario also adds randomness into the training process. Even if an agent manages to receive a positive reward at some point during the training, such occurrence can be too rare, or sparse, which means it might not have a significant enough impact on the training process. Sutton & Barto (2018) explain that in practice it is common to do a trial-and-error search for an appropriate reward signal that would result in an expected agent behavior. It is a job for a designer to figure out and make a decision on what signals work and which do not. This can be a tedious task, especially if the training process takes a lot of time. Therefore, specifically in such cases, more complex reward signals have to be introduced to the learning environment to facilitate the training process.

### 2.2.2 Reward shaping

A commonly used and a rather powerful technique for solving sparse reward problems is called *reward shaping* (Gullapalli & Barto, 1992; Sutton & Barto, 2018; Y. Ng, Harada, & Russell, 1999). The main idea behind reward shaping is to guide the agent towards states with high reward signal or the main goal of the reinforcement learning problem. It involves implementation of additional reward signals to the learning environment, which the agent receives for completing sub-goals. These rewards can be both negative and/or positive and have varying magnitude depending on the agent's progress. Gullapalli & Barto (1992) presented shaping approach where a batch of reward functions were used to solve a key-pressing task with RL. Each reward function corresponds to a specific behavior approximation. The agent would receive reward signal of 1 whenever it reaches a state where certain behavior approximation is considered to be

fulfilled. Moreover, the agent has to execute the first approximated behavior in order to receive subsequent reward signals. That is, to receive all positive rewards all sub-goals have to be executed in a strict sequence. The results showed that without reward shaping, the agent could not successfully complete the task even after 500.000 training sessions. On the other hand, with the mentioned approach, the agent managed to successfully complete the task even when the training setup had 10.000 different starting positions. Popov et al. (2017) demonstrated similar approach with a robotics problem where a brick had to be picked-up and placed on top of another brick. The main goal was deconstructed into sub-goals, which were used to form multiple reward functions. The results showed that the agent could not reach goal state with sparse reward signal. On the other hand, with reward shaping, the agent learned to successfully complete one of the sub-goals. A follow-up approach presented by Popov et al. (2017) to further facilitate the training process will be discussed in the Section 2.3.2.

Nevertheless, reward shaping can also result in unwanted behavior or negatively affect the training process if not designed with great care (Sutton & Barto, 2018; Y. Ng et al., 1999). Randløv & Alstrøm (1998) observed and explained a perfect example of such scenario occurring in reinforcement learning problem where an agent had to learn to drive a bicycle in a simulation. In their very first training the agent received a positive reward signal for driving in the direction of the goal. This resulted in an undesirable behavior - the agent learned to maximize that additional reward signal by driving in circles around the starting position. Another simple example can be found in a project by Yu, Palefsky-Smith, & Bedi (2016), where an agent had to learn to navigate a vehicle in a game using deep reinforcement learning algorithm. Two different reward functions were tested, which were designed to punish collision and inactivity while encouraging acceleration. Even though both functions were designed with same behavior in mind, the results clearly showed that one of them had a negative impact on training process as the agent showed no promises of learning an optimal policy.

Designing an effective reward function for any reinforcement learning problem can easily become a challenging task, especially when the solution for that problem is not obvious. From RL perspective, video games are generally considered as complex challenges due to the fact that they represent a continuous problems, which can have high observation and/or action space. However, in most cases there is no need to define complicated reward functions that have to encapsulate the rules of the game. The reason for that is simple - it is common for a game to have a score scheme that can be considered as a sufficient reward signal for the agent. In the DeepMinds' approach with Atari 2600 games, the agent receives positive (+1) or negative (-1) reward signal based on the changes in the game score, while receiving 0 reward when no changes occur (Mnih et al., 2013, 2015). Unfortunately, that is not the case with real world problems. For these to be solved with reinforcement learning, highly complex reward functions have to be designed, which can potentially consume tremendous amount of resources.

A number of approaches that showed great potential have been suggested as well as researched to address the challenge of designing a robust reward function. Some of them are:

1. Inverse Reinforcement Learning (IRL), which main purpose is to infer the reward function of a given MDP specifically from human demonstration (Ng & Russell, 2000). Kretzschmar, Spies, Sprunk, & Burgard (2016) showed application of their IRL method on a robot that learned to mimic navigation behavior of human pedestrians. Furthermore, Fu, Luo, & Levine (2017) proposed IRL algorithm that derives from adversarial reward learning formulation, which can learn both reward and value functions even from highly dynamic environments.
2. Imitation Learning (IL), similarly to IRL, is used to learn directly from an expert agent, such as human, with supervised learning (Sutton & Barto, 2018). Sermanet, Xu, & Levine (2016) showed how the reward function learned from a few demonstrations can be transferred to a real world RL agent to perform the same task. Zhu et al. (2018) demonstrated how an agent learned to solve a task based on motor activities that depend on visual coordination. Most importantly, they showed that with a combination of RL and IL a greater performance can be achieved compared to scenarios where only reinforcement or imitation learning was applied.
3. Application of Temporal Logic (TL) as specification language, which is used to assign tasks to reinforcement learning agents. One example of such is Temporal Logic Policy Search method that can find a policy based on provided TL definition (Li, Ma, & Belta, 2017). Another example is Truncated Linear Temporal Logic (TLTL) method that replaces reward function with real-valued TL formula (Li, Vasile, & Belta, 2016).
4. Curriculum Learning (CL), that is used to provide an optimal challenge for the RL agent by gradually increasing the difficulty of the problem until the goal in question is satisfied. This approach will be discussed in the following Section of this thesis.

## 2.3 Curriculum learning:

### 2.3.1 Definition

Curriculum learning, as defined by Bengio, Louradour, Collobert, & Weston (2009), is a training approach where the agent has to learn sub-goals, or simplified tasks, of increasing difficulty before the main goal is presented. In the scope of autonomous car problem, the original agent's goal can be as complex as driving a car in an area with high traffic density, while starting goal can be as simple as driving in a straight line. This means that the agent's chances of receiving a positive reward signal are relatively high starting from the very beginning of the training. More importantly, the agent is learning from sparse reward signals. However, even though this

approach eliminates the need of designing a complex reward function, the need shifts onto the construction of an appropriate curriculum that will be guiding the agent's training process.

This approach can be viewed as an extension of the transfer learning method, which is based on training an agent on one task to improve its learning performance on the second task (Bengio et al., 2009; Taylor & Stone, 2009). Transfer learning is based on a possibility of utilizing agent's knowledge, which is acquired from training on *source* tasks, to facilitate the training process on the *target* task. This, of course, means that the source tasks have to be similar to the target task to some degree. In the case of curriculum learning, the source tasks can be observed as gradually simplified target tasks that are presented to the agent in an order of increasing difficulty. The accumulated knowledge, in a perfect scenario, can reduce the time needed to solve the original target task and result in a less error-prone behavior.

### 2.3.2 Related work and forms of curriculum

Bengio et al. (2009) demonstrated how curriculum learning approach can be applied to a simple classification problem solved with supervised learning. The task itself was to classify a geometrical shape into 3 different classes based on a 32 by 32 pixel gray-scale image as an input. Their experiment involved dividing original dataset into 2: one with basic shapes and the other one with similar shapes that had high variation in position, size and orientation as well as coloring. The curriculum incorporated 2 stages: training started on first set with basic shapes and then, half way through the training, switched over to the second training set. The overall results showed that the model, which was trained without curriculum, had higher test error compared to the model that was trained with 2-stage curriculum. Bengio et al. (2009) made a conclusion that "*curriculum learning adds the notion of guiding the optimization process, either to converge faster, or more importantly, to guide the learner towards better local minima.*" (p. 8).

In respect to reinforcement learning, a significant number of CL applications have been demonstrated as well. Juliani (2017b) presented a simple navigation task where an agent has to jump over a wall using a block-shaped object and reach a certain location or a goal. The agent receives positive reward signal only when it reaches the goal. The challenging part is the fact that the block-shaped object has to be pushed against the wall, otherwise, the agent cannot jump over it. The curriculum, as shown on Figure 2.3, was designed in the following way:

- The goal would always stay the same - a certain position, which is picked randomly, has to be reached by the agent.
- The initial task does not contain the wall, therefore, the agent can reach the goal without facing any obstacles.

- Each subsequent task will contain the wall, whose height will be gradually increased with every subsequent lesson. It is important to note here that until the wall reaches a certain height, the task can be completed without the block-shaped object.
- The target task is the original problem, where the agent has to jump over the wall using the block-shaped object.

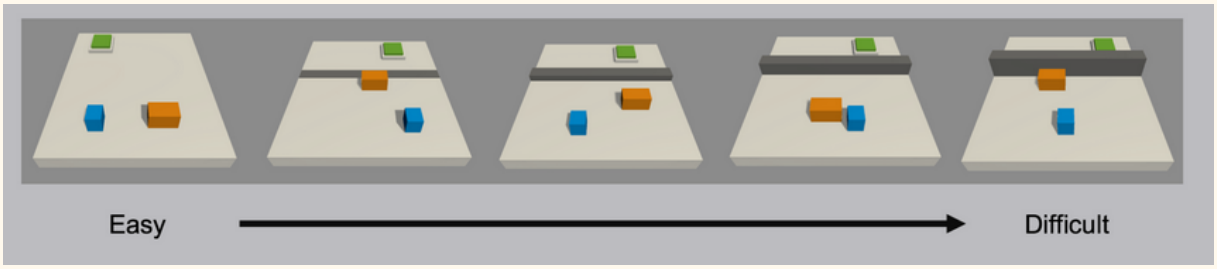



***Figure 2.3**: Example of a curriculum training scenario used for solving a navigation task, with increasing difficulty presented from left to right (Juliani, 2017b).*

The results show that without the curriculum, the agent could not learn an optimal policy even after 3 million training steps, thus, it could not solve the problem. The agent's mean accumulated reward has been gradually increasing throughout the training, however, this means that significantly more training time is needed to potentially solve the task. On the other hand, the agent that was training with the described curriculum managed to solve all the initial tasks as well as the target task within the given time.

Relatively similar, however, more complicated environment-centered curriculum learning approach for learning complex locomotion behavior has been presented by Heess et al. (2017). Here, environment-centered curriculum also means that the agent is trained in a changing learning environment. However, in this case, the environmental changes are more complex and can be scaled to represent different levels of difficulty, which is possible due to the variation of obstacles that can be added to the environment. Moreover, as the task itself involves locomotion and the agent is rewarded for moving forward through a course, the path itself can also be altered to add complexity. For example, the path can have vertical incline, contain gaps or rough patches. Furthermore, each new episode contains a newly generated course based on specific curriculum parameters, which describe what type of obstacles and/or path variations must be added. Figure 2.4 shows few examples of courses generated with different types of obstacles. Heess et al. (2017) demonstrated how training locomotion agents with environment-centered curriculum resulted in development of such robust behavior like jumping, ducking and obstacle avoidance without the need to design complex reward function.

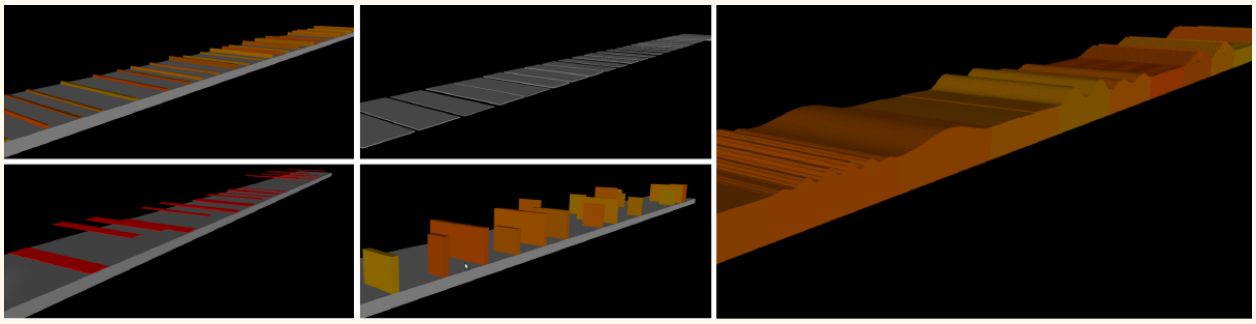


***Figure 2.4**: Examples of a different types of obstacles used to train locomotion agents (Heess et al., 2017).*

A different perspective on solving locomotion problem with curriculum learning has been demonstrated by Yu, Turk, & Liu (2018). In this case, the curriculum is based on different levels of physical assistance that facilitates the learning of locomotion skills. The assisting forces, which are calculated based on the current level of difficulty of the curriculum, help the agent to stay in a certain posture while, for example, moving forward. As the agent improves, the assisting forces are gradually decreased until the point when the agent can solve locomotion tasks completely on its own.

As mentioned in Section 2.2.2, Popov et al. (2017) demonstrated how a robotics problem of stacking one brick on top of another one couldn't be fully solved neither with sparse reward signals nor with a more complex reward functions, which rewarded agent for completing sub-goals. However, they proposed an additional method that can be used in combination with reward shaping, which they describe as "*a form of apprenticeship learning in which we provide teacher information by influencing the state visitation distribution.*" (Popov et al., 2017). This can be seen as a form of curriculum learning, where the initial state would be periodically set to the one in which, for example, the brick is already picked up by the robotic arm. Such initialization means that specific parts, or sub-goals, of the target task can already be considered as complete. Therefore, the agent is placed in such a state from where it is more likely for it to receive a positive reward signal or larger accumulated sum of rewards. The presented results show that such curriculum approach had no significant impact on the training with sparse reward signals, however, it had a notable impact on scenarios where it was applied in combination with reward shaping. More importantly, the agents could now solve the full task. Furthermore, such type of curriculum can be automated by applying reverse curriculum generation (Florensa, Held, Wulfmeier, Zhang, & Abbeel, 2017). This method can be used to generate simplified starting states based on the task and the initial state. Similarly to the previous method, the agent starts training with the initial state being relatively close to the goal state. The agent's progress is constantly monitored in order to adjust the next initial state if a certain performance level is reached.

# III. Method

In this chapter, the reinforcement learning problem of learning navigation as well as the tools used to create a learning environment and its setup are described. Afterwards, the structure of the environment-centered curriculum approach and the design of each individual lesson are presented.

