Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at
TU WIEN UB Universitätsbibliothek
The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

DIPLOMA THESIS

A Self Learning Embedded System

Submitted at the Faculty of Electrical Engineering and Information Technology, Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Sciences)

under supervision of

Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch
Projectass. Dipl.-Ing. Dr.techn. Alexander Wendt

by

David Bechtold
Matr.Nr. 01326422

David Bechtold

Vienna, April 4, 2019

Abstract

In recent years, interest in self-learning methods has increased significantly. A driving factor was the growing computing power, which first enabled machines to carry out such computationally intensive methods. Nowadays, machine learning is used in many areas, such as prediction, pattern recognition, and anomaly detection.

In this thesis, a self learning embedded system (SLES) is supposed to learn solving tasks completely independently and with as little prior knowledge about itself, the task and the environment, as possible. The learning process is guided by a reward signal, which punishes or rewards the performed actions. Our main focus is on the task of surviving as long as possible. For this purpose charging stations must be located. Subsequently, they must be approached properly to allow a successful charging of the battery.

In order to enable the independent learning of tasks, various methods from the field of Reinforcement Learning (RL), in particular from Q-learning, are used. In addition, several replay memories and exploration methods are implemented and modified. Further, completely new approaches and ideas are realized with the aim of achieving better results. Evaluations help to find the most appropriate methods for our problem. Finally, they are tested in a simulation environment to ensure that they can be applied to the final hardware without significant changes.

With the help of evaluations and simulations, this work shows the entire process, from the selection of the methods to the determination of which process parameters to use for the final SLES. In the future, it is planned to improve the selected methods, to reduce the memory requirements and to handle other tasks besides survival. In addition, the simulation model should be improved to be even closer to the real model.

Kurzfassung

In den letzten Jahren hat das Interesse an selbstlernenden Methoden deutlich zugenommen. Ein treibender Faktor war die wachsende Rechenleistung, die es Maschinen erst ermöglichte, solche rechenintensiven Verfahren durchzuführen. Heutzutage wird maschinelles Lernen in vielen Bereichen, wie zum Beispiel für sämtliche Vorhersagen, Mustererkennungen und Anomalieerkennungen, eingesetzt.

In dieser Arbeit soll ein selbstlernendes Embedded System (SLES) das Lösen von Aufgaben völlig selbstständig und mit möglichst wenig Vorwissen über sich selbst, die Aufgabe und die Umgebung, erlernen. Der Lernprozess wird von einem Belohnungssignal gesteuert, das die ausgeführten Aktionen bestraft oder belohnt. Unser Hauptaugenmerk liegt auf der Aufgabe, so lange wie möglich zu überleben. Zu diesem Zweck müssen Ladestationen geortet werden. Diese müssen anschließend korrekt angefahren werden, um ein erfolgreiches Aufladen der Batterie zu ermöglichen.

Um das selbstständige Erlernen von Aufgaben zu ermöglichen, werden verschiedene Methoden aus dem Bereich des Reinforcement Learning (RL), insbesondere aus dem Q-learning, eingesetzt. Darüber hinaus werden mehrere Wiedergabespeicher - und Erkundungsmethoden implementiert und modifiziert. Weiteres werden völlig neue Ansätze und Ideen, mit dem Ziel bessere Ergebnisse zu erreichen, realisiert. Mittels Evaluierungen werden die am besten geeignetsten Methoden für unser Problem gefunden. Schlussendlich werden diese in einer Simulationsumgebung getestet, um sicherzustellen, dass sie ohne erhebliche Änderungen auf die endgültige Hardware angewendet werden können.

Mit Hilfe von Auswertungen und Simulationen zeigt diese Arbeit den gesamten Prozess von der Auswahl der Methoden bis hin zur Bestimmung, welche Prozessparameter für das endgültige SLES verwendet werden sollen. In Zukunft ist geplant, die ausgewählten Methoden zu verbessern, den Speicherbedarf zu reduzieren und neben dem Überleben andere Aufgaben zu bewältigen. Außerdem sollte das Simulationsmodell verbessert werden, um dem realen Modell noch näher zu sein.

Table of Contents

1 Introduction

The interest in machine learning research has exploded in recent years. Nowadays, it helps us to accomplish tasks that could not be implemented from scratch because of the immense state space. These methods learn from experiences like humans do, but compared to us, much more additional experiences are necessary to learn a task. Humans immediately recognize the impact their actions have on the environment, but machines lack that understanding. For a machine, moving to the left is just a number, without realizing the effects of this action on the environment. However, due to the tremendous state space, the function to be learned is usually approximated. Among researchers, it has been established to use neural networks as these approximators. These networks, modeled after the human brain, are nonlinear and resource intensive. Although, the idea of neural networks exists since 1943 [MP43, pp. 1-21], it could not be implemented until several years ago, due to the lack of computation power [PG17, p. 1]. In addition, neural networks were completely underestimated because of the simple idea behind them. However, it has been found that these networks are very flexible and provide unbelievably good results, as can be seen in the following examples: Neural networks can be trained to understand the textual context with near-human accuracy, such as Facebook's DeepText, which understands several thousand posts per second in more than 20 languages, [2] or for the early detection of lung cancer using digital X-ray, CT or MRI images [SLHV18, p. 1]. They were even successfully used to defeat a Go Grandmaster, where artificial intelligence has not had a chance so far due to the high complexity of Go [SSS+17, pp. 1-2]. Last but not least, neural networks are even deployed for the detection of depressions only on the basis of speech patterns, without the need for further information about the questions or answers [HGG18, p. 1].

In this thesis, a self learning embedded system (SLES), in the further work designated as agent, should learn to solve tasks completely independently and with as little prior knowledge as possible. This also includes knowledge about the task, the environment exposed and the meaning of the inputs or outputs of the agent. To interact with the environment and to perceive its state the agent is equipped with sensors and actuators. The learning process is only guided by a reward signal, which punishes or rewards the performed actions. One issue is, that only a partial part of the environmental state, due to the small number of sensors, is perceived. In addition, a continuous and real state sensed

by the sensors results in a large state space, that needs to be explored. Since effective exploration of large state spaces is difficult to execute and due to the poor resolution of the environmental state, only sparse rewards are obtained. The task, which the agent should solve first, is to survive as long as possible. This enables solving several other tasks.

In order to gain experience with self-learning methods and to be flexible about which method is used on the final hardware, several state of the art methods have been implemented. Additionally, changes to the state of the art methods and new approaches have been realized to further improve our knowledge and to find the most appropriate method for the agent. Various methods from the field of Reinforcement Learning (RL) are used, in particular from Q-learning (Deep Q-learning, Double Deep Q-learning, Deep Deterministic Policy Gradient). In order to work, these methods require a replay memory, an exploration method, and a reward function. Therefore, several replay memories (Experience Replay, Priority Replay, Hindsight Experience Replay), exploration methods (ϵ-greedy, Ornstein Uhlenbeck process) and reward functions (shaped and not shaped) have been implemented.

Q-learning methods must explore the environment and the task to be solved in order to gain knowledge about it. The explored knowledge is accumulated and an expected reward (Q-value) can be calculated. This Q-value indicates how much reward to expect for performing an action from a certain state. A neural network can be trained by comparing the predicted Q-values to the received reward. To find the most appropriate method for the agent, each of them is evaluated quantitatively against the others. Furthermore, to ensure that our chosen method is usable on the final hardware without major changes, the agent in combination with the selected method, is simulated with Unity [12], a state-of-the-art game engine.

With the help of evaluations and simulations, this work shows the entire process, from the selection of the methods to the determination of which process parameters to use for the final agent. In addition, the question of how to design a reward function and how different sensors can be simulated is answered.

In the future, the memory of the selected method should be drastically reduced, with the aim of fitting on a small memory of a resource-limited system. To speed up the simulation time, an asynchronous algorithm should be implemented that allows parallel execution of multiple agents. The Q-learning algorithm should be improved to better respond to sparse reward problems. Consideration should also be given to formulating a better exploration method since exploration is a key RL task. Improvements can reduce training time and dramatically improve learning success. Other tasks, beside surviving, should be defined and resolved. Finally, the simulation model should be improved to be even closer to the real agent.

1.1 Motivation

Some tasks can not be completely solved by algorithms because of their complexity and enormous state space. For example, grabbing objects with a robotic arm is a difficult task to program because of the myriad positions an object can take in space. Even for the automated digitization of handwritten texts, finding an universal algorithm is not easy, because each of us writes in a different way. Another example would be the recognition of chairs on pictures. Due to the large number of different-looking chairs, this seemingly simple task is very difficult to solve in the end for a high hit probability.

A good approach to tackle such issues is to let the decision maker learn to solve the task completely independently. For scalability and reusability, the method applied to the agent should not be tied to any assumptions or restriction, such as the design of the environment, the agent or the task to learn. Therefore, the agent should have as little prior knowledge about the task, the environment and the meaning of its in- or outputs. This ensures that this thesis can serve as a basis for other work. In order for a new task to be learned, only a reward function, which rewards the agent for performed actions, has to be designed. In our vision, workers will only need to define a reward function while the robot learns the task by itself. The time-consuming and error-prone writing of algorithms and programs is eliminated. In addition, massive amount of computations are cheaper than paying employees. This will dramatically reduce the project costs and result in higher production yield and throughput.

1.2 Problem

The main objective of this thesis is to address the question how an agent can learn to solve tasks completely independently. Therefore, the agent should know as little as possible about the task, the environment and the meaning of its in- or outputs. Through a reward signal, which is used to punish or reward performed actions, the agent can learn more about the task to be solved. In order to be able to perceive its surroundings, the agent is equipped with various sensors. Thus, only a partial part of the environment is perceived. On the one hand, this can affect the learning performance of tasks, because the unperceived states can change or be forgotten by the method used. On the other hand, designing of a motivating reward signal depends on the perceived state. Therefore, the design of the reward function can only be as good as the state perceived by the sensors used. In addition, with the equipped sensors, a continuous state of the environment is perceived. Consequently, this results in a large state space because real numbers are uncountable. Therefore, a function approximator should be used to predict what action should be taken from a particular state, as saving the action state pair would require a lot of memory. However, this large state space needs to be explored, in order to learn which actions the agent should perform to solve the task. Therefore, the learning success heavily depends on a good exploration of the state space. If the task goal is underexplored, it is unlikely that the used method is able to learn to solve the task. Since one requirement is that the agent should have as little

prior knowledge as possible, only random exploration methods can be used. Not only that random exploration of large state spaces is already difficult, storing already visited states is not applicable because memory is a limited resource in an embedded system. However, due to the large state space the rewarding goal is not often achieved which yields to a sparse reward problem. This means that many of the gathered experiences do not contribute significantly to task resolution. The applied method should be able to learn to solve the tasks, even if it is flooded with many non-task-relevant transitions. Finally, the applied method should be able to learn to solve the tasks in the best way possible. The following list summarizes the requirements that the learning method should meet:

- Have as **little knowledge** about itself, the task and the environment as possible.
- Work only with a **partial part of the environment perceived**.
- Use a **function approximator** to predict which action to perform due to the large state space.
- Require **less memory** because it should run on an embedded system.
- Handle **continuous state spaces**, because sensory outputs are continuous.
- Deal with **sparse rewards** because random exploration of large state spaces is difficult.
- Able to learn to solve the task in the **best way possible**.