## 3.1 Problem description

The problem raised in this thesis is twofold. Firstly, the agent has to learn to navigate a car through a track area to reach a finish line, while in control of both, acceleration and steering. Secondly, the path that the agent has to take can be curved to a certain degree, which is dictated by the obstacles that are placed on the track area with some amount of randomization. Therefore, the agent has to learn to rely on the input of the car's sensors to avoid collision. Finally, the problem has to be solved using sparse reward signal.

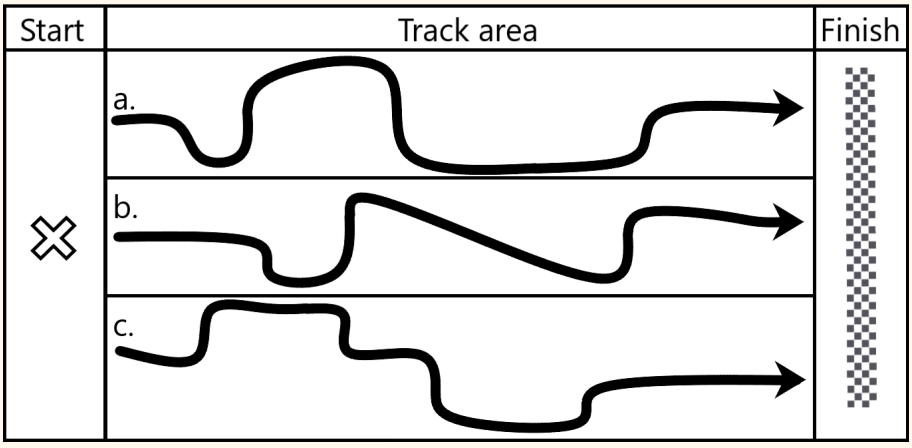



***Figure 3.1**: Training area example, showing 3 different path variations affected by the changing track area.*

The task itself involves the car driving from the left side of the track to the very end of it on the right. As show on Figure 3.1, the start is the initial position of the car, while the finish is a position that the agent is required to reach for the task to be considered as complete. Neither start nor finish positions change during the training process. However, that is not the case with the track area. This part is re-initiated each time the terminal state of the learning environment is reached. This means that all previously placed obstacles are discarded, while new ones are placed on the track area. This approach ensures that there is no single dominant path that the agent can take each time to successfully complete the task. Figure 3.1 presents 3 different path variations as an example. However, it is worth to point out that a single generated track area can have multiple paths that may lead the agent to the finish.

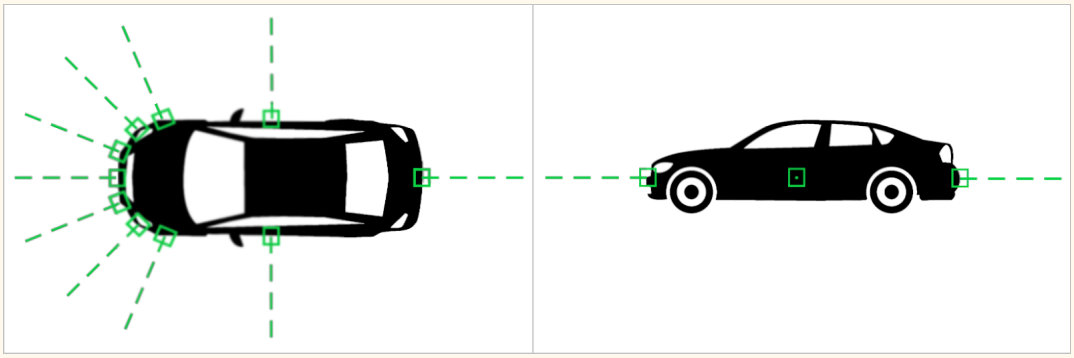


***Figure 3.2**: The top (left) and the side (right) view of the car demonstrating sensor layout.*

As previously mentioned, the car has a certain number of sensors that can detect obstacles in front of them. As presented on Figure 3.2, the car has 10 sensors in total. Based on the track design, the car is required to constantly drive forward, thus, most of the sensors are position on the frontal area of the car. Nevertheless, as the car is also capable of driving backwards, while also steering left and right, there are 3 sensors in total that are positioned on both, left and right, sides of the car as well as the back. Height-wise all the sensors are positioned just above the wheel axles and at points where both front and back bumpers are at their furthest from the center of the car.

## 3.2 Unity Engine and ML-Agents toolkit

### 3.2.1 Description

The learning environment is developed in Unity Engine. This game engine allows high flexibility when it comes to creating different types of physics-based simulations, animations, 3D world design, etc. (Unity Technologies, 2018a). Apart from that, one of the main reasons for choosing Unity Engine for this project is the machine learning plugin. Unity launched their machine learning solution called Unity Machine Learning Agents Toolkit (ML-Agents) in September of 2017 (Juliani, 2017a). ML-Agents is an open-source Unity Engine plugin developed to be used not only by video game developers but also by both academic and industry researchers. This toolkit allows to develop, train and test intelligent agents using different machine learning methods in simulated environments created within Unity Engine.

ML-Agents toolkit consists of 3 high-level modules:

- Learning Environment
- Python Application Programming Interface (API)
- External Communicator

The learning environment in this case is the simulation, which has to be designed and constructed in Unity Engine. On the other hand, the machine learning algorithms are implemented into the Python API, which is the part of ML-Agents toolkit that runs outside of the simulation. During the training process, the communication between the simulation and the Python API is handled by the External Communicator.

### 3.2.2 Proximal Policy Optimization

ML-Agents toolkit provides a possibility to train reinforcement learning agents using a method called Proximal Policy Optimization (PPO) (Juliani et al., 2018). PPO is the state of the art model-free reinforcement learning algorithm, which have been released by OpenAI (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017). This method is based on approximating the policy function that maps state observations to specific actions. The approximation is done with a neural network, where the state is the input and the corresponding action is the output. The neural network is used to optimize the policy in order to maximize the future reward.

Moreover, PPO has become the default RL algorithm in all future OpenAI researches due to its overall performance and straightforward implementation. One example of PPO application has been previously presented in Section 2.3.2, where the agent learned complex locomotion behavior using simple reward signals and curriculum learning approach. In the same paper, Heess et al. (2017) compare performance of their implementation of PPO algorithm against other two baseline algorithms: Trust Region Policy Optimization (TRPO) and Asynchronous Advantage Actor-Critic (A3C). Both, TRPO and A3C algorithms, have showed substantial improvements in either generalization and/or performance in the past (Mnih et al., 2016; Schulman, Levine, Moritz, Jordan, & Abbeel, 2015). Nevertheless, PPO has surpassed them by achieving similar level of performance within a shorter amount of time.

### 3.2.3 Basis of a learning environment

To create a machine learning environment in Unity Engine, it is necessary to setup these 3 components (Juliani et al., 2018):

- *Agent* - a component that collects the observation data from the environment, executes specific actions and receives reward signals based on the new state of the environment. The Agent cannot operate on its own as it linked to a Brain, which controls the action selection.
- *Brain* - specifies the size and type of the observation and action space in the learning environment. Both can be either continuous or discrete. As previously mentioned, it is the Brain that dictates which action the Agent has to carry out based on the learned

policy and the current state of the environment. Essentially, the Brain component takes observations and reward signals as an input and outputs actions.

- *Academy* - a final component of the learning environment that establishes a synchronous observation state and reward signal collection as well as action selection between each Agent/Brain pair. Furthermore, the Academy holds a set of variables that control the learning environment. For example, the length of a single episode and the overall speed at which the environment is running are set within the Academy.

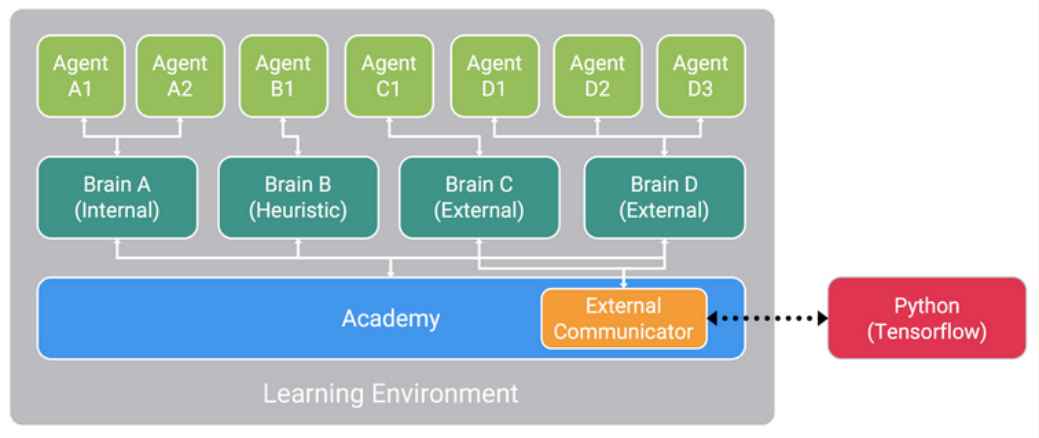



***Figure 3.3**: An example of how a learning environment structure can be configured using ML-Agents toolkit (Juliani, 2017a).*

The 3-component system makes ML-Agents a versatile toolkit that can be adapted to a great number of different training scenarios. As shown on Figure 3.3, both single and multi-Agent training scenarios can run within the same learning environment at the same time. Multiple Brains can be used to control one or more Agents of different types. A simple case where such setup could be applied is an example environment called "*Soccer Twos*" that is part of the ML-Agents toolkit (Juliani et al., 2018). This learning environment contains 4 Agents of 2 types: the striker and the goalie. Because each of them have different tasks as well as action spaces they must have separate Brains. On the other hand, the same Brain can control both striker type Agents at the same time. Training multiple instances of same Agent simultaneously means that the training time needed to reach certain reward levels can be reduced significantly. The Agent learns faster due to the fact that multiple state observation vectors are fed into the Brain at the same time rather than just one.

### 3.2.4 Training process

The training process performed with ML-Agents toolkit is not in any way different from a typical reinforcement learning training, which has been presented in Section 2.1. During training, as the very first step, the Agent collects the current state observations. Secondly, these observations are transferred to the Python API using the External Communicator. Subsequently,

the Python API processes the received data using Proximal Policy Optimization method, presented in Section 3.2.2, to learn the best policy for the presented problem. Finally, the Agent receives an action that it has to take in that specific state.

In the current version of the toolkit, which is v0.4 as of writing this thesis, the Brain object has 4 distinctive modes in which it can operate:

1. *Player* - the actions are selected using an input from, for instance, a keyboard. As the name suggests, in this case a human player controls the Agent. The observation states and reward signals have no impact at this point.
2. *Heuristic* - the actions are selected using a predefined behavior, which is hard-coded and cannot be altered during evaluation process.
3. *External* - the actions are selected by the previously presented Python API. In this case, the accumulated rewards and observations are fed to the Python API, which outputs an action the Agent has to make based on the learned policy.
4. *Internal* - the actions are selected by the pre-trained TensorFlow model, which represents previously learned policy. More specifically, the Brain uses this model, which is embedded using TesorFlowSharp, to output a corresponding action given an observation state and a reward signal.

### 3.2.5 Curriculum learning

As of version v0.2, the ML-Agents toolkit includes a built-in support for Curriculum learning (Juliani, 2017b). It is a rather simple feature that can be implemented to any machine learning project made in Unity Engine. Before starting the training process as usual, a JSON file, which defines the structure of the curriculum, has to be specified. The following parameters have to be defined in the curriculum's JSON file:

1. *Measure* - defines whether the lessons should change based on a certain percentage of the overall training steps or based on accumulated reward.
2. *Thresholds* - the actual values that the measured parameter has to reach. For example, that could be the accumulated reward of 0,5.
3. *Minimum lesson length* - this parameter determines how many times the threshold of the measured parameter has to be reached.
4. *Signal smoothing* - can be set to either *true* or *false*. If enabled, the measure signal is re-calculated by taking 75% of the newly received value and 25% of the previous one. Only afterwards, the re-calculated value is checked against the defined threshold.
5. *Reset parameters* - the parameters that are defined in the Academy component, which are accessible by any component or the Agent in the learning environment.

## 3.3 Learning environment

### 3.3.1 Environment setup

As the task involves only a single autonomous car driving through a generated track, the scene was set up in a rather simplistic manner:

- One Academy component to control global episode length as well as assign curriculum lesson if such is needed.
- One Brain component to control the only Agent in the scene, which in this case is the autonomous car.
- The Agent itself, that controls the autonomous car in the scene and collects states observations.

Such simple setup also meant that multiple Agents of the same type could be trained within the same scene simultaneously using a single Brain component. However, the training area as well as the curriculum generation had to be built with multi-Agent setup in mind. This meant that one Agent shouldn't be able to affect the observation space of the other. Therefore, each Agent had to be placed in a dedicated training area. Such area can be viewed as a car/track pair, where each Agent can train on a personal track without any interruptions until the end of the training episode.

The hierarchical structure of a single training area is:

- ***Training area*** - parent object that contains *Curriculum_final.cs* script (see Appendix A), which rearranges the track area based on the current lesson of the curriculum.
  - *Car* - object that contains *CarAgent_final.cs* script (see Appendix B), which is the Agent that collects state observations, passes reward signals and controls the car as well as the setup of necessary components. These components are:
    - *Colliders* - objects that are used to register collision of the car with the obstacles found on the track. 3 box-shaped colliders are used to define simplified car shape.
    - *Wheel Hubs* - parent object for wheel colliders, which are used as car wheels.
    - *SkyCar* - parent object that contains all visual car model parts that come with Standard Assets pack found in Unity Engine.
    - *Sensors* - simple box-shaped objects, which are used as reference points for the actual sensor rays. Both position and rotation vectors of the sensor object are used for a raycast.

- ❑ *Finish Line* - an object that is used as a target, which the car object has to reach in order to complete the task in question.
- ❑ *Obstacle Areas* - reference points, which define the starting points of separate obstacle grids. Each individual grid is generated using *ObstacleGridGenerator.cs* script (see Appendix C).
- ❑ *Ground and Walls* - objects that define the boundaries of the training area.