1.3 Task

In general, the agent possesses several inputs from sensors for state observation and outputs to actuators for movement. Additionally, it is equipped with communication hardware and a battery for self-power. To determine the boundaries of the environment, for orientation and for state observation, two rangefinders and a compass are mounted on it. More on, it is equipped with two photodiodes to facilitate locating the charging stations as they emit ultraviolet light. Additional hardware has been installed to determine the battery charge status. To enable movements, two actuators are placed at the bottom of the agent. For communication with a base station, a Bluetooth chip is attached. The architecture of the agent can be observed in figure 1.1. In the first approaches the processor is only used for communication with a base station, due to the low processing power and memory of the mounted processor. Hence, the perceived state is processed in the base station. Then the control values for the actuators are calculated and returned to the agent. Gradually, in the future, attempts will be made to move more and more computations from the base station to agent's processor.

The reward or punishment signal, that is used to motivate learning, is calculated in the base station, based on agent's current state. First, the agent will explore its

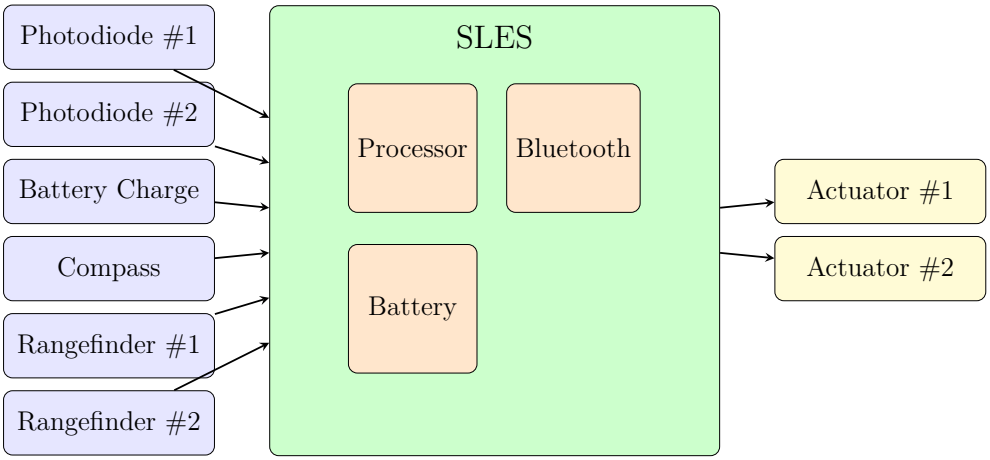


Figure 1.1: SLES Architecture.

surroundings, gather experiences and hopefully be rewarded for some actions. With these experiences, learning can be enabled and the agent can be taught which actions it has to perform in order to solve its task. Based on the knowledge it gained from its previous experiences, the agent learns to do his job better and better. The goal is to find the optimal strategy.

In order to enable learning multiple other tasks, the first task addressed within this thesis is to survive as long as possible. This can be achieved by locating the charging station and charge agent's battery. Since the real agent is not yet build, an attempt is made to solve this task by simulation. Therefore, the agent, the environment and the algorithm used are simulated as realistically as possible with Unity. After the task of survival has been successfully resolved, other, even more complex tasks can be tackled.

1.4 Methodology

The task is learned with machine learning methods. Machine learning can be considered as automated large scale data analysis [Bar12, p. 285] and allows learning of problems that can not be completely solved by algorithms due to their complexity. Machine learning can be applied to the following problems:

- Prediction
- Pattern recognition
- Anomaly detection

Since the agent has to predict which action to perform next in order to receive a high amount of reward, our issue can be considered as a *prediction* problem. Therefore, machine learning can be applied to our issue. However, since one goal is to learn to solve the

problem in the best possible way, the method used must be from the field of Reinforcement learning (RL) because the other fields are not applicable for this problem. Since Deep Q-learning (DQN) performed well on several Atari 2600 games (DQN outperformed a linear learning function in 43 out of 49 games and a professional human game tester in 29 out of 49 games [MKS+15, pp. 2-3]), algorithms from this area are selected. Q-learning calculates Q-values that show how much reward to expect by performing a particular action from a certain state. The *Deep* in Deep Q-learning means that a neural net is used as a function approximator. This neural net is used to predict the Q-values.

To familiarize ourselves with Deep Q-learning and to be flexible about which method is used on the final agent, several Q-learning algorithms are implemented such as:

- Deep Q-Network (DQN)
- Double Deep Q-Network (DDQN)
- Deep Deterministic Policy Gradient(DDPG)

DQN and DDQN can be used for discrete action spaces, while DDPG can be used for continuous action spaces. For example, if the movement of the agent is discretized to: forward, backward and turn; DQN or DDQN can be applied. On the other hand, if the raw continuous actuator values are used, DDPG can be applied.

Deep Q-learning requires other methods to work. First, a reward function is used to reward or punish the agent's actions. Without this function no learning can take place. To enable learning from past experiences and to guarantee the convergence of the Q-network a replay memory has to be used. Several replay memories are implemented in order to find the one which suits our needs best:

- Experience Replay (EXPR)
- Prioritized Experience Replay (PEXPR)
- Hindsight Experience Replay (HER)

By exploring, the agent can eventually gather information about the task and the environment unknown to him. This can be achieved by exploration methods. Without exploration, the agent learns little or nothing about the task and the environment. This drastically harms the tasks solving performance. On the other hand, only exploring the environment results in overfitting of the Q-network. In this case the Q-learning method is only able to solve a specific task and environment. Little changes to the environment causes the Q-learning method to fail in solving the task. Several exploration strategies are implemented in order to find the one which suits our needs best:

- ϵ-greedy (EG)
- Ornstein Uhlenbeck process (OU)

These methods and requirements mentioned are evaluated against each other in order to measure the performance. In addition, since the internal behaviour of neural networks is difficult to understand, the meta-parameters and methods are subjects to minor changes in order to study their impacts.

Furthermore, to make sure that our chosen Q-learning method and requirement will work and that they can be deployed on the final hardware without much change, they are simulated with Unity [12], a state-of-the-art game engine. To model neural nets we use Keras [5], a high-level neural network application programming interface (API), in combination with Tensorflow [9].

2 State of the Art

This chapter introduces the field of machine learning. First, a general overview of the styles of machine learning is offered. It is explained why only one particular style, namely Reinforcement Learning, can be considered for our work. Followed with important definitions and equations the reader is guided to Q-learning, a subfield of Reinforcement Learning. The most common and in this work used Q-learning algorithms are explained. Since Q-learning requires other methods such as exploration, reward functions, and transition storage, a detailed explanation of these techniques is provided. Eventually, related work will be presented.

2.1 Machine Learning

Machine learning uses experiences to train a function or a model. The experiences are either available in datasets or are collected. Predictions about the experiences can be used to calculate an error or loss that is then used to refine the function or model being trained. Machine learning distinguishes three learning styles. To solve a problem using machine learning, the learning style has to be determined first, because different styles can only solve certain problems. Therefore, follows an introduction of the different learning styles.

2.1.1 Styles of Learning

Machine learning aims to give computers the ability to learn the way humans do. In general, machine learning can be divided into three subfields:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

2.1.1.1 Supervised Learning

As the name already indicates, learning with this method is supported by a kind of supervisor. This supervisor can be considered as a dataset with known outputs. Usually, we call this dataset ground truth and the outputs labels. The goal is to learn the relationship between the input x and the output y. Since we want to classify novel inputs $x*$, which are not present in the ground truth, it is necessary to use a function approximator $y* = f(x*)$ to predict the output $y*$. By means of a loss function $L(y, y*)$, which calculates the error between the labels and the predictions, the function approximator can learn. In general, this learning method is used for two types of problems. First, when data must be classified with an integer label, such as the recognition of dogs or cats on pictures. Second, for regression tasks, which means that the data can not be classified by a simple integer value. For example, the forecast of a stock price because the output must be a real number. [Bar12, pp. 285-286]

As already mentioned in section 1.2 one goal is to master the tasks in the best way possible. Thus, we can not use supervised learning for this thesis. For example, imagine our goal is to master a computer game and we address this problem with supervised learning. To generate the ground truth we invite an expert player and let him play the game a millon times. After training our function approximator with this data, we find that our result is just as good as the expert player. Of course, we humans learn from observations too, but through practice we can become even better than this observed behaviour.

2.1.1.2 Unsupervised Learning

In unsupervised learning, we want to learn by observation and find the hidden structures behind unlabeled datasets. Usually, by hidden structure a compact data description of large datasets is meant. This can be used to cluster data, as object segmentation in pictures or for anomaly detection. However, it is not possible to determine the accuracy of such algorithms since there are no labels. [Bar12, pp. 286-287][SB98, p. 2]

Obviously, we can not use this method for our thesis, because the agent has to learn to solve tasks, rather than the hidden structures or patterns behind the perceived data.

2.1.1.3 Reinforcement Learning

Reinforcement Learning (RL) does not use labels for datasets in the same way as supervised learning does. There is no ground truth present, which describes the data to expect. Just because there are no labels from the beginning, one might think that RL can be considered as a subset of unsupervised learning, as it was often the case in the past. In RL the agent receives a reward or punishment signal through exploration of an environment. Obviously, it is attempted to maximize this reward signal in the long-term, because the amount of reward describes how well or worse the agent had performed. Learning can now be described as finding actions that result in a higher reward. Therefore, the reward signal is transferred to an expected reward (or expected return) $\mathcal{R}$, which defines how much reward to expect, for each state. These expected rewards can be used to calculate an error between different explorations. By ongoing explorations, the expected rewards can be refined iteratively. This usually happens by the improvement of a so-called *policy* or *value function*, which can be considered as learning. Although, the expected rewards can be understood as labels, as it is the case in supervised learning, the maximum improvement is not limited by a ground truth. In addition, these labels can be used to determine an accuracy. Therefore, RL can not be considered as supervised or unsupervised learning [SB98, p. 2].

In literature [SB98, p. 38] the standard interaction of an agent with its environment can be observed in figure 2.1. Actions A_t taken by an agent, who is learner and decision maker, change the state $S_t \rightarrow S_{t+1}$ of the environment. These changes may result in rewards or punishments R_t that are granted to the agent and reflect how appropriate his behaviour has been in the past. This experience is used to learn how to solve the task, which means choosing actions that maximize the reward in a long-term. [HMLIR07, p. 2]

Basically, RL methods can be divided into two groups: *Model-Free* or not. Not *Model-Free* methods require prior knowledge about the environment, which means to know the transition probabilities $\mathcal{P}$ and/or expected rewards $\mathcal{R}$. More information about transition probabilities $\mathcal{P}$ and expected rewards $\mathcal{R}$ are given in section 2.1.2. As mentioned in the paragraph above, our agent has no a priori knowledge about the environment. Therefore, we only discuss *Model-Free* methods within this thesis.