Moreover, such setup also allows to immediately reset individual training areas where the Agent has reached the terminal state. This can facilitate the training process in the multi-Agent scenario due to the fact that a single Agent can start a new local episode without the need to wait for each Agent to also reach its terminal state. Thus, one Agent can complete multiple local episodes until the global episode ends. The local episode is over when the Agent reaches the terminal state, which can happen due to the following reasons:

1. The car collides with an obstacle.
2. The car crosses the finish line.
3. The global episode, controlled by the Academy, is over.

### 3.3.2 Observation space

Each state in the learning environment is represented as a continuous vector of features. The vector is an array of numbers, which are collected by the Agent during training. Each state input consists of the following observations:

1. Relative position - X and Z coordinates of the car that are calculated relatively to the position of the finish line. The Y coordinate is discarded due to the fact that all the generated tracks are flat and do not contain any types of uneven terrain. The changes in the Y coordinate can be observed during training due to suspension that is active on each wheel of the car. However, this observation is considered insignificant.
2. Velocity - similarly to the relative position, only X and Z directions are taken as state input. Velocity vector measurements are taken directly from a Rigidbody physics component that is attached to the car object.
3. Sensor data - a float array, where each number represents an output of each individual sensor that is positioned on the car. The output number itself is a calculated distance from the sensors origin position to the closest obstacle, which is an object that can collide with the car. To detect such obstacles, each sensor creates a raycast, which registers a hit point if an obstruction is observed on the ray's path. If the sensor is not registering any obstacles within specified range in front of it, the output number is set to 0. The sensor data is gathered at every step of the simulation before the state observations are collected.

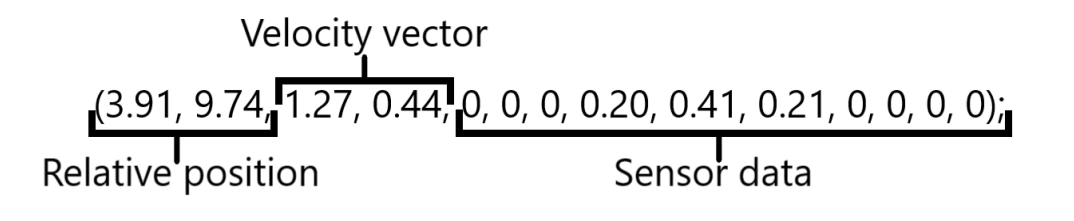



***Figure 3.4**: Structure of a single state observation vector.*

As shown on Figure 3.4, the total observation space size equals to 14. This includes: two inputs that indicate car's relative position, two that are based on car's velocity and ten collision-based sensory inputs.

### 3.3.3 Action space

The car's control scheme is designed around two specific parameters: acceleration and steering. The control interface has been set up in such way that the acceleration is handled by applying a specific amount of torque onto the car wheels, while the steering is controlled by passing a certain number as an angle at which the wheels should be positioned. This is made possible by using in-built *Wheel Collider* component in Unity Engine (Unity Technologies, 2018b). This component is designed to be used specifically for simulation of realistic vehicle wheels. Each wheel can be configured to meet certain specifications as each of them has its own mass, suspension spring, friction and damping rates, etc. The developed interface allows to add specific number of axles to the car. Each of these axles has to have one left and one right wheel assigned to it. Moreover, the interface allows to specify whether a specific axle has a motor and if the car's steering controls the rotation of the wheels on it. The car is designed to have 2 axles with 4 wheels in total. The car is setup to be a typical four-wheel drive - motor torque applies to both axles, while the steering only controls the front axle.

The action space has been chosen to be of a discrete type due to the limiting capabilities of the machine that is used to train the RL Agents, as mentioned in Section 1.3. However, it still must contain a fair amount of possible actions for the Agent to choose from in order to develop a robust behavior. Thus, as summarized in the Table 3.1 below, the developed action space contains eight possible actions in total. The motor torque and the steering angle parameter values are assigned using two separate parameters, which define the maximum possible value the original parameters can have. Moreover, both motor torque and steering angle parameters can have negative values. With negative torque, the car will start accelerating backwards, while negative steering value will rotate car wheels counterclockwise, rather than clockwise. It is worth pointing out that the car wheels will have the same direction vector as the car itself when the steering angle value is set to 0.

| Index | Action description | Value assignment | Result |
|---|---|---|---|
| 0 | Forward torque - half speed | motorTorque = maxMotorTorque / 2 | Torque = +500 |
| 1 | Forward torque - full speed | motorTorque = maxMotorTorque | Torque = +1.000 |
| 2 | Backward torque - half speed | motorTorque = -maxMotorTorque / 2 | Torque = -500 |
| 3 | Steering angle - full right | steeringAngle = maxSteeringAngle | Wheel angle = +50 |
| 4 | Steering angle - half right | steeringAngle = maxSteeringAngle / 2 | Wheel angle = +25 |
| 5 | Steering angle - full left | steeringAngle = -maxSteeringAngle | Wheel angle = -50 |
| 6 | Steering angle - half left | steeringAngle = -maxSteeringAngle / 2 | Wheel angle = -25 |
| 7 | Steering angle - straight | steeringAngle = 0 | Wheel angle = 0 |

***Table 3.1:*** *The list of actions that are available to the Agent during the simulation.*

### 3.3.4 Reward signals

The developed interface allows to assign the following reward signals during training:

- *Sparse reward* - the Agent receives a reward of a defined value as soon as the car crosses the finish line. The state, during which the Agent receives this reward, is considered as the terminal state, thus, the episode concludes afterwards.
- *Distance-based reward* - a calculated reward, which the Agent receives after the car covers a certain distance, while navigating towards the finish line. The value that is assigned as the reward is the same value, which is defined by the user. However, the same value is used to calculate a section of the distance based on the initial distance between the car and the finish line. As an example, with a defined reward being set to +0,1 with initial distance to the finish line set as 10 units, the Agent will receive +0,1 reward each time the section with length of 1 unit is covered. If the defined reward is set within the range of [0 < x < 1], the maximum accumulated reward will never be higher than +1. This is a finite reward signal, which should tackle the issue of driving in circles, previously described in Section 2.2.2.
- *Step reward* - a defined value, no matter positive or negative, which the Agent receives as a reward at every step of the simulation.

Moreover, a combination of these reward signals can also be used during the same training process. For instance, the Agent can potentially receive a reward signal of +1 for reaching the target state, while a small negative step-based reward could be assigned for reaching other states of the simulation.

## 3.4 Environment-centered curriculum

### 3.4.1 Curriculum description

The problem of learning collision avoidance while navigating towards a certain location have to be tackled separately, similarly to the wall-jump example presented earlier. First of all, the Agent has to learn to navigate in a very simplistic scenario to be able to reach the finish line. Otherwise, with a full set of obstacles present in the training area, possibility to receive a positive sparse reward signal can be close to impossible. Therefore, the initial lesson has to be rather easy to solve. This means that additional obstacles, that could shape the track, have to be left out until the Agent learns how to navigate the car. Nevertheless, the track still contains boundaries, thus, some notion of obstacles is maintained. After a certain accumulated reward threshold is reached, the Agent can move on to the second stage of the curriculum, which involves additional randomly generated obstacles. The difficulty of the second stage depends directly on a number of obstacles placed on the track. Therefore, each subsequent lesson will have a higher number of generated obstacles than the previous one.

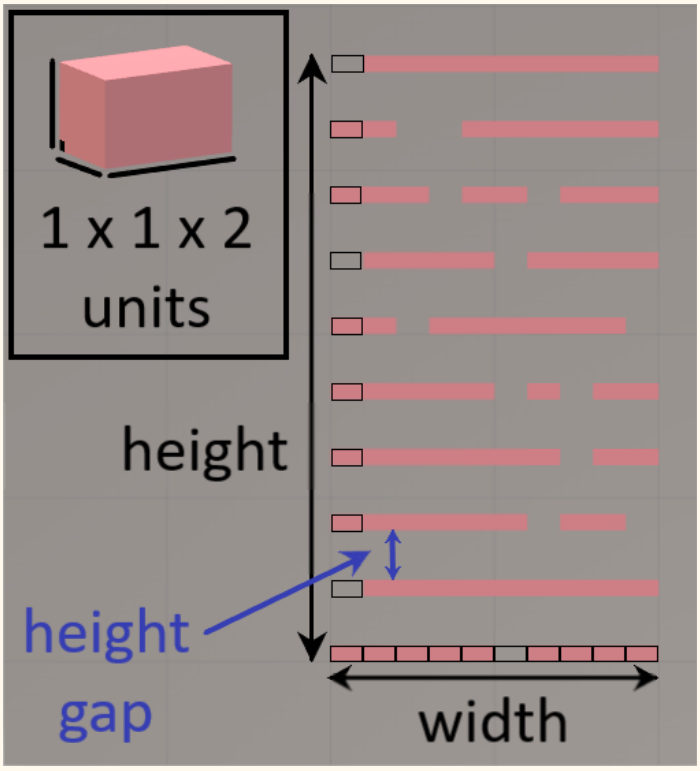



***Figure 3.5**: Example of a generated 10 by 10 obstacle grid with minimum and maximum gap numbers set to 1 and 2 respectively.*

### 3.4.2 Obstacle generation

As stated previously in Section 3.3.1, *ObstacleGridGenerator.cs* script (see Appendix C) is used to generate a random grid of obstacles based on user specified parameters. These parameters are: *width*, *height*, *height of gap*, *minimum number of gaps and maximum number of gaps*. The *height* value determines the number of obstacle rows, while the *width* sets the number of

actual obstacle objects in a single row. An example of an obstacle grid with both the *height* and the *width* set to 10 is shown on Figure 3.5. The *height of gap* parameter defines the interval in between rows. Moreover, the size of the generated grid also depends on the scale of the obstacle object. In this case, the object has its height and width set to 1, while its length is set to 2. For instance, if the length of the obstacle object is two times greater, a grid with twice longer rows will be generated. Finally, the last two parameters, the *minimum* and the *maximum number of gaps*, define a range, which is used to randomly set a number of gaps in each individual row. For example, if a specific row has to have one gap, then one random obstacle object will not be placed in that row. In the example shown on Figure 3.5, the *minimum number* is set to 1, while the *maximum* is set to 2. Thus, each row has either one or two randomly picked obstacle objects removed from it.

### 3.4.3 Curriculum lesson setup

As mentioned in the Section 3.3.1, the Academy component determines the current curriculum lesson. The lesson number is based on a single *CurriculumLesson* reset parameter that is defined by the CL feature of the ML-Agents toolkit. The built-in curriculum learning feature is used to measure the accumulated reward with a threshold set to 0,75. This specific threshold value has to be reached at least ten times. Afterwards, the *CurriculumLesson* reset parameter is incremented. Moreover, the signal smoothing is also applied ir order to avoid lesson change if a random surge of the accumulated reward is registered.

The environment-centered curriculum is defined in the *Curriculum_final.cs* script (see Appendix A), previously mentioned in Section 3.3.1. Whenever necessary, the curriculum can be reset based on the current lesson number provided by the Academy component. Each lesson introduces specific changes to the training area, which determine its complexity. The lessons are setup in the following way:

**Lesson 1**: The initial lesson of the curriculum, which does not contain any additional obstacles apart from the boundaries that isolate the training area. Furthermore, as shown on Figure 3.6, the finish line is moved closer to the car's initial position, which makes the track area approximately 3 times shorter than what is intended.

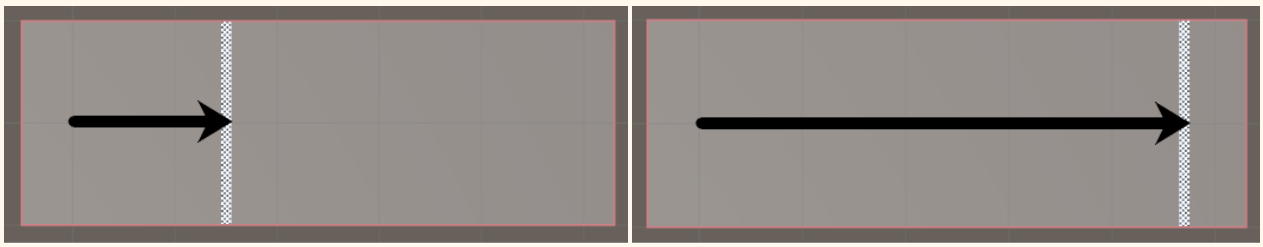


***Figure 3.6**: 1st (left) and 2nd (right) lessons of the designed curriculum.*

**Lesson 2**: This lesson is designed to be a more complex version of the initial one, where, as presented on Figure 3.6, the finish line is moved to the very end of the training area. Moreover, the car is now randomly rotated on the Y axis at the beginning of each new episode. This adds additional challenge due to the fact that car no longer can simply drive forward and is forced to steer towards the finish line. The location of the finish line and the random rotation of the car defined in this lesson will persist in the subsequent lessons of the curriculum.

**Lesson 3**: The lesson that transfers the Agent to the second stage of the curriculum, which presents randomly generated obstacles for the first time in the training process. In this lesson, the obstacle grid is generated using the following parameters:

- *Width*: 10
- *Height*: 8
- *Height of gap*: 3
- *Minimum number of gaps*: 9
- *Maximum number of gaps*: 9

With such parameter setup, the generated grid will have 8 rows in total with only one obstacle object placed in each of them. As shown on Figure 3.7, such obstacle grid can have a slight or almost nonexistent impact on the navigation trajectory. Moreover, *width*, *height* and *height of gap* parameters setup presented in this lesson will remain unchanged in the upcoming lessons.

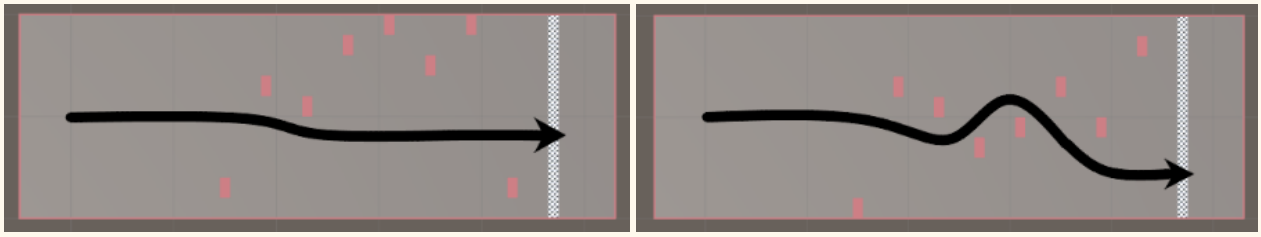


***Figure 3.7**: Two examples of the 3rd curriculum lesson.*

**Lesson 4**: This lesson is designed to add more complexity to the training process, compared to the previous one, by introducing a higher number of obstacles. This is achieved by changing the *minimum* and *maximum number of gaps* parameters, related to obstacle grid generator, to 6 and 8 respectively. Figure 3.8 presents two separate instances of the 4th curriculum lesson.

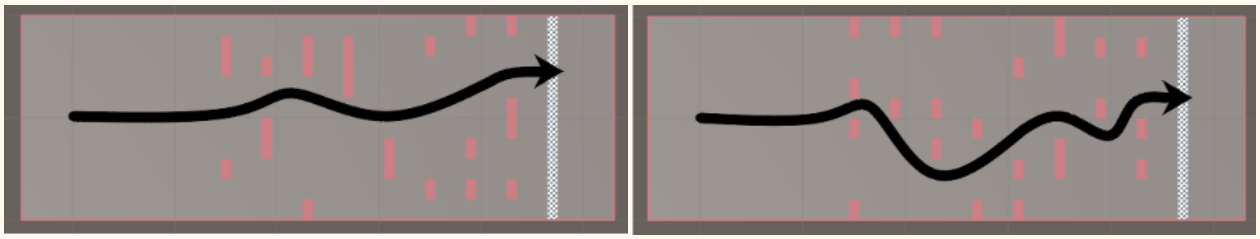


***Figure 3.8**: Two examples of the 4th curriculum lesson.*

**Lesson 5**: The final lesson of the curriculum presents the target task to the Agent. In this lesson, the *minimum* and *maximum number of gaps* parameters are set to 3 and 4 accordingly. As shown on Figure 3.9 below, this lesson, while taking into consideration all the changes that were introduced with the previous lessons, can still be of either low (left) or high (right) complexity.

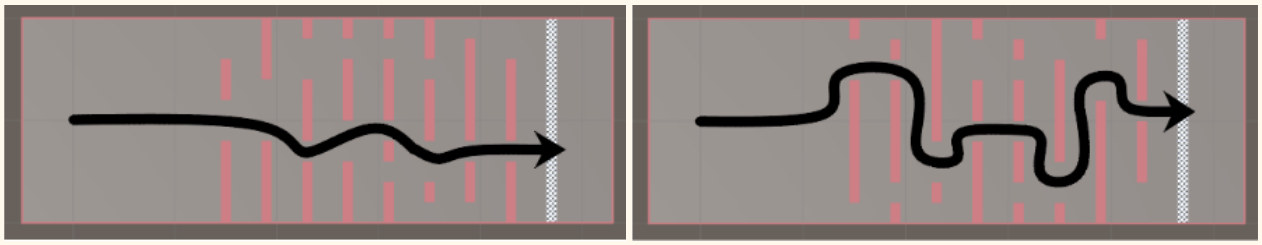


***Figure 3.9**: Two examples of the target task, or the 5th curriculum lesson.*

# IV. Experiments and results

The following chapter describes the conducted experiments, their conditions as well as the acquired results and their evaluation criteria. Firstly, the initial experiments, where extensive testing of specific learning environment components and parameter were done, are presented. Secondly, the main experiment, where multiple training scenarios in respect to curriculum learning and reward shaping approaches were conducted, is introduced.

## 4.1 Initial experiments

### 4.1.1 Conditions

Over 300 separate training sessions were conducted to explore and test initial implementations as well as different iterations of the final learning environment (see Appendix D and E). These iterations included changes to: hyperparameters of the RL model, both observation and action spaces, Academy and Agent setups, Car components, track area and, most importantly, the curriculum lessons. The subsequent Sections will go through each major component iteration process. Moreover, the most suitable parameters and setups were chosen based on these criteria:

1. Amount of time required to finish the training process.
2. Amount of steps required to reach the final lesson of the curriculum.
3. Mean accumulated reward.

Finally, all the tests presented below were conducted on the curriculum-based training scenario with sparse reward signals. Most training sessions were limited to half a million steps, while others were stopped at the point when certain results were observed.

### 4.1.2 Academy episode length

Multi-Agent setup, while bringing the benefit of accelerated training process, also led to a certain drawback. As mentioned in Section 3.3.1, each Agent can complete multiple local episodes until the global simulation episode is over. When this happens, every single Agent is reset despite their current state. Therefore, in the training scenario, where sparse reward signals are used, the accumulated reward in the very last local episode always equals to 0 due to the fact that the Agents are rewarded only when they successfully complete the given task.


***Figure 4.1**: Mean cumulative reward results of two initial training sessions demonstrating an issue of high value fluctuation, which appears at the end of each global episode.*

The Figure 4.1 shown above, represents two training sessions, where the Agents had to reach a randomly picked location on the track area. The only major difference here was that the yellow line represents the Agents, which were trained with few additional obstacles placed on the track. Moreover, this was a simplified version where the action space was limited to only 3 choices. The global episode length was set to 3.000 steps, while the local episodes were limited to just 300 steps. The situation described above can be observed as soon as the Agents reach mean reward of 0,8 - each time the global episode ends, the accumulated mean reward displays a slight drop. Even though it does not affect the training process in a negative way, the issue might arise when the curriculum learning approach is introduced. The reason for that is simple - the defined threshold of the mean cumulative reward, which has to be registered a certain amount of times in a row in order to start a new lesson, might not be reached due to such fluctuation. In order to tackle this problem beforehand, the global episode length had to be prolonged, specifically to 10.000 steps, to decrease such variation of the mean accumulated reward. Moreover, the local episode limit of 300 steps was removed, to provide Agents with more time to solve the problem. It is also important to note that the Academy component controls the curriculum lesson parameter, thus, without the step limit on the global episode the lesson change would not be triggered.

### 4.1.3 Car sensor count

As described in Section 3.1, the car is designed to have 10 collision-based sensors in total (see Appendix F). Due to the fact that each individual sensor adds additional input value to the reinforcement learning model, it was important to test whether this number can be decreased, which subsequently can potentially accelerate the training process.


***Figure 4.2**: Mean cumulative reward results of the same Agent trained with 3 different sensor setups.*

The Figure 4.2 shows the results of 3 training sessions conducted in 3 separate learning environments. The only difference between these environments were the number of sensors that were active on the car. It is important to point out that the sensors that were deactivated during these tests were positioned on the front of the car, meaning that the right and left side sensors as well as the sensor on the back were always active. The results have showed that decreasing the amount of active sensors on the front of the car negatively affected the Agents' performance. More importantly, the mean cumulative reward has dropped significantly. Therefore, the sensor setup was kept as originally designed.

### 4.1.4 Decision Frequency

In ML-Agents toolkit, by default, all Agents are set to request a decision from the Brain component at every step of the simulation (Juliani et al., 2018). However, this can be changed by adjusting *Decision frequency* parameter. Setting this parameter to a specific number, for instance, to 3, means that the Agent will make a request for a new decision only every 3 simulation steps. Moreover, it is important to note, that the same action will be picked by the Agent for the same amount of steps. In this case, if the newly picked action is acceleration, the car will accelerate for the next 3 simulation steps. Setting *Decision frequency* parameter above 1 can be beneficial in this particular case due to the fact that the impact of an action, which is executed in one state, might not be visible in the following state. Most straightforward example of such scenario can be observed in the very beginning of the training process when the car is stationary. Even if the picked action is acceleration, the change in the car's position might simply be insignificant and unobservable in the next state. However, that would not be the case if, for instance, the next state is observed after 3 full simulation steps. On the other hand, *Decision frequency* also implies that the Agent's reaction time increases proportionally.

Therefore, if this parameter is set to a significantly higher number, the Agent might not be able to react to a sudden change in the learning environment.

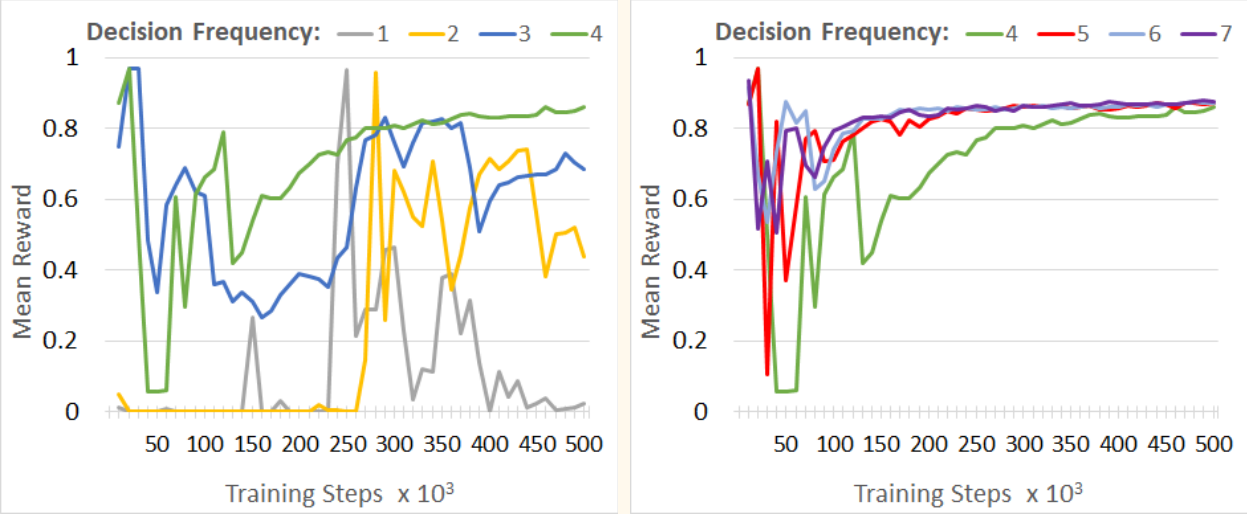



***Figure 4.3**: Two graphs showing mean cumulative reward results of the same Agent trained with different Decision Frequency.*

The two graphs presented above on Figure 4.3, show how much impact the *Decision frequency* parameter can have on the training process. With the parameter set to anything lower than 4, the Agents need at least 400.000 steps to reach the final lesson of the curriculum, thus, longer training sessions are required. On the other hand, with the *Decision frequency* set to 5, 6 and 7, the Agents showed rather similar results in regards to the accumulated reward and step count ratio. However, a correlation between this parameter and the time needed to complete the training sessions was observed - higher frequency results in slightly longer training sessions. For example, it took 2 hours and 30 minutes for the Agents with *Decision frequency* of 5 to complete the training session. An additional 8 minutes were needed to complete the same sessions with frequency set to 6. Thus, the frequency of 5 has been chosen as the default parameter value.

### 4.1.5 Multi-Agent setup

As the setup of the learning environment allows to train multiple Agents at the same time, different combinations had to be tested. As shown on Figure 4.4, the highest accumulated rewards within the same amount of training steps are received in the learning environments with the Agent count between 40 and 60. However, training a high number of Agents at the same time comes with a cost - higher amount of Agents takes significantly more time to train. Depending on the current amount, adding 10 additional Agents can prolong the training process by approximately 30 minutes. The decision was made to use the setup with 50 Agents as

it showed overall highest mean of the accumulated reward in the second half of the training process, or after 250.000 steps.

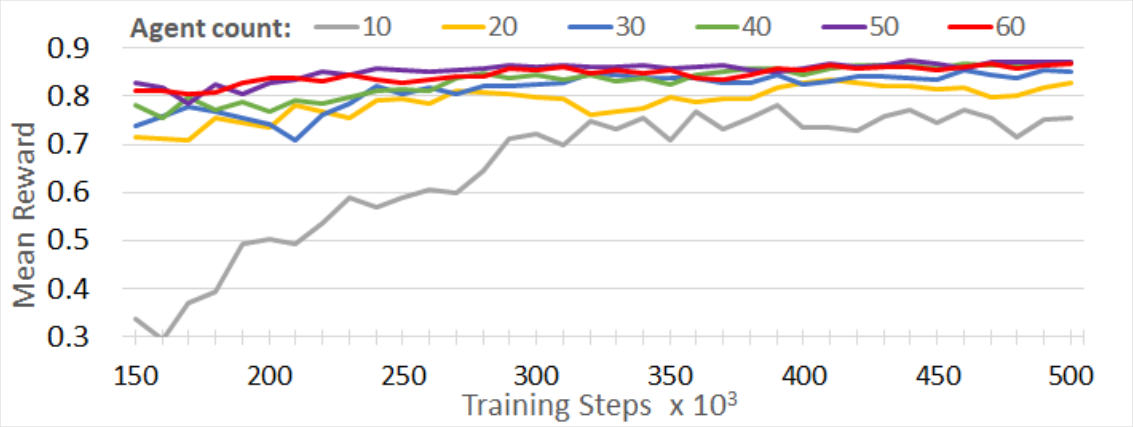



***Figure 4.4**: Mean cumulative reward results of the training sessions, which were conducted with different multi-Agent setups. The graph starts from 150.000 steps to show results, which were acquired specifically on the target task, or in lesson 5.*

### 4.1.6 Hyperparameter tuning

The hyperparameter tuning process was conducted continuously throughout the development of the implementation. The ML-Agents toolkit documentation was used to get a proper insight on each hyperparameter. The hyperparameters, which had the highest impact on the training process, are the following:

- *Beta* - controls magnitude of the entropy regularization, which subsequently controls the randomness of the policy.
- *Batch size* - defines a number of experiences that are needed for a single gradient descent.
- *Buffer size* - defines a number of experiences that are needed to update the model. Experiences include: state observations, picked actions and received rewards.
- *Learning rate* - defines magnitude of a single gradient descent.
- *Hidden units* - defines the number of units present in each fully connected neural network layer.
- *Gamma* - the discount factor of the future rewards. Defines the magnitude of how important the future rewards are.
- *Epsilon* - a value used as a threshold, which defines the magnitude of allowed change in the policy.
- *Time horizon* - defines the amount of experiences, which has to be collected per-Agent that is allowed to be added to the experience buffer.