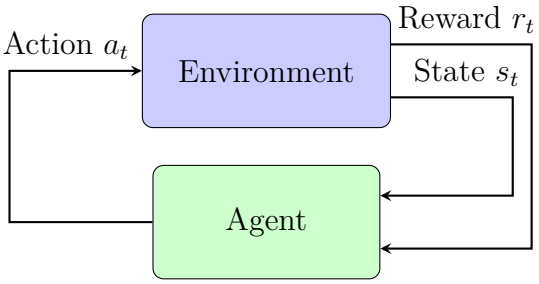


Figure 2.1: Interaction between agent and environment.

2.1.1.4 Policy

A *policy* π is a function which maps states s to actions a. At a certain time, the agent perceives a state s. The *policy* determines which action s the agent should perform. In most cases, the *policy* aims to maximize the cumulative reward R. A *policy* can either be stochastic, for example if the agent's movement is unpredictable due to a slippery surface, observable in equation 2.1, or deterministic, observable in equation 2.2 .

$$\pi_t(s|a) \tag{2.1}$$

$$\pi_t : s \rightarrow a \tag{2.2}$$

In general, RL algorithms can be classified as *On-* or *Off-Policy* learning methods. *On-Policy* methods are based on a known policy $\pi(s)$. This means that in each state s the action a, which leads to the highest amount of reward, must be known. Obviously, in this case the agent requires prior knowledge about the task and environment.
With *Off-Policy* algorithms, the policy $\pi(s)$ is learned, which means the action a leading to a high amount of reward is unknown. Thus, the agent has to experiment with his environment and after receiving a reward R, the actions $(a_0, a_1, a_2, \ldots, a_n)$ leading to it are known. On this basis, the policy $\pi(s)$ can be improved.

Due to the fact that our agent has no prior knowledge about the environment or task, mentioned in section 1, we use *Off-Policy* algorithms for this thesis.

2.1.1.5 Reward

The reward (or penalty) R is a numeric scalar value granted to the agent through actions. Each action a ends in a reward, so that the total reward which the agent receives over time can be expressed as follows:

$$R_t = r_1 + r_2 + r_3 + \cdots = \sum_{t=0}^{N} r_{t+1} \tag{2.3}$$

Usually, there is no reward r_0 given for the initial state s_0, as observable in equation 2.3. N is the *horizon* or *episode length*, that can be infinite. In general, with regard to their delay, rewards can be divided into three classes:

- immediate reward
- delayed reward
- pure-delayed reward

Immediate rewards mean that a reward is given directly after an action and no future rewards need to be considered. To maximize the reward, only the selected action and the current state of the agent are important. For example, if you play with a k-armed

bandit, you will be rewarded right after pulling the lever. No past experiences need to be considered.
Delayed rewards mean that an action can generate immediate rewards, but the future must be considered; at least the next state of the environment. These problems are more difficult to solve because the agent has to choose actions that pay off in the future. For example, cooking could be considered as such a problem. If the dish is removed from the pan too soon and some of the ingredients are still not cooked properly, waiting would have been better and thus more rewarding.
Pure-delayed reward means that the reward is the same for all states except the last state of an environment. These problems can be very difficult to solve if reaching the final state is not very likely. Playing chess, for example, could be such a problem because you only give the agent a reward for winning or losing the game [Gas02, p. 7].

2.1.2 Markov Decision Process

Basically, RL techniques try to solve finite Markov Decision Processes (MDPs), which were first mentioned in the article [Bel57, pp. 1-6]. With MDPs decision making situations can be modeled. In principle MDPs are finite, discrete time and stochastic automata defined as a 5-tuple (S, A, P_a, R_a, γ) 2.4 [HMLIR07, pp. 1-2].

$$\begin{aligned} &S \ldots \text{finite set of states} \\ &A \ldots \text{finite set of actions} \\ &P_a(s, s') = Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \ldots \text{state transition probabilities} \\ &R_a(s, s') \ldots \text{reward for state transition} \\ &\gamma \ldots \text{discount factor} \end{aligned} \tag{2.4}$$

Figure 2.2 shows an example of a discrete time and stochastic MDP consisting of two states $\{S_0, S_1\} \in S$, two actions $\{a_0, a_1\} \in A$ and two different rewards $\{-1, 1\}$. Starting from the initial state S_0 for each timestep t the agent can perform two actions $\{a_0, a_1\}$. Action a_0 will result with a probability of *0.6* in state S_1. Otherwise the agent remains in state S_0. Action a_1 will always result in state S_0 due to the transition probability of *1.0*. For actions which results in state S_0 the performer is penalized with a reward of *-1*. For actions which yields to the state S_1 the reward is set to a value of *1*. Performing only the action a_0 on this MDP will result in the highest possible reward.

MDPs describe the agent's interaction with the environment and vice versa. At each timestep t, the agent is located in state s_t and performs an action a_t selected by a *policy* $\pi(s_t)$ that leads to a successor state s_{t+1} and receives a reward $R_{a_t}(s_t, s_{t+1})$. Obviously, the amount of rewards depends on the used policy π, so our goal is to find the optimal policy $\pi_*(s)$. Since a policy, described in section 2.1.1.4, aims to maximize the cumulative reward, equation 2.3, this can be formulated as an optimization issue where the total discounted rewards, equation 2.5, must be maximized over time.

$$G_t = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \tag{2.5}$$

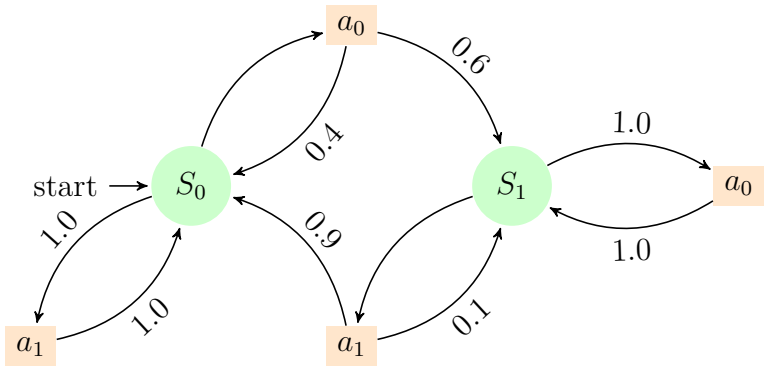


Figure 2.2: Example of a discrete time and stochastic MDP.

Note that the total discounted rewards 2.5, have an infinite time horizon $N \to \infty$. To guarantee the convergence of G_t the discount factor $\gamma \in [0, 1]$ has to be introduced.

2.1.2.1 Value Functions

As explained in section 2.1.2 the goal is to optimize the total discounted rewards over time G_t but this knowledge does not help our agent to decide which action a it should choose in a certain state s. This would require a scalar number that estimates how good it is to be in specific state s or which action $a \in A$ should be performed next to receive a high amount of reward.

This can achieved with the help of the so called *state-value function* $v_\pi(s)$ 2.6 and *action-value function* $q_\pi(s, a)$ 2.8. The *state-value function* $v_\pi(s)$ provides the expected cumulative discounted reward (expected return) for a state s under a policy $\pi(s)$. On the other hand, the *action-value function* $q_\pi(s, a)$ provides the expected return for a state s taking the action a under a policy $\pi(s)$. $\mathbb{E}$ denotes the expected value of a random variable. [SB98, pp. 45-53]

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^t r_{k+t+1}|S_t = s\right], \text{for all } s \in S \tag{2.6}$$

Optimal:

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.7}$$

$$q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^t r_{k+t+1}|S_t = s, A_t = a\right] \tag{2.8}$$

Optimal:

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{2.9}$$

Intuitively, if the state visits of an agent in any MDP will emerge to infinity, the expected return will converge to the optimal value functions $v_*(s)$ and $q_*(s, a)$. Then the optimal

value of a state s under a policy $v_\pi(s)$ must be equal to the expected return from the best action from that state, as can be observed in equation 2.10.

$$v_*(s,a) = \max_{a \in A(s)} q_*(s,a) \tag{2.10}$$

In order to get in touch with the value functions, an example is presented at this point. Imagine an agent being exposed to a random location in a grid cell maze environment consisting of 16 states (floors), observable in figure 2.2. At any state the agent can go up, down, left, right or stay at the current position. Grid cells are drawn with a thin black line, while walls are drawn with a larger one. If an action pushes the agent into a wall, it remains at its current position. The agent's goal is to learn the quickest way through the maze and reach the exit floor which is located in the bottom left. After reaching the exit floor, the agent perceives a reward of *+1* and is exposed at a random position again. As can be seen in the reward map 2.1a, all other rewards are set to zero. The discount factor γ is chosen as $\gamma = 0.9$. The converging of the state-value function $v_\pi(s)$ to the optimal state-value function $v_*(s)$ can be observed in figure 2.1b. The same applies to the action-value function $q_\pi(s,a)$ which will converge to the optimal action-value function $q_*(s,a)$ 2.1d. As already apparent in the equation 2.10, the optimal action-value $q_*(s,a)$ corresponds to the optimal state-value $v_*(s)$.

(a) Reward Map

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	1.0

(b) Optimal State-Value Function

0.53	0.59	0.66	0.73
0.59	0.53	0.48	0.81
0.66	0.48	0.81	0.90
0.73	0.81	0.90	1.0

(c) Optimal Policy

→	→	→	↓
↓	←	←	↓
↓	↑	→	↓
→	→	→	↻

(d) Optimal Action-Value Function

0.53 →	0.59 →	0.66 →	0.73 ↓
0.59 ↓	0.53 ←	0.48 ←	0.81 ↓
0.66 ↓	0.48 ↑	0.81 →	0.90 ↓
0.73 →	0.81 →	0.90 →	1.0 ↻

Figure 2.2: Value functions description using a simple maze world.

2.1.3 Bellman Equations

Using the value functions equations: 2.6 and 2.8; our agent can already decide which actions a to select in each state s, but so far it has not been clear how to calculate these values through exploration. This is where the *Bellmann equations* [Bel54, pp. 1-19] comes into play. With the use of the recursive property of the total discounted rewards over time $\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}$ the state-value function $v_\pi(s)$ 2.6 and action-value function $q_\pi(s,a)$ 2.8 can be expanded to the Bellman equations 2.11 2.13:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) \left[r + \gamma v_\pi(s') \right] \end{aligned} \tag{2.11}$$

Optimal:

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.12}$$

Considering the final expression from equation 2.11, $\gamma v_\pi(s')$ can be seen as the expected reward from the next state s' multiplied with the discount factor γ to make this equation convergent. Note that the expected return $\mathbb{E}_\pi$ following an policy π can be rewritten as $\sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)$, which can be seen as summing over all possible actions and all possible returning states [SB98, p. 46].