The Table 4.1, presented below, summarizes the whole process as well as shows the chosen hyperparameter values.

| Beta | Batch size | Buffer size | Learning rate | Hidden units | Gamma | Epsilon | Time horizon |
|---|---|---|---|---|---|---|---|
| 0,0005 | 64 | 1.024 | 0,00001 | 16 | 0,96 | 0,1 | 32 |
| 0,001 | 128 | 2.048 | 0,0001 | 64 | **0,99** | 0,15 | **64** |
| 0,0025 | **256** | **2.560** | 0,0002 | **96** | 0,995 | **0,2** | 96 |
| **0,005** | 512 | 4.096 | **0,0003** | 128 | 0,999 | 0,25 | 256 |
| 0,01 | 1.024 | 5.120 | 0,0004 | 160 | | | |
| 0,02 | 2.048 | 10.240 | 0,0005 | | | | |
| 0,03 | | | | | | | |
| 0,05 | | | | | | | |
| 0,07 | | | | | | | |
| 0,09 | | | | | | | |

***Table 4.1:*** *The list of PPO hyperparameters that were tested throughout different versions of the developed implementation. The values that were picked are marked in green.*

### 4.1.7 Curriculum changes

The general curriculum structure has been tested on previous versions of the simulated learning environment. Firstly, the Agent was trained to navigate towards a randomly positioned target behind an obstacle (see Appendix E.1). The curriculum lessons were set up in a rather simplistic manner. Lesson 1 did not include the obstacle object, while Lesson 2 introduced only a part of it. The final lesson was the target task. Although the Agent managed to solve the task, the learned behavior had a negative side effect - the Agent would still try to navigate around the obstacle even if it was removed from the training area. Therefore, additional randomization in respect to the placed obstacles in the training area had to be added in order to train a more generalized behavior.

Secondly, the Agent was trained and tested on a more complex task: a longer distance had to be covered and the obstacles were randomly positioned within a specific area (see Appendix E.2). This specific design had two flaws that were discovered during the training process. First of all, the Agent was able to learn the lessons 1-4 relatively fast, meaning that this part of the curriculum had to be shortened as the lessons do not add significant amount of complexity.

Thus, lessons 2 and 3 were removed. Second of all, the Agent managed to solve all the lessons as well as accumulate a rather high mean reward. However, it was observed that the Agent developed a dominant strategy by taking a specific path on the track area, where the obstacles were placed approximately once in every ten episodes. Therefore, the random obstacle generation method presented with this curriculum version had to be redesigned. As described in Section 3.4.2, the new obstacle generation method allows to set up a grid of obstacles, while also providing more control over the randomization process.

In the first stage of the final curriculum, as mentioned previously in Section 3.4.3, lessons 1 and 2 are designed specifically with the idea of providing the Agent with an obstacle-free training environment where it can learn to navigate without any additional obstructions. However, the importance of having a significantly simplified initial lesson had to be tested.

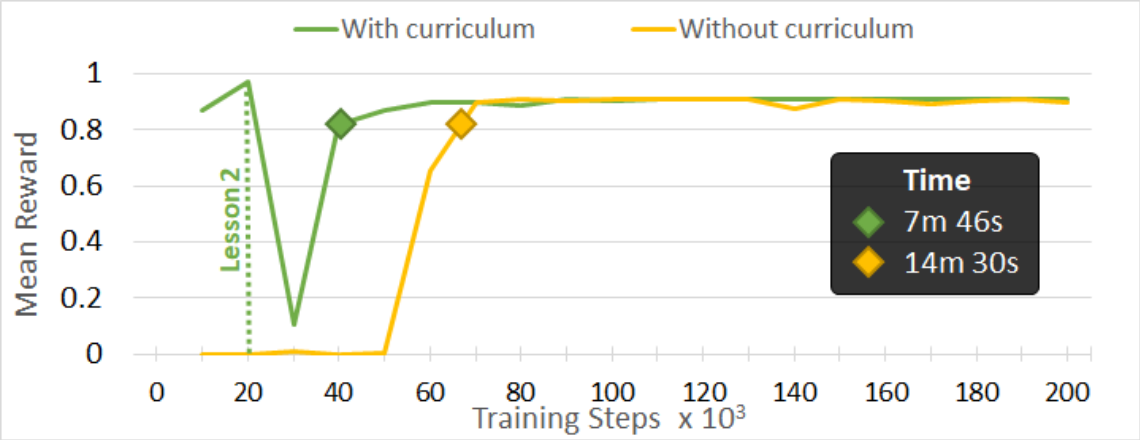



**Figure 4.5**: *Mean cumulative reward results of two training sessions where Agents were trained with and without curriculum. In this specific case, the target task is lesson 2. Two rhombus-shaped icons mark step where mean reward reaches value of +0,8.*

As shown on Figure 4.5, the Agent that started training from lesson 1 (with curriculum), managed to switch to the lesson 2 and reach mean accumulated reward of over 0,8 within 40.000 steps. On the other hand, the Agent that started immediately from lesson 2 (without curriculum), was able to reach the same reward level only around 70.000 steps. Most importantly, with curriculum the same mean accumulated reward was reached twice as fast in terms of time.

## 4.2 Main experiment

### 4.2.1 Conditions

The main experiment revolves around the main goal of this thesis - training the reinforcement learning Agent, which in this case is the autonomous car, with sparse reward signal using

curriculum learning approach. Furthermore, the Agent was trained with and without CL to evaluate its effectiveness towards facilitating the training process as well as learning an optimal policy. In the following Sections, these Agents are referred to as CL-Agent and non-CL-Agent. The non-CL-Agent, which was trained without curriculum learning, was presented with the target task, or lesson 5, from the very beginning of the training process. Finally, additional tests were conducted to study a combined application of curriculum learning and reward shaping approaches.

The training of all Agents was conducted on the same learning environment, however, with different reward signals. The environment was set up as described in the methodology, including the changes that were introduced during the pre-training stage, as presented in the previous Section. These changes are summarized below:

- Global episode length was set to 10.000 steps.
- The Agent step limit was discarded.
- The number of Agents training at the same time was set to 50.
- The Agent's *Decision frequency* parameter was set to 5.
- Overall hyperparameter tuning.

Each training scenario was completed five times using the same set of predefined seeds. These seeds are: 123, 456, 789, 97531 and 86420. A seed is used within the ML-Agents toolkit for random number generation, thus, allowing to initialize multiple training sessions with the same initial parameters (Juliani et al., 2018). Therefore, the results presented below show average values that are based on five separate training sessions.

Moreover, it was observed that the average training time fluctuated between 2h 20m and 2h 40m in all the presented scenarios, thus, it has not been taken into consideration as a valuable criteria.

Finally, the acquired results are evaluated based on the following criteria:

- The highest mean accumulated reward value.
- The point when the mean reward value of 0,75 was reached, in terms of simulation steps and training time.
- The robustness of the developed behavior.

The mean accumulated reward value of 0,75 has been chosen as a point when the task is considered as solved and an optimal policy is learned. This is done in regards to the fact that the same value is used as the threshold to trigger the curriculum lesson change.

### 4.2.2 Sparse reward

In the first stage of this experiment, the Agents were trained using sparse reward signal. Taking into consideration that the Agent's goal is to reach the finish line, the reward signal setup was the following:

- +1 - for reaching a set of states where the car is located behind the finish line.
- 0 - for being in any other state of the learning environment.

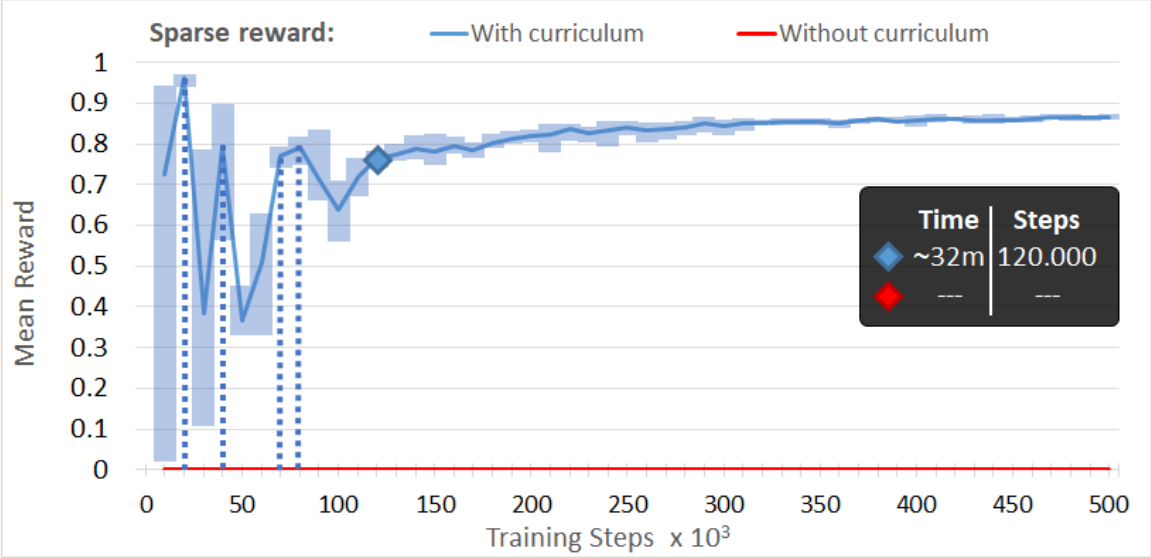



***Figure 4.6**: Mean cumulative reward results of Agents trained using sparse reward signal, with and without curriculum. The results of the two training scenarios are based on five separate training sessions. The lines show average values, while the bars correspond to the lowest and highest values acquired at the end of the global episode. The dotted lines represent lesson change. One rhombus-shaped icon mark step where mean reward value reaches the threshold of +0,75.*

The results, as presented on Figure 4.6, clearly show that the non-CL-Agent (red), which was presented with the target task from the beginning of the training process, was not able to learn an optimal policy. Furthermore, it showed no potential in solving the problem within the given amount of time using sparse reward signal as it never managed to reach any state that would yield a positive reward. On the other hand, the CL-Agent (blue), which was trained with curriculum learning approach, not only managed to solve all the lessons of the designed curriculum, but also solve the target task. As shown on Figure 4.6, the first 4 lessons were solved within, approximately, 20 minutes of training, or 80.000 steps. The mean reward threshold of 0,75 on the target task was reached in roughly 32 minutes, or 120.000 steps. Moreover, the CL-Agent kept improving the learned policy by gradually increasing the mean cumulative reward throughout the rest of the training process. The highest mean accumulated reward of 0,87 was reached using curriculum learning approach.

### 4.2.3 Distance-based reward

In the second stage, the Agents were trained using distance-based reward signal, which was previously described in Section 3.3.4. The reward signal setup was the following:

- +0,1 - during each state, when the distance between the car and the finish line is decreased by an amount, which equals to 10% of the initial distance. Due to this percentage, this reward signal can only be received ten times during one local episode.
- 0 - for being in any other state of the learning environment.

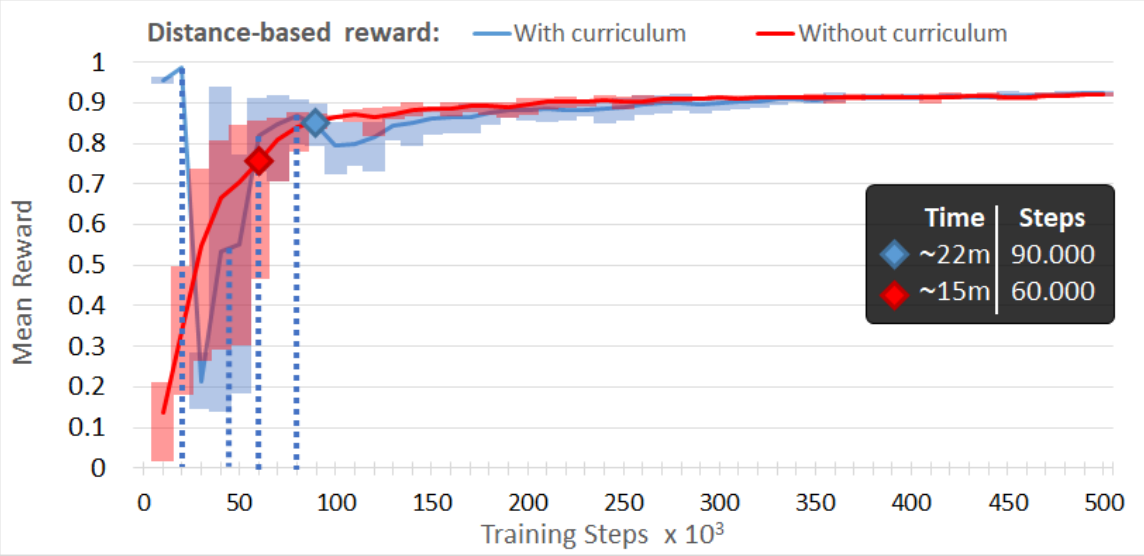



***Figure 4.7**: Mean cumulative reward results of Agents trained using distance-based reward signal, with and without curriculum. The results of the two training scenarios are based on five separate training sessions. The lines show average values, while the bars correspond to the lowest and highest values acquired at the end of the global episode. The dotted lines represent lesson change. Two rhombus-shaped icons mark step where mean reward value reaches the threshold of +0,75.*

As shown on Figure 4.7, the overall results of the CL-Agent (blue) demonstrate a similar pattern in respect to the CL-Agent, which was trained with sparse reward signal, presented in the previous Section. The lessons 1-4 were also solved and the final lesson, or the target task, was reached in around 20 minutes, or under 80.000 steps. As the mean cumulative reward was above the defined threshold after the CL-Agent switched to the target task, it can be considered as solved at the end of the subsequent global episode, or at 90.000 steps. Overall, the CL-Agent managed to solve the target task faster using distance-based reward signal. On the other hand, the non-CL-Agent (red) showed a rapid growth of the mean accumulated reward from the very beginning of the training. Most importantly, it reached the mean reward threshold of 0,75 in just 15 minutes, or 60.000 steps. Which is around ⅔ of both, time and steps, compared to the

CL-Agent. Moreover, the CL-Agent experienced a slight drop of mean cumulative reward after 80.000 steps when it switched to the target task. The value dropped from 0,87 to 0,80. Although, CL-Agent was able to recover and kept gradually increasing the mean accumulated reward after 100.000 steps, this drop created a significant gap between it and the non-CL-Agent. This gap disappeared only after 330.000 steps. At the end of the training, both CL-Agent and non-CL-Agent reached the same highest mean accumulated reward of 0,93.