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &= \sum_{s',r} p(s',r|s,a) \left[r + \gamma v_\pi(s') \right] \end{aligned} \tag{2.13}$$

Optimal:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.14}$$

The *Bellman optimality equation* 2.12 is a fix point equation which provides a unique solution for a finite MDP. Any finite MDP with n states can be converted into n equations with n unknowns [SB98, pp. 50-51]. Learning algorithms which use equation 2.12 to find an optimal solution $v_*(s)$ are called *value iteration* algorithm, observable in listing 1. Although, these algorithms are *Off-Policy*, because $v_*(s)$ can be calculated without a policy $\pi(s)$, they are not *Model-Free* since solving equation 2.12 requires the transition propabilities $\mathcal{P}$ and expected rewards $\mathcal{R}$.

Initialise $V(s) \in \mathbb{R}, \text{e.g} V(s) = 0$
$\Delta \leftarrow 0$
while $\Delta < \theta$ *(a small positive number)* **do**
  foreach $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end
end
output: Deterministic policy $\pi \approx \pi_*$ such that
$\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma V(s')\right]$

Algorithm 1: Value Iteration

2.1.4 Q-Learning

As mentioned in section 1.2 and 2.1.1.3, a *Model-Free* algorithm is required for our issue. Consequently, the just mentioned *value iteration* algorithm 1 can not be used, because it would presuppose to know the transition probabilities $\mathcal{P}$, as well as the expected rewards $\mathcal{R}$. May it is possible to obtain these unknowns by allowing an agent to experiment with the consequences of its actions taken in a MDP. However, since the policy $\pi(s)$ is also unknown, the agent has no knowledge how to act at any given state. Overall, the agent has to learn the transition probabilities $\mathcal{P}$, the expected rewards $\mathcal{R}$ as well as the policy $\pi(s)$. To solve this problem, at least two policies are typically used: One just for exploring the environment an a second one which is trained. For example, a random policy $\pi_{\epsilon-greedy}(s)$ where the agent performs random actions on a MDP and collects experiences. These experiences contain small partial parts of the unknown variables. With this knowledge the second policy $\pi(s)$, which is used after training to exploit the environment, can be trained. When the state visits approach towards infinity, the trained policy $\pi(s)$ will converge to the optimal policy $\pi_*(s)$. At that point the agent can stop performing random actions and start to act accordingly to the trained policy $\pi(s)$. That is exactly the idea behind *Q-learning*. [WD04, pp. 1-14]

In order to understand *Q-learning* the *Q-function* $Q(s_t, a_t)$ 2.15 has to be explained. This function provides the expected reward (total expected cumulative discounted reward) performing an action a_t from state s_t and following policy π thereafter. From the *Bellmann equation* 2.13 the *Q-function* 2.15 can be derived [HMLIR07, pp. 2-3]:

$$Q_\pi(s, a) = \sum_{s',r} p(s', r|s, a) \left[r + \gamma V_\pi(s')\right] \tag{2.15}$$

Optimal:

$$Q_*(s, a) = \sum_{s',r} p(s', r|s, a) \left[r + \max_{a' \in A} Q_*(s', a')\right] \tag{2.16}$$

An agent using the *Q-learning* approach updates the corresponding *Q-value* (expected reward) after each observed transition. Any action taken by an agent on an MDP will result in such a transition. A transition is a 4-tuple $(s_t, a_t, r_{t+1}, s_{t+1})$, that consists of the current state s_t, the performed action a_t, the resulting state s_{t+1} and the reward r_{t+1} earned. *Q-values* can be calculated with the help of the Q-function 2.15 and by approximating the state-value function v_π. This can be done by using the *Temporal Differences*(TD) method [Day92, pp. 2-3] resulting in equation 2.17. The *Q-values* can be stored by the agent in a look-up table, better known as *Q-table* [HMLIR07, p. 6].

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \left[\underbrace{r_{t+1}}_{\text{reward}} - \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_{a' \in A} Q(s_{t+1}, a')}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right] \tag{2.17}$$

With equation 2.17 we can present the *Temporal Differences Q-learning* algorithm:

Initialise $Q(s, a)$ arbitrarily and $Q(terminal - state,) = 0$
foreach *episode* $\in$ *episodes* **do**
  while *s is not terminal* **do**
    Choose a from s using policy derived from Q (e.g., ϵ-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
    $s \leftarrow s'$
  end
end

Algorithm 2: TD Q-learning

2.1.5 Deep Q-Learning

A big problem with the *TD Q-learning* algorithm, observable in listing 2, is, that the *Q-values* must be stored in a *Q-table*. Suppose that the TD Q-learning algorithm 2 is used for a simple computer game with a screen size of 64x64 pixels. Of course, the computer game uses coloured images, but since this example can not derive any helpful information from the image colors, they are converted into greyscale images with 256 grey levels. For motion tracking four consecutive images are used. This would result in a huge Q-table with $256^{64*64*4}$ rows. In literature this problem is called *Curse of Dimensionality* [KP12, p. 5].

Since our agent operates in huge continuous state spaces and memory is limited, the *TD Q-learning* approach can not be used. In addition, as mentioned in section 1.2, it is required to use a resource-saving method. The team of Google *DeepMind* were facing the same problem. [MKS+13, p. 19] They overcame this issue by using a *neural network* (called *Q-network*) with weights θ to approximate the *action-value* function $q_\pi(s, a)$2.11 resulting in $Q(s, a; \theta) \approx Q_*(s, a)$. Neural networks are non linear function approximators

which are very flexible. Appendix A provides more information about neural networks. Consequently, the use of Q-networks leads to a change in the learning architecture, which can be seen in figure 2.3. Instead of using a state s and an action a to update the Q-table, we only feed the Q-network with a state s and obtain a Q-value prediction for each action a. As always, the highest Q-value represents the best action that can be performed from this state s. The *Deep Q-learning network* (DQN) is trained with a *loss function*. With a

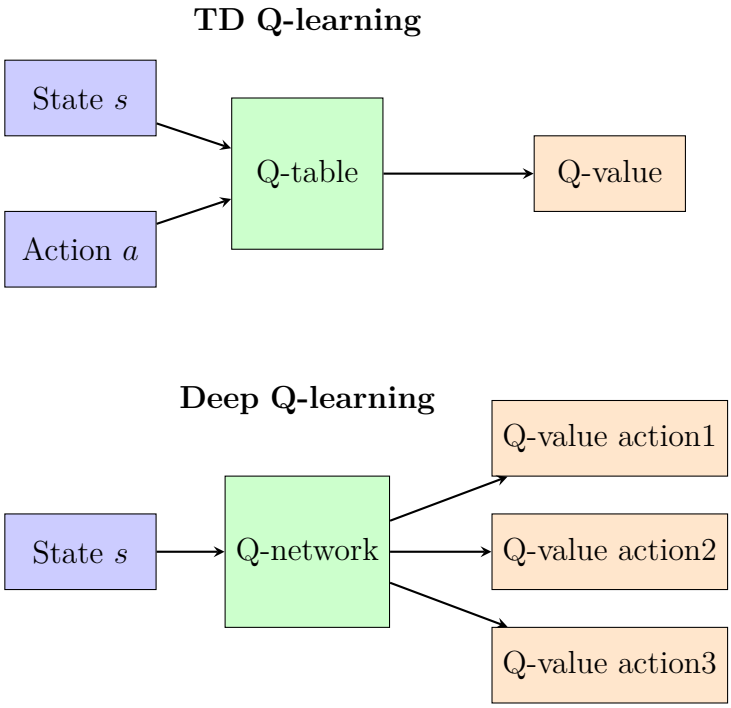


Figure 2.3: Difference between TD Q-learning and Deep Q-learning architecture.

loss function 2.18, the error of a neural network can be determined and with the gradient of its derivative, the weights θ are changed.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\underbrace{(r + \gamma \max_a Q(s', a'; \theta_{i-1})}_{\text{target}} - \underbrace{Q(s, a; \theta_i))^2}_{\text{prediction}}\right] \tag{2.18}$$

$U(D)$ is the *experience replay history* which is stored in the *replay memory*. This memory stores the last N transitions and at every learning step a minibatch is sampled and used to train the neural net. More information about replay memories can be read in section 2.1.9. The DQN learning algorithm can be observed here 3.

Initialise replay memory D with capacity N
Initialise exploration method E
Initialise $Q(s, a)$ arbitrarily
foreach *episode* $\in$ *episodes* **do**
  while s *is not terminal* **do**
    Use exploration method E to choose an action $a \in A(s)$
    Take action a, observe r, s'
    Store transition (s, a, r, s') in D
    Sample random minibatch of transitions (s_j, a_j, r_j, s'_j) from D
    Set $y_j \leftarrow \begin{cases} r_j & \text{for terminal } s'_j \\ r_j + \gamma \max_a Q(s', a'; \theta) & \text{for non-terminal } s'_j \end{cases}$
    Perform gradient descent step on $(y_j - Q(s_j, a_j; \Theta))^2$
    $s \leftarrow s'$
  end
end

Algorithm 3: Deep Q-learning algorithm.

2.1.6 Double DQN

One problem with DQN, described in section 2.1.5, is that small changes in the Q-values can lead to rapid policy changes and thus the policy can begin to oscillate. This leads to an overestimation of the Q-values, which harms the task solving performance. [HGS15, p. 1] To prevent this particular case, *Double DQN (DDQN)* [HGS15] uses two Q-networks: $Q(s, a; \theta)$ and $Q(s, a; \theta^-)$. Q-network $Q(s, a; \theta^-)$ is usually called target Q-network in literature. While the Q-network $Q(s, a; \theta)$ is used for action selection only, the other Q-network $Q(s, a; \theta^-)$ evaluates actions. This means that the Q-network $Q(s, a; \theta^-)$ predicts the Q-values of the next states (s', a') during training. It is attempted to keep this Q-network $Q(s, a; \theta^-)$ as stable as possible over several transitions. To achieve this, its weights θ^- are updated slowly with equation $\theta^- = (1 - \tau)\theta^- + \tau\theta$. If τ is to be chosen small enough, usually $\tau \leq 10^{-3}$, the weights of the evaluation Q-network converges slowly to the weights of the action Q-network $\theta^- \to \theta$. As a consequence, small changes in the Q-values do not lead to rapid policy changes anymore. The *DDQN* algorithm can be observed in listing 4.