### 4.2.4 Step reward

In the third stage, the Agents were trained again in the same two scenarios as presented above. However, in this stage, a small negative step reward has been added as a secondary reward signal. The goal here is to investigate how an additional reward signal can impact the training process as well as the developed behavior. Firstly, the Agents were trained with the following setup:

- +1 - for reaching a set of states where the car is located behind the finish line.
- -0,0001 - for being in any other state of the learning environment.

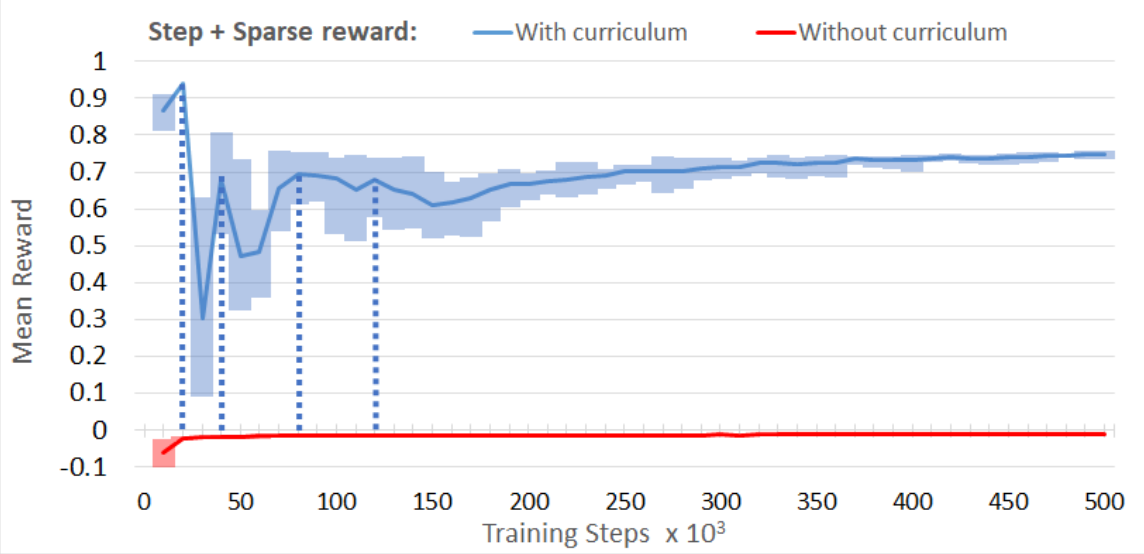



***Figure 4.8**: Mean cumulative reward results of Agents trained using step and sparse reward signals, with and without curriculum. The results of the two training scenarios are based on five separate training sessions. The lines show average values, while the bars correspond to the lowest and highest values acquired at the end of the global episode. The dotted lines represent lesson change.*

As presented on Figure 4.8, the results of both, the CL-Agent (blue) and the non-CL-Agent (red), are similar to the results of the Agents trained with sparse reward signal previously discussed in Section 4.2.2. Nevertheless, due to the additional negative reward signal, the mean accumulated reward is significantly lower. The CL-Agent still was able to reach the reward threshold of 0,75

in all the curriculum lessons. However, this has significantly prolonged the training time needed to switch to the subsequent lessons. Moreover, this threshold was not reached in the target task. This is due to the fact that the Agents need at least 1.150 steps to reach the goal of the target task. Thus, a negative cumulative reward of around -0,115 is acquired in the process. Furthermore, the non-CL-Agent never managed to reach the finish line, thus, never received a positive reward signal. Moreover, it learned to minimize the negative step reward by colliding with the closest obstacle in the training area, thus, reaching the terminal state. In this scenario, the highest mean accumulated reward of 0,74 was reached using curriculum learning approach.

Secondly, the Agents were trained with negative step and distance-based reward signals. Thus, the following setup was used:

- +0,1 - during each state, when the distance between the car and the finish line is decreased by an amount, which equals to 10% of the initial distance.
- -0,0001 - for being in any other state of the learning environment.

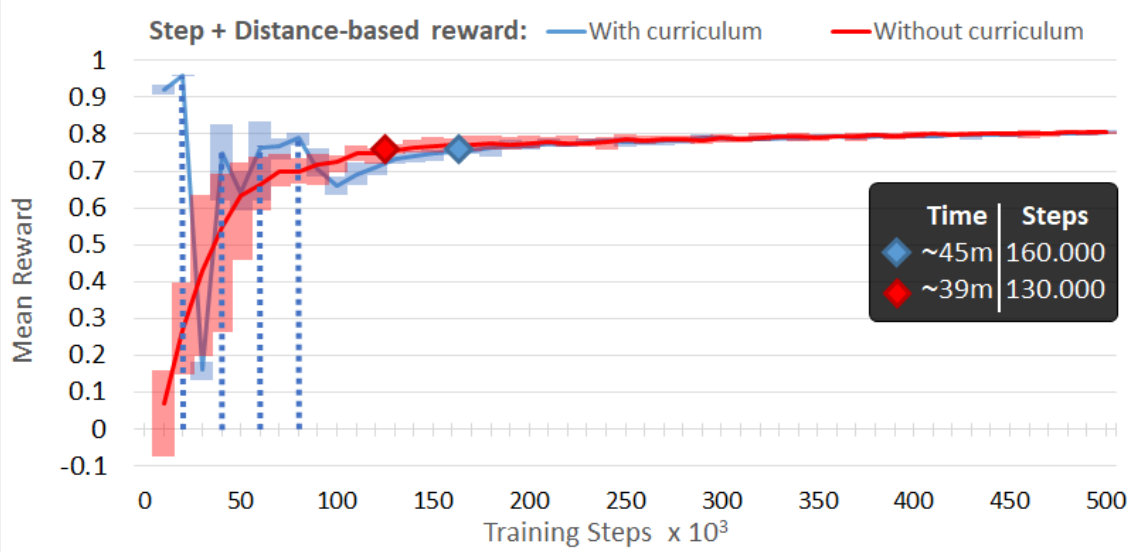



***Figure 4.9**: Mean cumulative reward results of Agents trained using step and distance-based reward signals, with and without curriculum. The results of the two training scenarios are based on five separate training sessions. The lines show average values, while the bars correspond to the lowest and highest values acquired at the end of the global episode. The dotted lines represent lesson change. Two rhombus-shaped icons mark step where mean reward value reaches the threshold of +0,75.*

As expected, the results, as shown on Figure 4.9, have a considerable resemblance to the training scenario where only distance-based reward signal was used, as described in Section 4.2.3. The main difference between the two scenarios is that, in this case, it took both Agents significantly longer to reach the defined mean reward threshold on the target task in terms of

time and training steps. However, it is, again, the impact of the negative step reward signal. Both, CL-Agent and non-CL-Agent, reached the same highest mean accumulated reward of 0,80.

### 4.2.5 Agent comparison

As all the Agents were trained under different conditions, the acquired results cannot be used to compare their performance. Nevertheless, it is possible to evaluate them based on their ability to solve the target task. As mentioned in Section 4.2.1, for each training scenario, the Agents were trained five times in the separate sessions. For this comparison, the Agents were tested using learned policies, which were selected from the sessions with the highest mean cumulative reward acquired towards the end of the training process.

As presented in the Table 4.2, the highest overall success rate, 99% to be specific, was reached by the Agent, which was trained using sparse reward signal with curriculum learning approach. The success rate of 98% was reached by the Agent, which was trained under the same conditions in addition to the negative step reward signal. Furthermore, the Agents trained using distance-based reward signal, without and with additional negative step-based reward, reached success rates of 97% and 94% respectively. More importantly, that the same percentage was achieved when trained with and without curriculum learning.

| Training scenario | Failed episodes (out of 1.000) | Success Rate (%) |
|---|---|---|
| Sparse reward with Curriculum | 14 | 99% |
| Sparse reward without Curriculum | 1.000 | 0% |
| Distance-based reward with Curriculum | 26 | 97% |
| Distance-based reward without Curriculum | 32 | 97% |
| Step + Sparse reward with Curriculum | 17 | 98% |
| Step + Sparse reward without Curriculum | 1.000 | 0% |
| Step + Distance-based reward with Curriculum | 60 | 94% |
| Step + Distance-based reward without Curriculum | 61 | 94% |

***Table 4.2**: Comparison of different training scenarios in respect to the success rate achieved on the target task, based on the best Agents performance.*

# V. Discussion and conclusion

This chapter is a conclusive discussion of the conducted experiments and their results. Overarching ideas and main observations are introduced as well as the goal of this thesis is addressed and briefly assessed.

All agents, presented in this thesis, were trained using Proximal Policy Optimization method that has shown remarkable performance in the past (Heess et al., 2017; Schulman et al., 2017). However, the reinforcement learning agent trained with this method could not solve the navigation task. In fact, this agent never managed to reach a state with a positive reward signal within the given time in five separate training sessions. Thus, due to the complexity of the task, the chances of solving it with a sparse reward signal are close to impossible. Potentially, the agent might reach a state with a positive reward based on random occurrences, however, these could be too sparse and have no effect on the training process (Sutton & Barto, 2018). On the contrary, the curriculum-based agent has managed to solve the same task using only a sparse reward signal. The constructed curriculum has introduced the agent to 4 lessons of increasing difficulty before it was presented with the target task. The first two lessons were designed to encourage navigation, while the two subsequent lessons introduce additional obstacles, thus, stimulating collision avoidance. Without any specific reward-based encouragement, the agent learned the following behavior: to keep a stable and consistent trajectory; to make a full U-turn; to reverse from a position, where the path ends due to the surrounding obstructions. Furthermore, the curriculum-based agent managed to reach a successful task completion rate of 99% in the final part of the experiment.

The combined application of reward shaping and curriculum learning has also been investigated. In the second training scenario, where distance-based reward signal was used instead of the sparse one, both agents managed to learn an optimal policy. In this case, the application of the curriculum learning approach was unnecessary as the reward signal clearly defines the goal of the target task and the expected behavior - the accumulated reward increases by reducing the distance to the finish line. Thus, the states that yield a positive reward can be reached relatively early in the training process. Furthermore, as one of the agents had to solve four other tasks, or lessons, until it was presented with the target task, it took a longer time for it to reach the defined threshold in comparison to the agent, which was trained without the curriculum. Although, the typical reinforcement learning agent did solve the target task using distance-based reward signal, it only managed to reach the success rate of 97%, thus, falling behind the curriculum-based agent trained with sparse reward signal.

The additional step reward signal had no positive impact on the trained agents. As expected, the overall results of the mean cumulative reward were lower due to the fact the step reward

was set as a negative value. Moreover, the final test have shown a 1-3% decrease in the agents' ability to successfully complete the target task compared to the agents that were trained without the additional step reward. When trained only with sparse and step reward signals, one agent developed an unwanted behavior of navigating towards the closest obstacle at the beginning of each episode. This is a result of agents inability to reach any state that yields positive reward, thus, it learned that immediate episode termination minimizes the accumulated negative reward. In this specific case, the additional reward signal has resulted in a negative side effect, that was described by Amodei et al. (2016).

As discussed in Section 4.2.3, it was concluded that curriculum learning can be used in combination with reward shaping, however, this method might not be beneficial to the training process if the two approaches are overlapping. The application of the curriculum learning approach can significantly prolong the training process if the reward function clearly defines the goal of the problem. For instance, in one of the training scenarios, the distance-based reward was used to encourage navigation towards the finish line, while the first and the second lessons of the curriculum were designed specifically for the same purpose. Thus, training the agent on these two curriculum lessons could be excessive. Moreover, as described in Section 4.2.4, the additional negative step reward signal has also affected the training process by significantly delaying the curriculum lesson change. The results revealed that this happened due to the fact that the defined mean cumulative reward threshold was too high. Therefore, not only the lesson structure, but also curriculum parameters have to be constantly adjusted based on the changes done to the reward function.

To sum up, it was shown that a state of the art algorithm failed to train the agent that would be able to solve the presented navigation task in the simulated environment without a complex, goal defining, reward function. However, the application of the environment-centered curriculum learning approach has shown how this task can be solved using only a sparse reward signal. Finally, it was observed that application of curriculum learning in combination with reward shaping can negatively affect the training process if the two approaches are overlapping by encouraging the same type of behavior.

# VI. Future work

The navigation task, presented in this thesis, as well as the curriculum constructed specifically to solve it could be extended in multiple directions. First of all, the complexity level of the task is determined by the distance that have to be covered by the agent and by the amount and placement of the obstacles. However, similarly to the approach taken by Heess et al. (2017), additional types of obstacles could be introduced to make the training area more complex as well as diverse. These could be: narrow obstacles (lamp posts, stationary pedestrians), moving obstacles of different sizes and shapes (other vehicles, pedestrians), ramps (bridges, road inclines) and unconventionally-shaped obstacles (parked vehicles, rock formations). Moreover, the track area could also be randomly generated to contain uneven terrain. These examples would also require the car to have additional multiple layers of sensors to register inclines and declines of the track.

Secondly, depending on the problem in question, manual construction of separate curriculum lessons could become as tedious and time consuming as designing a goal-oriented reward function. Therefore, as described previously in Section 2.3.2, one solution is to generate the curriculum lessons based on the agent's performance (Florensa et al., 2017). In respect to the same navigation task, the agent could be placed further away from its destination with each consecutive lesson, while the distance could be calculated based on the mean reward accumulated within the last global episode.

Finally, several examples have been mentioned previously in the Section 1.1, where the policies learned entirely in the simulated environments were successfully transferred to the real-world robots without the need for further training. Thus, it would be interesting to see, whether a policy learned by curriculum-based agent, trained in a diverse simulated environment with sparse reward signal, could be transferred to a real-world robot to solve a complex navigation task.