2.1.7 Deep Deterministic Policy Gradient

As already observable in figure 2.3 with DQN (section 2.1.5) and DDQN (section 2.1.6) the predicted Q-values only determine which action leads to the highest amount of reward. It does not describe, with how much energy or force this action should be performed. For robotic control, this information is essential. Of course, we can discretize each action of the action space to extend DQN or DDQN for continuous action spaces. This approach has many drawbacks: First, the curse of dimensionality. Imagine an agent with four actions and a discretization of ten. The corresponding Q-network will have an output

Initialise replay memory D with capacity N
Initialise exploration method E
Initialise τ
Initialise $Q(s, a; \theta)$ and $Q(s, a; \theta^-)$ arbitrarily
foreach *episode* $\in$ *episodes* **do**
  while s *is not terminal* **do**
    Use exploration method E to choose an action $a \in A(s)$
    Take action a, observe r, s'
    Store transition (s, a, r, s') in D
    Sample random minibatch of transitions (s_j, a_j, r_j, s'_j) from D
    Set $y_j \leftarrow \begin{cases} r_j & \text{for terminal } s'_j \\ r_j + \gamma Q(s', \max_a Q(s', a'; \theta); \theta^-) & \text{for non-terminal } s'_j \end{cases}$
    Perform gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
    Update weights $\theta^- = (1 - \tau)\theta^- + \tau\theta$
    $s \leftarrow s'$
  end
end

Algorithm 4: Double Deep Q-learning algorithm.

size of forty and an action space size of $4^{10} = 1048576$. Efficient exploration of such large action spaces is very difficult and successful learning can not longer be guaranteed. In addition, instead of four output neurons, this Q-network will need forty, which will slow down training drastically. Second, discretization implies the loss of information that can cause states to become inaccessible. This can make learning very difficult or even impossible. [LHP+15, pp. 3-4] *Deep Deterministic Policy Gradient (DDPG)* introduces an actor-critic (AC) algorithm which can deal with continuous action spaces. AC algorithms are hybrid methods which combine policy-based and value-based RL algorithms. Figure 2.4 explains the relationship of this methods.

DDPG uses two Q-networks, of which one learns to act (actor), while the other

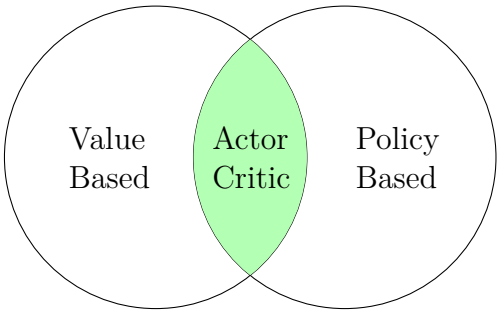


Figure 2.4: Realtionship between value-based, policy-based and AC algorithms.

learns to criticize the taken action (critic). Just as DQN and DDQN the critic is trained with the value-based learning approach, which means the approximation of the *action-value* function $q_\pi(s, a)$ and is therefore trained with the standard Deep Q-learning

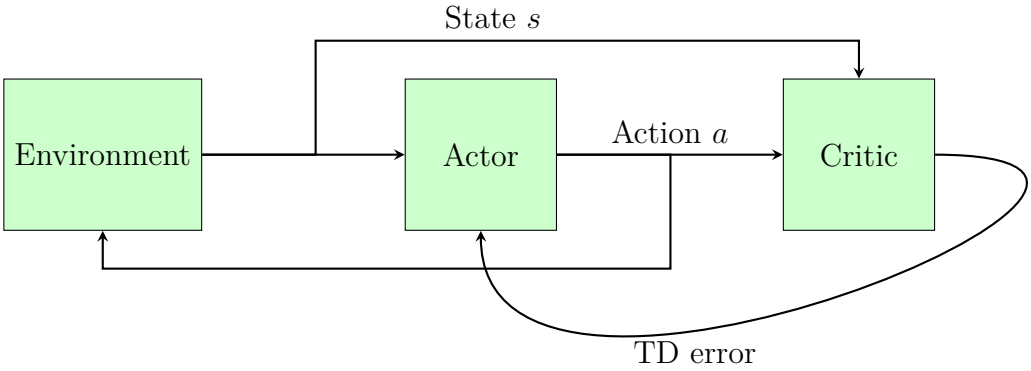


Figure 2.5: DDPG Architecture.

approach, mentioned in section 2.1.5. On the other hand, the actor uses the policy-based learning approach and tries to approximate the policy function $\pi(s)$. The actor Q-network has to be trained with the policy gradient $\nabla_\theta J = \frac{\partial Q(s,a;\theta^Q)}{\partial a} * \frac{\partial \mu(s;\theta^\mu)}{\partial \theta^\mu}$ which is proved that this equation calculates the gradient of the policy's performance [LHP+15, p. 3]. The architecture of this approach can be observed in figure 2.5. The environment provides the state for the actor and critic Q-network. During the training, the actor predicts the next action to perform, which is then criticized by the critic. This criticism can be used to train the actor while the critic is trained with the starndard Deep Q-learning approach. The DDPG algorithm can be observed in listing 5.

Initialise replay memory D with capacity N
Initialise exploration method E
Initialise Critic $Q(s, a; \theta^Q)$ with random weights θ^Q
Initialise Actor $\mu(s; \theta^\mu)$ with random weights θ^μ
foreach *episode* $\in$ *episodes* **do**
  while s *is not terminal* **do**
    Use exploration method E to choose an action $a \in A(s)$
    Take action a, observe r, s'
    Store transition (s, a, r, s') in D
    Sample random minibatch of transitions (s_j, a_j, r_j, s'_j) from D
    Perform Q-learning e.g. DQN or DDQN
    Update critic with $(y_j - Q(s_j, a_j; \theta))^2$
    Update actor with policy gradient $\nabla_\theta J = \frac{\partial Q(s,a;\theta^Q)}{\partial a} * \frac{\partial \mu(s;\theta^\mu)}{\partial \theta^\mu}$
    $s \leftarrow s'$
  end
end

Algorithm 5: DDPG algorithm.

2.1.8 Reward Function

The *reward function* determines for which states the agent is rewarded or penalized. This function is crucial for solving a task correctly. Minor errors or few adjustments to this function, such as providing too little or too much reward, giving reward at the wrong time or rewarding subgoals, can cause the agent to learn a completely different task. As mentioned in this dissertation [Gas02, p. 8], researchers worked on a robot pushing task, in which a robot had to push boxes. Unremarkable minor mistakes in the reward function have caused the robot to crash into the wall or bypass the boxes completely. In addition, the amount of immediate reward given should be chosen very carefully. Giving naive high rewards can make the gradients of a Q-network large and unstable when backpropagated [8, p. 28]. Therefore, rewards should be clipped.

A simple approach for designing a reward function is to reward all the small successes of an agent. Through this approach, it is likely that the agent learns to master a subgoal rather than the main goal of a task. The reason for that is that Q-learning tries to maximize the total cumulative reward it receives in the long term. Repeated execution of a subgoal can lead to such high rewards that achieving the main goal is no longer attractive. As rewarding subgoals is very difficult, it is recommended in literature to omit this completely [SB98, p. 42]. Now it is more obvious, why RL faces many problems with sparse rewards.

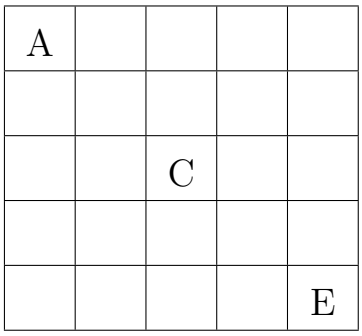
Therefore, a task usually consists only of one goal state, in which a positive reward is achieved, or of an shaped reward. Shaped rewards mean that the amount of perceived reward depends on a function. For example, this function may be the distance between the agent and its destination to be reached. This would encourage the agent to reduce this distance. However, reward shaping requires a very good domain knowledge [AWR+17, p. 8], because if the underlying function is poorly designed, it can be seen as rewarding subgoals, causing the agent to learn a completely different task. Negative rewards encourage the agent to leave the current state as fast as possible. Moreover, in common Q-learning methods, these negative rewards are not backprobagated more than one state, since only the maximum argument $\max_a Q(s', a'; \theta)$ is considered [FAS10, p. 1]. Therefore, negative rewards can be applied for each state that the agent should leave as soon as possible.

In order to understand this context better, an example is provided in figure 2.6. Here, the agent's (sign A in figure) main goal is to reach the exit point E of a maze. For reaching the exit point E the agent will be rewarded with *1.0*. He is powered with a battery and therefore one charging field C is placed in the middle of the maze. At any state the agent A can move up, down, left, right or stay at his current position. Actions that move the agent A out of the maze are not possible. In this case, he will be moved back to the last correct position. Figure 2.2a shows the initial state in which the agent A starts. After the charging field C has been reached, the battery of the agent is loaded. When the battery power drops to zero, the agent A will be penalized with *-1.0* and will be reset to the initial state 2.2a. Since overcharging the battery is not useful, the agent will

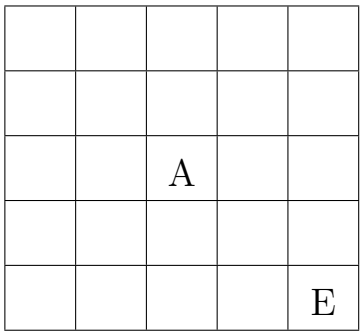
be punished with *-1.0* for such actions. After six actions the battery power will drop to zero, so the agent has to charge his battery in order to reach the exit point E. Therefore, he will be rewarded with *0.1* for charging. Now charging can be seen as a subgoal and rewarding this subgoal will result in a wrong behaviour. The agent will learn to move on a charging field C 2.2b and charge his battery. After the battery is full, he will move away from the charging field C 2.2c. Finally, he will move back to the charging field 2.2d again. This behaviour is repeated forever, as more reward is generated in the long term $t \to \infty$ than reward is perceived for reaching the exit point E.

Figure 2.6: Example maze where a subgoal is rewarded.

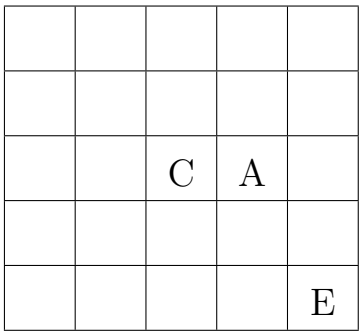
(a) Initial state



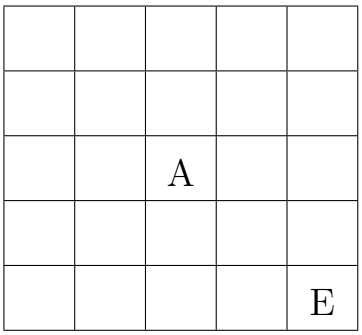
(b) Timestep $n + 0$



(c) Timestep $n + 1$



(d) Timestep $n + 2$



2.1.9 Replay Memory

All Deep Q-learning algorithms, mentioned in this work, 3 4 5, need to store transitions in a replay memory. If it is not the case, it is obvious that the Q-network must be trained by successive transitions. This is like learning only on the basis of immediate experiences, without considering the past. Imagine a child who is trying to learn how to walk without past experiences. It may learn to move forward somehow, but it is very unlikely that it will succeed in walking on its feet. That is because past experiences are missing. Therefore, past experiences have to be considered to enable a successful learning approach. This can be done by saving transitions in a so called replay memory. Moreover, the learning of

successive transitions is very inefficient due to the strong correlation between them. To break up these correlations, the transitions are usually sampled randomly. In addition, after each action, a minibatch, which contains at least two transitions, is sampled from the whole memory. Therefore, it is likely that every transition is replayed often enough. This results in more weight updates and thus in better data efficiency. After the memory is full, the oldest transitions are deleted. [MKS+13, pp. 4-5]

The collaboration of the environment, Q-learning method and replay memory can be observed in figure 2.7. The Q-learning method perceives the state s from the environment and predicts the next action a to perform. Then the next state s' and reward r is available. Finally, the transition (s, a, r, s') is stored in the replay memory.