# Bibliography

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., & Mané, D. (2016). Concrete Problems in AI Safety. *ArXiv:1606.06565 [Cs]*. Retrieved from http://arxiv.org/abs/1606.06565

Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A Brief Survey of Deep Reinforcement Learning. *IEEE Signal Processing Magazine*, *34*(6), 26–38. https://doi.org/10.1109/MSP.2017.2743240

Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning (pp. 1–8). ACM Press. https://doi.org/10.1145/1553374.1553380

Florensa, C., Held, D., Wulfmeier, M., Zhang, M., & Abbeel, P. (2017). Reverse Curriculum Generation for Reinforcement Learning. *ArXiv:1707.05300 [Cs]*. Retrieved from http://arxiv.org/abs/1707.05300

Fu, J., Luo, K., & Levine, S. (2017). Learning Robust Rewards with Adversarial Inverse Reinforcement Learning. *ArXiv:1710.11248 [Cs]*. Retrieved from http://arxiv.org/abs/1710.11248

Gullapalli, V., & Barto, A. G. (1992). Shaping as a method for accelerating reinforcement learning (pp. 554–559). IEEE. https://doi.org/10.1109/ISIC.1992.225046

Heess, N., TB, D., Sriram, S., Lemmon, J., Merel, J., Wayne, G., … Silver, D. (2017). Emergence of Locomotion Behaviours in Rich Environments. *ArXiv:1707.02286 [Cs]*. Retrieved from http://arxiv.org/abs/1707.02286

Intel. (2018, August 28). Intel Product Specifications. Retrieved August 27, 2018, from https://ark.intel.com/products/88967/Intel-Core-i7-6700HQ-Processor-6M-Cache-up-to-3_50-GHz

Juliani, A. (2017a, September 19). Introducing: Unity Machine Learning Agents Toolkit – Unity

Blog. Retrieved July 14, 2018, from https://blogs.unity3d.com/2017/09/19/introducing-unity-machine-learning-agents/

Juliani, A. (2017b, December 8). Introducing ML-Agents Toolkit v0.2: Curriculum Learning, new environments, and more – Unity Blog. Retrieved July 14, 2018, from https://blogs.unity3d.com/2017/12/08/introducing-ml-agents-v0-2-curriculum-learning-new-environments-and-more/

Juliani et al., A. (2018, July 14). Unity Machine Learning Agents Toolkit. Retrieved July 14, 2018, from https://github.com/Unity-Technologies/ml-agents

Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., … Shah, A. (2018). Learning to Drive in a Day. *ArXiv:1807.00412 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1807.00412

Kretzschmar, H., Spies, M., Sprunk, C., & Burgard, W. (2016). Socially compliant mobile robot navigation via inverse reinforcement learning. *The International Journal of Robotics Research*, *35*(11), 1289–1307. https://doi.org/10.1177/0278364915619772

Li, X., Ma, Y., & Belta, C. (2017). A Policy Search Method For Temporal Logic Specified Reinforcement Learning Tasks. *ArXiv:1709.09611 [Cs]*. Retrieved from http://arxiv.org/abs/1709.09611

Li, X., Vasile, C.-I., & Belta, C. (2016). Reinforcement Learning With Temporal Logic Rewards. *ArXiv:1612.03471 [Cs]*. Retrieved from http://arxiv.org/abs/1612.03471

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., … Wierstra, D. (2015). Continuous control with deep reinforcement learning. *ArXiv:1509.02971 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1509.02971

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., … Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. *ArXiv:1602.01783 [Cs]*. Retrieved from http://arxiv.org/abs/1602.01783

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *ArXiv:1312.5602 [Cs]*. Retrieved from http://arxiv.org/abs/1312.5602

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., … Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. https://doi.org/10.1038/nature14236

Ng, A. Y., & Russell, S. J. (2000). Algorithms for Inverse Reinforcement Learning. In *Proceedings of the Seventeenth International Conference on Machine Learning* (pp. 663–670). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. Retrieved from http://dl.acm.org/citation.cfm?id=645529.657801

NVIDIA. (2018, March 27). NVIDIA Introduces DRIVE Constellation Simulation System to Safely Drive Autonomous Vehicles Billions of Miles in Virtual Reality. Retrieved August 27, 2018, from http://nvidianews.nvidia.com/news/nvidia-introduces-drive-constellation-simulation-system-to-safely-drive-autonomous-vehicles-billions-of-miles-in-virtual-reality

OpenAI. (2018). Learning Dexterous In-Hand Manipulation. *ArXiv:1808.00177 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1808.00177

Popov, I., Heess, N., Lillicrap, T., Hafner, R., Barth-Maron, G., Vecerik, M., … Riedmiller, M. (2017). Data-efficient Deep Reinforcement Learning for Dexterous Manipulation. *ArXiv:1704.03073 [Cs]*. Retrieved from http://arxiv.org/abs/1704.03073

Randløv, J., & Alstrøm, P. (1998). Learning to Drive a Bicycle Using Reinforcement Learning and Shaping. In *Proceedings of the 15th International Conference on Machine Learning* (pp. 463–471).

Sadeghi, F., & Levine, S. (2016). CAD2RL: Real Single-Image Flight without a Single Real Image. *ArXiv:1611.04201 [Cs]*. Retrieved from http://arxiv.org/abs/1611.04201

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2015). Trust Region Policy Optimization. *ArXiv:1502.05477 [Cs]*. Retrieved from http://arxiv.org/abs/1502.05477

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *ArXiv:1707.06347 [Cs]*. Retrieved from http://arxiv.org/abs/1707.06347

Sermanet, P., Xu, K., & Levine, S. (2016). Unsupervised Perceptual Rewards for Imitation Learning. *ArXiv:1612.06699 [Cs]*. Retrieved from http://arxiv.org/abs/1612.06699

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., … Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. https://doi.org/10.1038/nature16961

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning*: *an introduction* (Second edition). Cambridge, MA: The MIT Press.

Taylor, M. E., & Stone, P. (2009). Transfer Learning for Reinforcement Learning Domains: A Survey. *J. Mach. Learn. Res.*, *10*, 1633–1685.

Unity Technologies. (2018a). Unity. Retrieved August 16, 2018, from https://unity3d.com/unity

Unity Technologies. (2018b, July 31). Unity - Manual: Wheel Collider. Retrieved August 16, 2018, from https://docs.unity3d.com/Manual/class-WheelCollider.html

Waymo. (2018, August 27). On the Road. Retrieved August 27, 2018, from https://waymo.com/ontheroad/

Y. Ng, A., Harada, D., & Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping.

Yu, A., Palefsky-Smith, R., & Bedi, R. (2016). Deep Reinforcement Learning for Simulated Autonomous Vehicle Control, 7.

Yu, W., Turk, G., & Liu, C. K. (2018). Learning Symmetric and Low-energy Locomotion. *ArXiv:1801.08093 [Cs]*. https://doi.org/10.1145/3197517.3201397

Zhu, Y., Wang, Z., Merel, J., Rusu, A., Erez, T., Cabi, S., … Heess, N. (2018). Reinforcement and Imitation Learning for Diverse Visuomotor Skills. *ArXiv:1802.09564 [Cs]*. Retrieved from http://arxiv.org/abs/1802.09564

# Appendices

## A - Curriculum_final script

The curriculum lesson number is used to reposition the finish line, set specific parameters and activate specific objects in the simulated environment. This is done using a switch statement within GenerateCurriculum() method. ResetCurriculum() method is used to disable all the objects that are used in curriculum lessons.

```
public class Curriculum_final : MonoBehaviour {

    public CarAgent_final carAgent;
    public GameObject finishLine;
    public GameObject[] walls = new GameObject[0];
    public GameObject[] obstacleAreas = new GameObject[0];

    public void GenerateCurriculum(int level)
    {
        ResetCurriculum();

        switch (level)
        {
            case 0:
                finishLine.transform.localPosition = new Vector3(0f, 0.001f, 15f);
                break;
            case 1:
                carAgent.minRotationAngle = 0f;
                carAgent.maxRotationAngle = 360f;
                carAgent.randomRotation = true;
                finishLine.transform.localPosition = new Vector3(0f, 0.001f, 47f);
                break;
            case 2:
                carAgent.minRotationAngle = 0f;
                carAgent.maxRotationAngle = 360f;
                carAgent.randomRotation = true;
                obstacleAreas[0].SetActive(true);
                finishLine.transform.localPosition = new Vector3(0f, 0.001f, 47f);
                break;
            case 3:
                carAgent.minRotationAngle = 0f;
                carAgent.maxRotationAngle = 360f;
                carAgent.randomRotation = true;
                obstacleAreas[1].SetActive(true);
                finishLine.transform.localPosition = new Vector3(0f, 0.001f, 47f);
                break;
            case 4:
                carAgent.minRotationAngle = 0f;
                carAgent.maxRotationAngle = 360f;
                carAgent.randomRotation = true;
                obstacleAreas[2].SetActive(true);
                finishLine.transform.localPosition = new Vector3(0f, 0.001f, 47f);
                break;
        }
    }
```

## B - CarAgent_final script

### B.1 - Observation space

The observation space is defined within CollectObservations() method. Specific features are calculated and added to a single feature vector through AddVectorObs() method.

```
103     public override void CollectObservations()
104     {
105         // Relative position towards target
106         relativePosition = finishLine.transform.position - this.transform.position;
107         relPos = new Vector2(relativePosition.x, relativePosition.z);
108         AddVectorObs(relPos);
109 
110         // Velocity of the car
111         vel = new Vector2(rb.velocity.x, rb.velocity.z);
112         AddVectorObs(vel);
113 
114         // Collision sensors
115         sensorData = checkSensors();
116         AddVectorObs(sensorData);
117     }
```

## B.2 - Action space

The Agent uses discrete action space, which is defined with a switch statement. After an action is chosen, the moveCar() method is called, which sets the new steering angle and motor torque.

```
public void MoveAgent(float[] action)
{
    if (brain.brainParameters.vectorActionSpaceType == SpaceType.continuous)
    {
        actTorque = Mathf.Clamp(action[0], -1f, 1f);
        motorTorque = (actTorque > 0) ? maxMotorTorque * actTorque : (maxMotorTorque / 2) * actTorque;
        steeringAngle = maxSteeringAngle * Mathf.Clamp(action[1], -1f, 1f);
    }
    else
    {
        int act = Mathf.FloorToInt(action[0]);

        switch (act)
        {
                case 0:
                    motorTorque = maxMotorTorque / 2;
                    break;
                case 1:
                    motorTorque = maxMotorTorque;
                    break;
                case 2:
                    motorTorque = -maxMotorTorque / 2;
                    break;
                case 3:
                    steeringAngle = maxSteeringAngle;
                    break;
                case 4:
                    steeringAngle = maxSteeringAngle / 2;
                    break;
                case 5:
                    steeringAngle = -maxSteeringAngle;
                    break;
                case 6:
                    steeringAngle = -maxSteeringAngle / 2;
                    break;
                case 7:
                    steeringAngle = 0;
                    break;
        }

    }
    moveCar(motorTorque, steeringAngle);
}

public void moveCar(float torqueAmount, float steeringAmount)
{
    foreach (AxleInfo axleInfo in axleInfos)
    {
        if (axleInfo.steering)
        {
            axleInfo.leftWheel.steerAngle = steeringAmount;
            axleInfo.rightWheel.steerAngle = steeringAmount;
        }

        if (axleInfo.motor)
        {
            axleInfo.leftWheel.motorTorque = torqueAmount;
            axleInfo.rightWheel.motorTorque = torqueAmount;
        }
    }
}
```

## B.3 - Reward signal

At each simulation step the Agent calculates state reward through CalculateReward() method. Here, 3 Booleans define which reward signals have to be checked. These Booleans as well as the reward values are set by the user. Moreover, the Agent's position, relative to the finish line, is checked in each case in order to catch the terminal state.

```
119     public void CalculateReward()
120     {
121         if (sparse)
122         {
123             if (this.gameObject.transform.position.z > finishLine.transform.position.z)
124             {
125                 //AddReward(MathF.Min(2 * MathF.Abs(if - ((float)GetStepCount() / (float)agentParameters.maxStep)), 1f));
126                 SetReward(sparseReward);
127                 Done();
128             }
129         }
130 
131         if (eachStep)
132         {
133             AddReward(stepReward);
134 
135             if (this.gameObject.transform.position.z > finishLine.transform.position.z)
136             {
137                 Done();
138             }
139         }
140 
141         if (overDistance)
142         {
143             if (this.transform.localPosition.z > distanceTarget)
144             {
145                 AddReward(distanceReward);
146                 distanceTarget += distanceGap;
147 
148                 if (this.gameObject.transform.position.z > finishLine.transform.position.z)
149                 {
150                     Done();
151                 }
152             }
153         }
154     }
```