Unfortunately, using this kind of replay memory leads to more issues, such as *catastrophic forgetting* [KPR+17, p. 1]. This means the Q-network has either learned a task correctly, but forgets about it by simply being trained with many useless transitions, or it was not trained with transitions that solve the task at all. The first case, for example, occurs if an agent finds the exit point of a maze early and the Q-network learns about it. However, due to worse exploration, the exit point is not found again and therefore no more transitions, leading to this exit point, are stored in the replay memory. After all, the Q-network is trained too rarely with useful transitions and it will slowly forget about the exit point. The second case occurs due to random sampling or deletion of transitions from the replay memory. These deleted or not sampled transitions could have contained important information required to accomplish the task. It may happen that these transitions were not experienced by the Q-network at all. A successful learning process can be difficult or even impossible on this basis. This section lists popular replay memory methods.

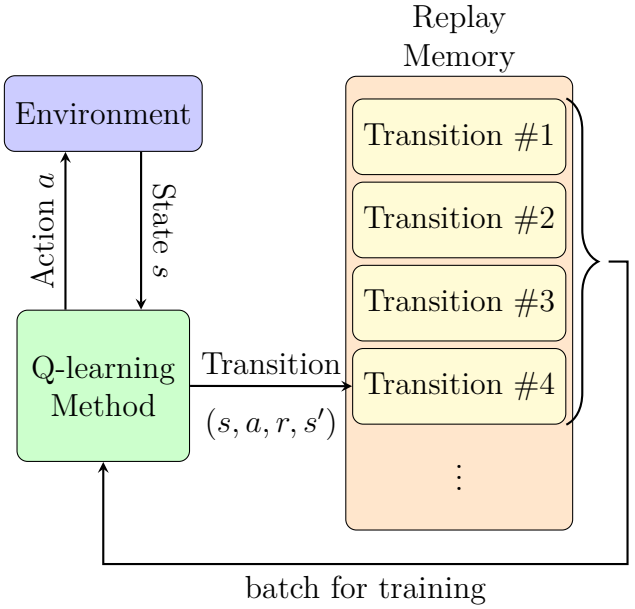


Figure 2.7: Collaboration of the replay memory with the Q-learning method and environment.

2.1.9.1 Experience Replay

Experience Replay (EXPR) [MKS+13, pp. 4-5] is the simplest and most widely used one. Transitions are stored one after the other in a memory of size N. The most recent transitions experienced are always stored at the end of the memory. Minibatches are sampled randomly distributed from the whole memory. After the memory is full, the oldest transitions, which are the transitions at the beginning of the memory, are discarded. Intuitively, this memory is most susceptible to catastrophic forgetting.

2.1.9.2 Prioritized Experience Replay

Introduced in paper [SQAS15, pp. 1-10] *Prioritized Experience Replay (PEXPR)* takes advantage of the fact, that the temporal difference (TD) error δ of transitions can easily be calculated with algorithms using Q-learning approach: $\delta = |Q_{new} - Q_{old}|$ with $Q_{old} = Q(s, a; \theta)$ and $Q_{new} = r + \gamma \max_a Q_(s', a'; \theta)$ (in the case of DQN). This TD error shows how surprising or unexpected a transition is: in other words, the higher the TD error of a transition, the more the agent can learn from these transitions. Sampling only transitions with high TD error can make a system prone to overfitting, due to the lack of diversity. Therefore, the TD error is converted to a priority. The probability of sampling transition i is defined as 2.19

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2.19}$$

where $pi > 0$ is the priority of transition i and exponent *alpha* determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case. To guarantee that even transitions with low priorities are sampled with a non zero probability from the memory, multiple methods are used, like setting the priority of a transition to maximum for the first insert. This ensures that these transitions are sampled for sure in the upcoming minibatch. The paper introduces to types of prioritizations: *rank-based (PRANK)* and *proportional-based (PPROP).*

2.1.9.3 Hindsight Experience Replay

In a sparse reward environment, where rewards are only given if a goal is reached, usually multiple entire episodes without any positive rewards are stored in the memory. Assume, an agent is equipped with a battery and his goal is to reach a charging station. When the charging station is not reached within an episode, the episode ends without a positive reward stored. From such episodes, the Q-learning approach only learns which actions should not be performed. Of course, this is not a completely useless knowledge, but in the end it does not help to solve the task. So Q-learning learns little or even nothing from such episodes. However, these episodes contain useful information, such as how non goal states can be reached. This knowledge is useful and can help the agent to solve the task better and faster. [AWR+17, pp. 3-4]

Consider an episode sequence like observable in figure 2.8 where the agent failed to reach a desired goal and received only negative rewards. This sequence is not helpful

$$\begin{array}{lccccccccccc} \textit{states}: & s_0 & \rightarrow & s_1 & \rightarrow & s_2 & \rightarrow & \dots & \rightarrow & s_{t-1} & \rightarrow & s_t \\ \textit{rewards}: & -1 & \rightarrow & -1 & \rightarrow & -1 & \rightarrow & \dots & \rightarrow & -1 & \rightarrow & -1 \end{array}$$

Figure 2.8: Example of an episode in which no goal state was reached.

for learning how to reach the goal state, but it tells us how to reach the state s_t. In addition, if this state must be visited in order to solve the task, the knowledge about how to reach it, can be considered as very important. To enable learning we define s_t as a *virtual goal* or *additional goal*, reward it with zero and add it to the memory. So the episode contains at least one non-negative reward that supports learning. Figure 2.9 shows an example of an episode with one virtual goal inserted. This is exactly the

$$\begin{array}{lccccccccccc} \textit{states}: & s_0 & \rightarrow & s_1 & \rightarrow & s_2 & \rightarrow & \dots & \rightarrow & s_{t-1} & \rightarrow & s_t \\ & & & & & & & & & \searrow & & \\ & & & & & & & & & s_{virtual} & & \\ \textit{rewards}: & -1 & \rightarrow & -1 & \rightarrow & -1 & \rightarrow & \dots & \rightarrow & -1 & \rightarrow & -1 \\ & & & & & & & & & \searrow & & \\ & & & & & & & & & 0 & & \end{array}$$

Figure 2.9: Episode with one virtual goal inserted.

idea behind *Hindsight Experience Replay (HER)* [AWR+17, pp. 1-11]. With HER, each state of an episode is treated as a virtual goal and for each of them, k virtual goals are inserted into the memory additionally. The article presents several policies for sampling the k virtual goals. However, we use the *future* method, which samples the virtual goals from the same episode. However, these virtual goals has to be observed after the current treated state. An example of this can be observed in figure 2.10. The ability to learn

$$\begin{array}{lccccccccccc} & s_{1\dots t_{v0}} & & s_{2\dots t_{v0}} & & s_{3\dots t_{v0}} & & & & s_{t_{v0}} & & \\ & \uparrow & & \uparrow & & \uparrow & & & & \uparrow & & \\ \textit{states}: & s_0 & \rightarrow & s_1 & \rightarrow & s_2 & \rightarrow & \dots & \rightarrow & s_{t-1} & \rightarrow & s_t \\ & \downarrow & & \downarrow & & \downarrow & & & & \downarrow & & \\ & s_{1\dots t_{v1}} & & s_{2\dots t_{v1}} & & s_{3\dots t_{v0}} & & & & s_{t_{v1}} & & \end{array}$$

Figure 2.10: HER example with $k = 2$. $s_{1\dots t_{v0}}$ defines a virtual goal which is randomly drawn from state 1 to t. With s_t this episode ends.

more from episodes that did not reach a main goal allows HER to deal well with delayed and sparse rewards. Additionally, HER only instructs how virtual goals are generated and not how to save them. Therefore, HER is with all other replay memory methods, like Experience Replay 2.1.9.1 or Prioritized Experience Replay 2.1.9.2 combinable. In order to decide which goal the Q-network should follow, the state of the goals, virtually or main, is additionally provided as input. Therefore, the input size (state space) of the Q-network

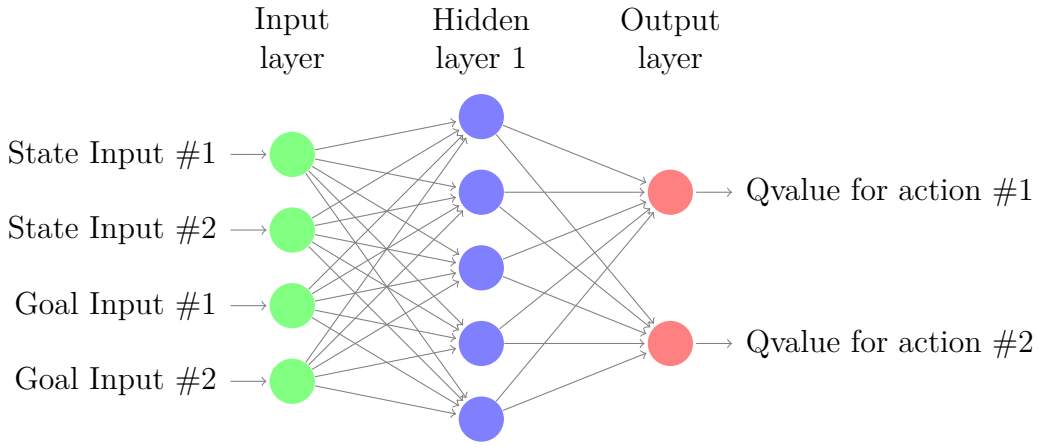


Figure 2.11: Example of a Q-network combined with HER, where main or virtual goals must be provided as input.

must be doubled: current perceived state of the environment as well as the actual goal to follow. An example of the changed Q-network architecture can be observed in figure 2.11.

This means that, for HER, the main goals G of the tasks must be known and inputted to the Q-network. Providing the main goals as an input for the Q-network is, in our opinion, like fraud. The Q-network simply learns that it must reach the state which is equal to the goal input. This is supported by the virtual goals, that reward the learning algorithm with zero when the current action a leads to a state s' which is equal to the goal input. The learning algorithm slowly learns to acquire this behaviour. Another limitation is that the problem to be solved should be divisible into episodes. For example, it is not limiting for tasks, in which the agent might "die", but for continuous tasks, where the agent should never stop to learn, HER is not usable. Another disadvantage is that the input size must be twice the state space size. The resulting Q-network's number of input neurons is increased, which will drastically slow down training and reduce scalability. In addition, HER performs bad in combination with shaped reward functions [AWR+17, p. 8]. Finally, if the main goals are dynamic, which means that they can change during task execution, without adjustments, HER is not applicable.

2.1.10 Exploration

For RL issues, an agent must make two far-reaching decisions: how long should the environment be *explored* and when should it be *exploited*? *Exploration* means that an agent performs actions to gather more information about the task and environment. On the other hand, *exploitation* means that an agent only takes the best action possible from a state. When an environment is under-explored, the agent learns a suboptimal policy that does not or poorly solves the task. However, only exploring one environment results also in a suboptimal policy, due to overfitting of the Q-network or congestion of the replay memory with unprofitable transitions. Finding out when to explore and when to exploit is a key challenge in RL, known as the *exploration-exploitation dilemma*.[SB98, p. 2]

Exploration methods can be divided into two groups: *directed* and *undirected*. [Thr92, p. 3] Directed exploration uses information about the task and/or environment. One of the requirements mentioned in section 1.2 is that the agent should know as little as possible about the task and the environment. For this reason, directional exploration is not used within this thesis. However, undirected exploration methods use no information about environment and task, and thus use random exploration strategies.

2.1.10.1 ϵ-greedy

ϵ-greedy (EG) is a non-greedy exploratory method in which the agent chooses a random action with a probability of $0 \leq \epsilon \leq 1$ at each timestep instead of performing the action with the highest Q-value:

$$\pi(s) = \begin{cases} \text{random action from action space } \mathcal{A}(s), & \text{if } \zeta < \epsilon. \\ \text{argmax}_{a \in \mathcal{A}(s)} Q(s, a), & \text{otherwise.} \end{cases} \tag{2.20}$$

with an uniform random number $0 \leq \zeta \leq 1$ which is drawn at each timestep.

To avoid the *exploration-exploitation dilemma*, ϵ is decreased at each timestep by a fixed scalar number κ. The main challenge is to find the right value for κ.

2.1.10.2 Ornstein-Uhlenbeck process

For physical environment with momentum an *Ornstein-Uhlenbeck process (OU)*, observable in equation 2.21, is usually used as additive noise to enable exploration. This process models the velocity of a Brownian particle with friction [LHP+15, p. 11]. Especially in the case of robot control, such a process is used, due to the drifting behaviour of the output values. The parameters can be set to produce only small drift-like values, which are quite friendly for robot joints. In general, the *Ornstein-Uhlenbeck process* is a stochastic process with medium-reversing properties:

$$dX_t = \theta \cdot (\mu - X_t)dt + \sigma dW_t, X_0 = a \tag{2.21}$$

where θ means how fast the variable reverts towards the mean. σ is the degree of the process volatility and μ represents the equilibrium or mean value and a is the start value of the process, which is usually chosen to be zero.

The Ornstein-Uhlenbeck process can be considered as noise process $\mathcal{N}(a, \theta, \sigma, \mu)$ which generates temporarily correlated noise. This noise is then added to the action to enable exploration, observable in equation 2.22

$$\pi(s) = \pi(s) + \mathcal{N}(a, \theta, \sigma, \mu) \tag{2.22}$$

2.2 Related Work

In this chapter similar articles and works are presented. Although RL is a hot topic these days, finding up-to-date articles which use real physical robots, that learn to solve problems on their own, is a difficult task. For computer programs, dozens of articles and simulation environments exist. For example, the OpenAI Gym website [6] offers more than sixty environments in which learning algorithms can be evaluated and compared to the results of other competitors. In addition, there are not many articles that deal with few sensors and thus with a poor resolution of the environmental state. In this context, no robotic arms, that are used for learning to solve tasks, are meant, of which are countless of articles available.

Since robotic tasks are often associated with complex robot motion models, poor environmental state resolution and sparse rewards, the article [VHS+17] introduces a new DDPG methodology called *Deep Deterministic Policy Gradient from Demonstrations (DDPGfD)*, that should help to solve those issues. The idea is to store a defined number of task solving demonstrations in the replay memory and keep them forever. As replay memory *PPROP* is used. In addition, a new loss function is introduced, which helps to propagate the Q-values backwards along the trajectories [VHS+17, p. 3]. Finally, multiple learning updates per environment step should help to deal with sparse rewards. However, this third modification should be used with caution as it may cause the Q-network to diverge. With these three modifications, environments with very sparse rewards can be successfully solved even with non shaped reward functions.

In this article, *An Adaptive Strategy Selection Method With Reinforcement Learning for Robotic Soccer Games* [SLH+18], researchers from China used Q-learning to learn which strategy small robots should follow in certain situations to successfully play football. Each team consists of four robots and the game state was observed with a camera filming the entire football field. The main issue addressed by this work is that a very dynamic environment, such as soccer with multiple teammates, requires timely and precise decision making [SLH+18, p. 1]. This work is very interesting considering how the strategy selection is learned but compared to our work, the behaviour of the robots is not learned by a machine learning method. In addition, due to the camera, the entire state of the environment is accessible.

A very interesting article, *Control of a Quadrotor With Reinforcement Learning* [HSSH17], introduces the control of a quadrotor with RL methods. The 18-dimensional state vector of the quadrotor includes a rotation matrix, the position, the linear velocity and the angular velocity. The policy is optimized with three different methods: A new optimization algorithm developed by the authors, Trust Region Policy Optimization [SLM+15] and DDPG [LHP+15]. While TRPO and DDPG performed poorly, the algorithm of the authors performed well. Unfortunately, they used a model-based learning approach, therefore their work is not fully comparable to ours.

3 Selection of Algorithms

This chapter introduces an adapted interaction model, in section 3.1, that is required because the reward can not be calculated and granted by the environment through agent's sensors. Followed by an improvements section 3.2, which presents new techniques and improvements done to the state of the art methods with the aim of achieving better results overall. Afterwards, the evaluation setup is introduced in section 3.3. This section contains a description of all important meta parameters. The experiments section 3.4 contains an evaluation of the Q-learning algorithms, replay memories, exploration methods, reward functions and improvements mentioned in this thesis. The performance of these methods are measured in a discrete and continuous action space and state space environment. In addition, the effects of changed meta parameters, such as the *batch size* or the *OU* 2.1.10.2 parameters, are observed. Finally, the best combinations of these techniques are discussed and presented in the measurements section 3.5.

3.1 Adapted Interaction Model

A problem with the standard interaction model, presented in section 2.1.1.3 and observable in figure 2.1, of an agent with its environment is, that the reward is provided by the environment. For computer-written programs, this assumption holds, but for environments in the real world, this model can not be used. Since such environments only provide the state that is captured with agent's sensors. Therefore, the reward must be calculated by the agent itself. Thus, an adapted interaction model, which is used within this thesis, is introduced at this point and can be observed in figure 3.1. The state of the environment is provided to the agent and to the reward function. Then the reward can be calculated by the reward function, based on the current state of the agent. Finally, with this reward, the Q network can learn how to reach the rewarding states of the environment.

Obviously, the motivation quality of the reward function depends heavily on how well the state of the environment is resolved by the agent's sensors. Therefore, choosing the right sensors for the task to be solved can be considered as a key challenge.

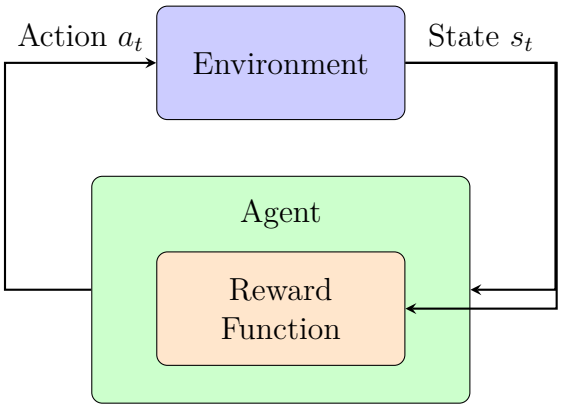


Figure 3.1: Interaction between agent and a real world environment.

3.2 Improvements

In this section, the improvements to the state of the art methods is introduced. Goal of these enhancements is to achieve better results overall. The performance of these changes is then measured in the experiments section 3.4.

3.2.1 Hindsight Experience Replay With Goal Discovery

The idea behind *Hindsight Experience Replay With Goal Discovery (HERGD)* is that the environment's main goal has to be discovered first and only after its discovery it is provided to the Q-network. This approach offers more flexibility than standard HER 2.1.9.3, in which the main goal has to be provided right from the beginning to the Q-network. However, the virtual goals are inserted as defined in standard HER algorithm. Of course, in environments where it is unlikely that the target will be reached with random exploration methods, this approach will struggle the same way as EGC 2.1.10.1 or OU 2.1.10.2. However, once the goal is finally found, this approach can help to get to the optimal policy faster.

3.2.2 ϵ-greedy Continuous

Since ϵ-*greedy*, described in section 2.1.10.1, is only applicable to integer actions, ϵ-*greedy Continuous* (EGC) is introduced. EGC extends the idea of standard ϵ-greedy in order to support continuous action spaces as well. The key idea behind this approach is that every action from the action space $a \in \mathcal{A}(s)$ has its own action range $[a_{lower}, a_{upper}]$. For each action a a random uniform value λ is drawn, for which $a_{lower} \leq \lambda \leq a_{upper}$ holds:

$$\pi(s) = \begin{cases} \text{apply } \lambda \text{ to each action } a \in \mathcal{A}(s), & \text{if } \zeta < \epsilon. \\ \text{take actions from } \pi(s), & \text{otherwise.} \end{cases} \tag{3.1}$$

In order to avoid the *exploration-exploitation dilemma* and to switch from exploring to exploiting, ϵ is decreased at each timestep by a fixed scalar number κ.

3.2.3 Ornstein-Uhlenbeck Annealed

One issue concerning standard *Ornstein-Uhlenbeck process* 2.1.10.2 is that switching from exploring to exploiting is done immediately. This means the process outputting noise until the exploration stops and the exploitation begins. This can harm the learning process, because actions that are already learned optimally can be totally overwritten by the outputted noise. On the other hand, limiting the outputted noise by adjusting θ or σ leads to under exploration. Therefore, the idea behind *Ornstein-Uhlenbeck Annealed (OUA)* $\mathcal{N}_A(a, \theta, \sigma, \mu)$ is to reduce the generated outputed noise after every timestep by a function $f(t)$:

$$\pi(s) = \pi(s) + \mathcal{N}_A(a, \theta, \sigma, \mu) * f(t) \tag{3.2}$$

For this evaluation, the function $f(t)$ is selected to reduce $\mathcal{N}_A$ linearly after every timestep. Of course, more complex reducing functions $f(t)$, such as exponentially reducing the noise $\mathcal{N}_A$, can be designed as well.