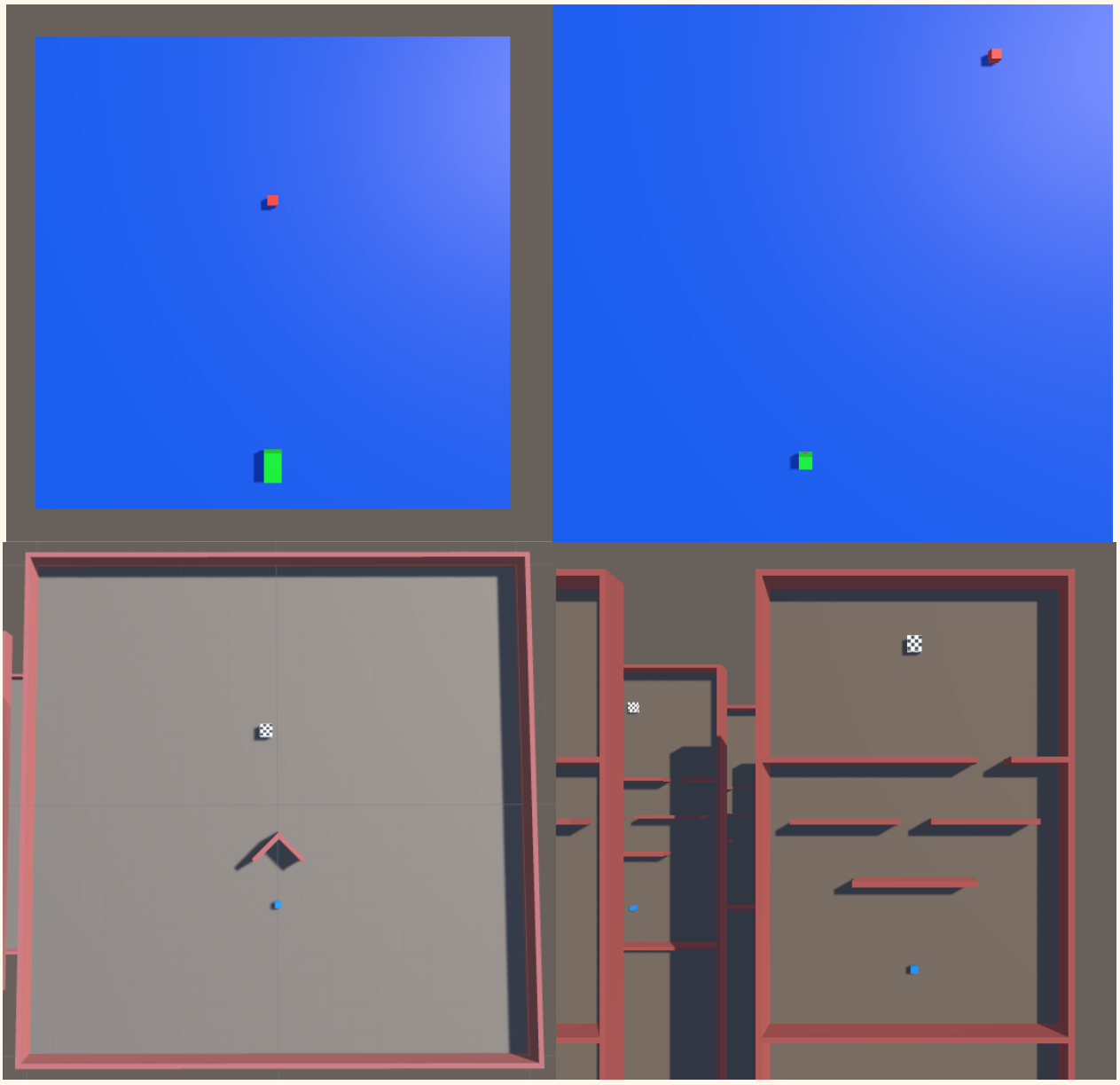
## C - ObstacleGridGenerator script

The obstacle grid is generated in rows defined by *width*, while the number of columns is defined by *height*. In each row, a random number of obstacles is removed, based on the defined *minimum* and *maximum number of gaps*. When the grid is initiated, each position is checked whether an obstacle has to be placed inside.

```
37      void SpawnObstacles()
38      {
39          for (int y = 0; y < height; y++)
40          {
41              List<int> newRow = RandomizedRow(width, minGaps, maxGaps);
42
43              for (int x = 0; x < width; x++)
44              {
45                  if (newRow.Contains(x))
46                  {
47                      GameObject obs = Instantiate(obstacle);
48                      obstacles.Add(obs);
49                      obs.transform.parent = this.transform;
50                      obs.transform.localPosition = new Vector3(x * obs.transform.localScale.x,
51                                                                0.5f,
52                                                                y * (heightGap + obs.transform.localScale.z));
53                  }
54              }
55          }
56      }
57
58      List<int> RandomizedRow(int w, int min, int max)
59      {
60          List<int> row = new List<int>();
61
62          for (int i = 0; i < w; i++)
63          {
64              row.Add(i);
65          }
66
67          for (int r = 0; r < Random.Range(min, max + 1); r++)
68          {
69              row.RemoveAt(Random.Range(0, row.Count));
70          }
71
72          return row;
73      }
74
```

## D - Initial implementations

Initially, the most simplistic environments were built to start testing different action and observation spaces, multi-agent setups as well as the ML-Agent toolkit itself. The initial challenge was to only train navigation. The Figures below represent four different learning environments that were tested before the final implementation was built.



# E - Curriculum iterations

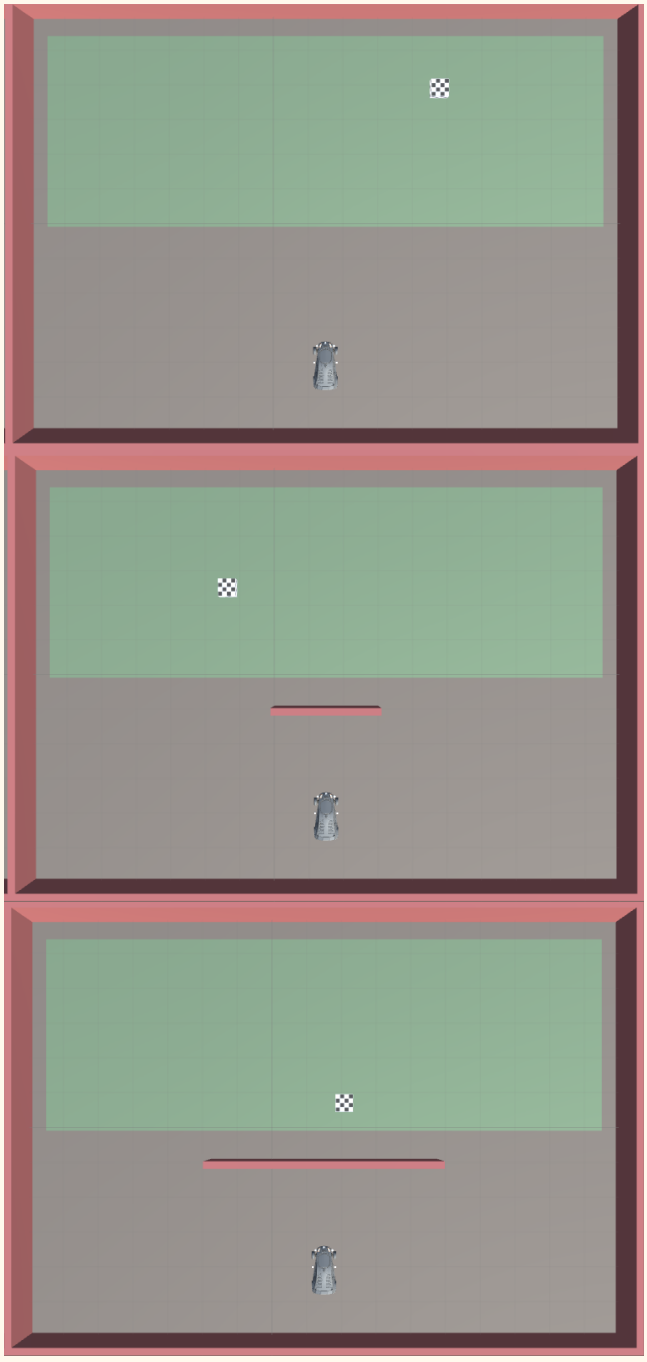
## E.1 - Target behind the wall

The first curriculum lesson structure. The agent's goal is to navigate towards a randomly positioned target behind a wall.

Lesson 1: wall is completely removed.

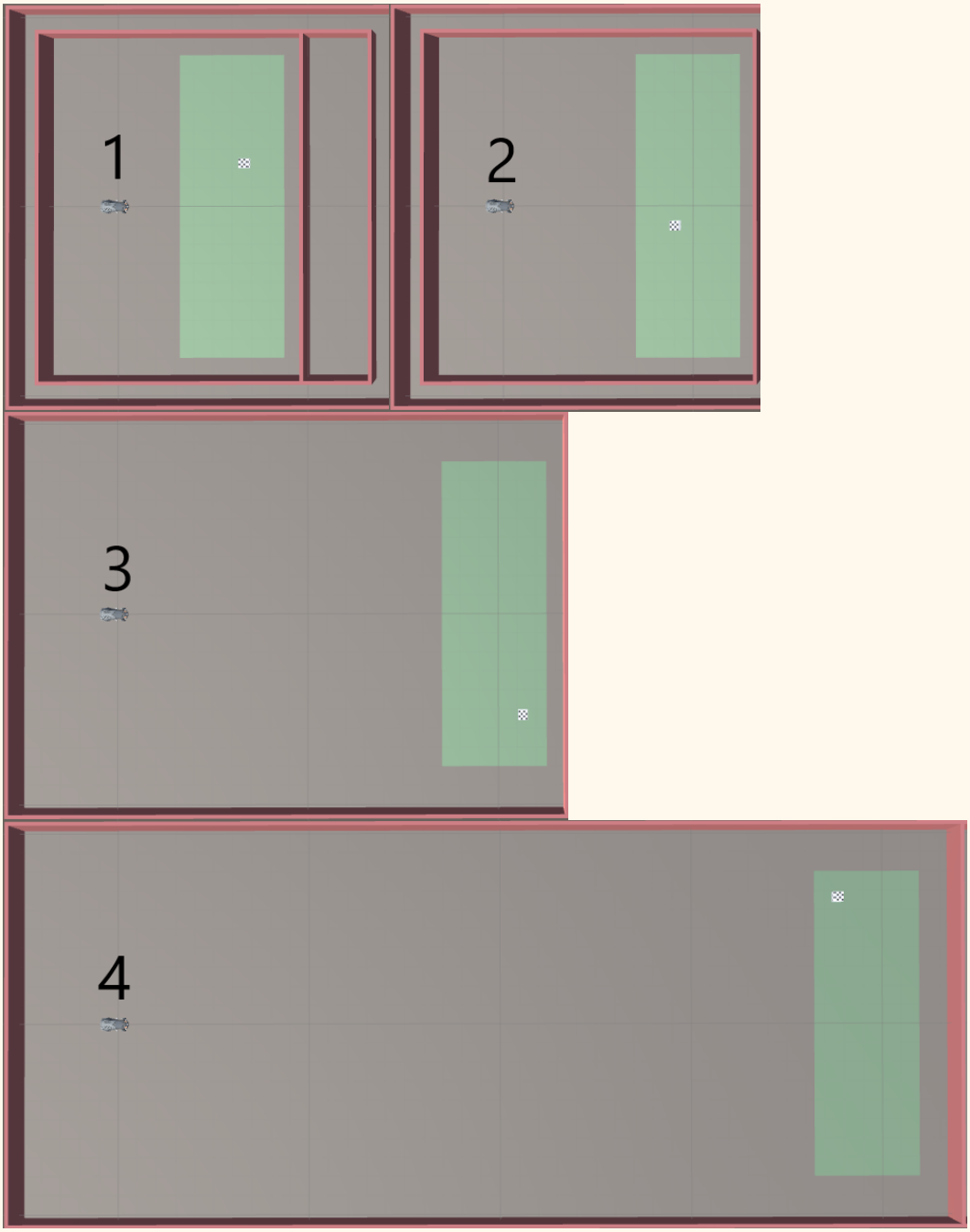
Lesson 2: part of the wall is introduced.

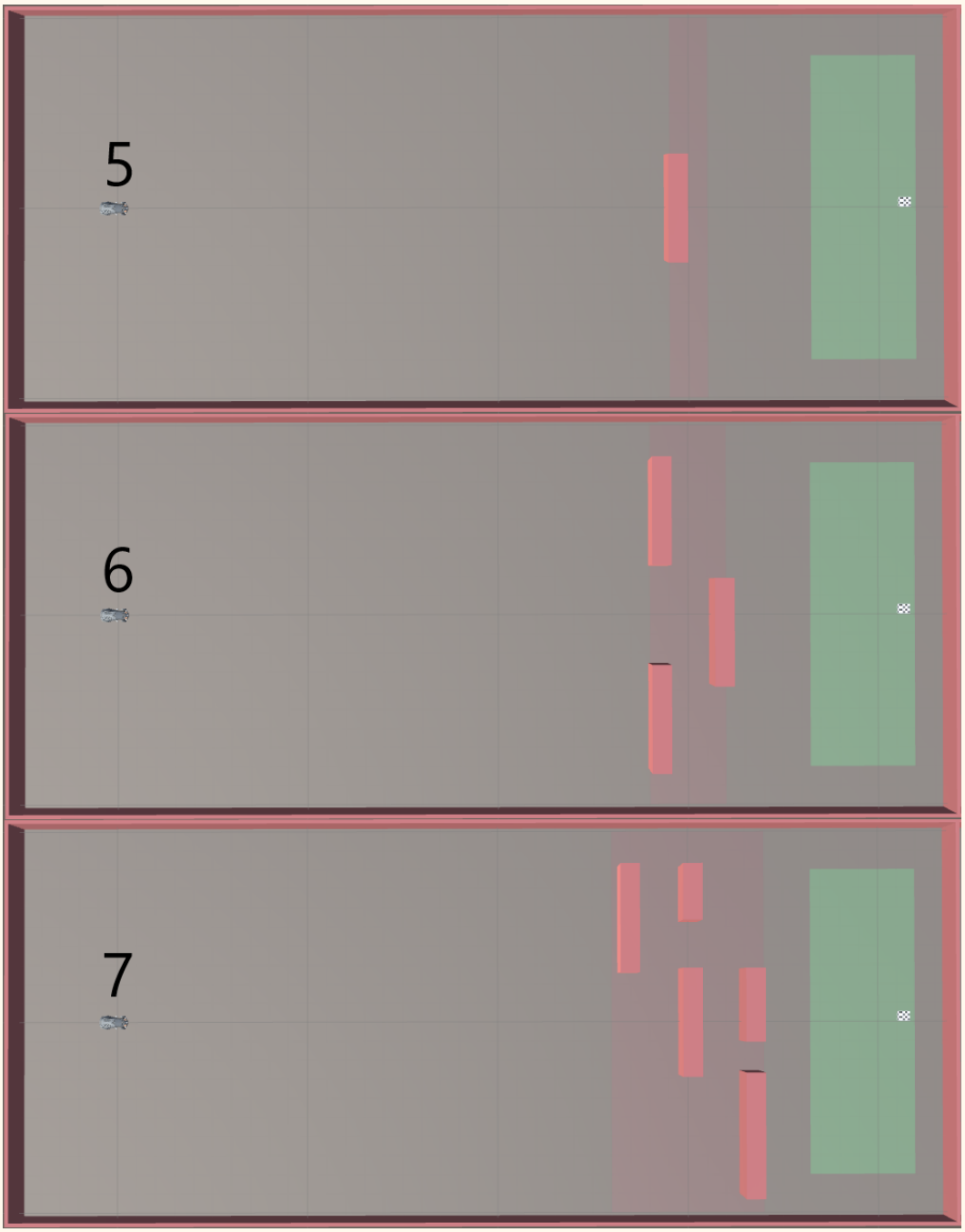
Lesson 3: full wall is introduced.



## E.2 - 7 lesson curriculum

The agent's goal is to navigate towards a randomly positioned target at the end of the training area. Lessons 1-4 do not contain additional obstacles, however, the target area is positioned further from the agent with each consecutive lesson. Lessons 5-7 introduce additional obstacles, which are randomly generated using Perlin noise.




5
6
7

## F - Car sensor setup

The preview of the 10 sensors positioning on the car that were used in the final experiment.