3.3 Setup

As already mentioned in section 2.1, a Q-learning method requires other methods in order to work. It consists of an algorithm, an exploration method, a replay memory and a reward function. For simplicity and abbreviation, a four tuple named $\mathbb{RL} \in (\mathcal{M}, \mathcal{E}, \mathcal{RM}, \mathcal{RF})$ method is introduced at this point:

$$\begin{array}{lll} \mathcal{M} & \dots & \text{Q-learning algorithm such as DQN, DDQN, ...} \\ \mathcal{E} & \dots & \text{Exploration method such as } \epsilon\text{-greedy, ...} \\ \mathcal{RM} & \dots & \text{Replay Memory method such as Experience Replay, ...} \\ \mathcal{RF} & \dots & \text{Reward function} \end{array} \tag{3.3}$$

A $\mathbb{RL}$ method includes all the pieces required to work. The Q-learning algorithm $\mathcal{M}$ can be considered as the brain. Learning and decision making is done here. The important parameters are the number of layers and neurons, the activation functions, the optimizer, the learning rate α, the discount factor γ and the soft target update factor τ. τ is only required for *DDQN* and *DDPG* because *DQN* Q-networks do not include a second neural net. The optimizer calculates an update value for the Q-network weights. Applying these calculated values to the weights can be considered as learning. As in the *DDPG* article [LHP+15, p. 11], all Q-networks in this work use the Adam [KB14, pp. 2-3] optimizer. The learning rate α is multiplied by the values computed by the optimizer. Therefore, α determines how fast the Q-network learns. Since some of our evaluations have to deal with sparse rewards, which means that the past experiences are important, the discount factor γ is chosen to be close to *1.0* with $\gamma = 0.98$ The learning rate α and soft target update factor τ, described in section 4.3, is chosen to be small $\alpha = \tau = 0.001$, because solving sparse reward problems requires a lot of transitions, since the rewarding state is not often experienced. *DQN* and *DDQN* use standard feed-forward multilayer neural networks, described in section A, while *DDPG* uses the same architecture as described in section 4.3

and observable in figure 4.15 and 4.16. All hidden layers use *RELU* units as activation function, as they are currently the most successful and widely used [RZL17, pp. 1-2].

The replay memory $\mathcal{RM}$ can be considered as the counterpart to the biological memory. All experiences, in our case transitions, are stored in the replay memory. The most important parameter is the size of the replay memory, which determines how many transitions can be stored. While large memory sizes can only slow down learning, too small memory sizes can drastically reduce learning success or even make it impossible [LZ17, pp. 4-5]. Therefore, this parameter is usually selected to be large enough. For *HER* also the sampling method and the quantity of additional goals k to sample has to be chosen. As sampling method, the *future* method is selected because it gives the best overall results [AWR+17, p. 9]. Since article [AWR+17, p. 9] showed that for the *future* method the best value for parameter k is $k = 4$ or $k = 8$, $k = 4$ is used. For *PPROP* the article [SQAS15, p. 6] mentioned that a good value for prioritization factor α is $\alpha = 0.6$.

The exploration $\mathcal{E}$ method can be considered as curiosity, which inspires us to learn more and more. Here, the sweet spot between *exploration-exploitation* has to be found. To switch from exploration to exploitation, most methods are somehow annealed. If annealing is used in this work, it is done in a linear manner. However, only standard *OU* is not annealed. If the exploration methods were not somehow annealed, but rather switched from exploration to exploitation instantly, which means that after n actions the exploration ends and the exploration begins without any annealing, the learning performance can be harmed drastically. When exploring, it is sometimes necessary to carry out already learned actions in order to be able to further refine them and finally to solve the task in the best way possible. The most important parameter here is the exploration rate e which determines after how many actions performed the exploration ends and the exploitation begins. Of course, it is common practice to place the value for e exactly at the end of an entire learning run $e = \text{episodes} * \text{steps}_{\text{per episode}}$. In addition, for an *OU* process θ, σ and μ has to be chosen. θ describes how fast the output noise returns to the mean, σ is the degree of process volatility and μ indicates the equilibrium or the mean.

Finally, the reward function $\mathcal{RF}$ motivates us to learn by giving a reward for good actions and punishing bad actions. Obviously, this function is crucial and determines if the task is learned or not. Special attention should be given to this function because minor errors can cause the agent to learn a completely different task. This function can be designed *shaped* or *non shaped*. While *non shaped* reward functions usually provide only a positive reward for the main goal of the task, *shaped* reward functions are usually designed to guide the learner towards the main goal.

The evaluations are carried out with Keras [5] framework, which is programmed with Python [11]. Keras is used to model neural networks and runs on top of the symbolic math libary TensorFlow [9]. All global parameter settings used in all evaluations are summarized in table 3.1.

	Q-learning method $\mathcal{M}$	
General	Optimizer	Adam
	Learning rate α	0.001
	Discount factor γ	0.98
	Hidden layer 1 neurons	64
	Hidden layer 2 neurons	128
	Hidden layer activation function	*RELU*
	Batch sizes	16, 32, 64, 128
DQN & DDQN	Output layer activation function	*Linear*
DDPG	Learning rate actor	0.001
	Learning rate critic	0.001
	Output layer activation function	*Tanh*
	Replay memory $\mathcal{RM}$	
General	Size	500000
PPROP	Prioritization factor α	0.6
HER	Sample method	*future*
	Additional goals	4
	Exploration method $\mathcal{E}$	
ϵ-greedy	Start value ϵ	1.0
	End value ϵ	0.1
	Annealing	*linearly*
Ornstein-Uhlenbeck	θ	0.15
	μ	0.0
	σ	0.2

Table 3.1: Global parameter settings for all evaluations.

3.4 Experiments

In this section the state of the art methods, mentioned in chapter 2, and improvements, mentioned in section 3.2, are evaluated. All experiments use the global parameter settings presented in section 3.3. To get an idea of which reward function, whether shaped or not, works well, all environments are evaluated with a *shaped* and *non shaped* reward function. In addition, several *minibatch* sizes are tried out in order to find a meaningful value.

Firstly, various parameter settings of an *OU* and *OUA* process are evaluated in order to find meaningful values. Secondly, the evaluations are performed in a *bit-flip environment*, presented in section 3.4.2, which itself consists of a discrete state and action space. This is followed by a short discussion about the results, observable in section 3.4.2. Thirdly, to measure the performance in a continuous state and action space environment

the evaluations are performed in the *pendulum environment*, introduced in section 3.4.3. This is followed by a short discussion about the results, observable in section 3.4.3. Finally, a measurements section 3.5 discusses and presents the $\mathbb{RL}$ methods performed best.

3.4.1 Ornstein Uhlenbeck Process Parameter Determination

In this section several different parameter settings for an *OU* and *OUA* process, introduced in sections 2.1.10.2 and 3.2.3, are evaluated in order to find meaningful values. Observing the impacts of the various settings will help to determine which values are usful for certain environments. In addition, since exploration is very important for successful learning a wrong understanding of this settings hinders task solving performance.

All Q-networks in this thesis use an output range of $[-1.0; 1.0]$. Therefore, the *OU* parameters are determined for that range. As already known from section 2.1.10.2, an *OU* process consists of four parameters to adjust: θ, σ, μ and a. Parameter μ, which represents the mean value, and a, which represents the starting value of the process, are set to *zero*. Parameter θ, which indicates how fast the process reverts towards the mean and σ, which determine the maximum volatility, must be found. If θ is chosen to be very large, the process will produce values close to the mean μ. On the other hand, if θ is chosen to be very small, the output drifts in the direction of *+1* or *-1* and only returns to μ after a very long time. Parameter σ indicates how fast the output can drift. Selecting σ to be very large will generate a lot of values close to *1* and *-1*. However, if σ is chosen to be very small, the drift speed is impaired and values close to μ are generated.

The evaluation only records the noise $\mathcal{N}(a, \theta, \sigma, \mu)$ and $\mathcal{N}_A(a, \theta, \sigma, \mu)$ generated by the *OU* or *OUA* process. Two graphs are generated per evaluation. One graph shows the output of the process over time while the other one counts how many times a noise value occurs. Therefore, the range $[-1.0; 1.0]$ is discretized with a step size of *0.001*. Then it is counted how often an output value occurs in a certain discretization level. This process is executed for *50* times and finally the average is calculated.

Results

Figure 3.2 show the evaluations of the standard *OU* process with different settings for θ and σ. In the first figure ($\theta = 0.15$ and $\sigma = 0.3$), it can be observed that setting θ smaller than σ will cause the process to output a lot of values near to the range boundries. This can be useful for agents where abrupt control of the actuators is required. For example, if a robotic arm is used to control a heavy mass object, abrupt controlling can be useful [RNS13, p. 1]. The next graph ($\theta = 0.3$ and $\sigma = 0.2$) shows the opposite of the first one. Here θ is set to be greater than σ. This results in many output values being close to zero. For agents, where fine steering is necessary, these settings can be used. Finally, the third graph ($\theta = 0.3$ and $\sigma = 0.3$), shows what happens if θ is set to be equal to σ. In this case,

the output values are concentrated close to zero and at the boundaries of the range. This setting may be useful for environments where the entire output range must be approached.

Finally the evaluations of the *OUA* process, introduced in section 3.2.3, is shown in figure 3.3. Especially in the second ($\theta = 1.0$ and $\sigma = 0.5$) and third evaluation ($\theta = 0.5$ and $\sigma = 1.0$) a linear decreasing of the output values can be observed. Unfortunately, since an *OU* process uses the previous outputted value, the output value increases over some time again. This behaviour, which is contra productive, since it was planned to decrease the outputted noise slowly to enable soft switching from exploring to exploiting, can be recognized best in the third evaluation ($\theta = 0.5$ and $\sigma = 1.0$). Therefore, the *OUA* process can not be used. The proposed settings for different controlled agents for θ and σ is summarized in table 3.2.

Control of the agent		
fine control	abrupt control	entire control range
$\theta > \sigma$	$\theta < \sigma$	$\theta \simeq \sigma$

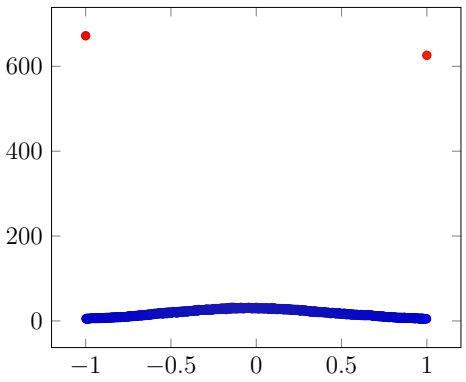
Table 3.2: Summary of OU parameters for various controlled agents.

3.4.2 Bit-flip Environment

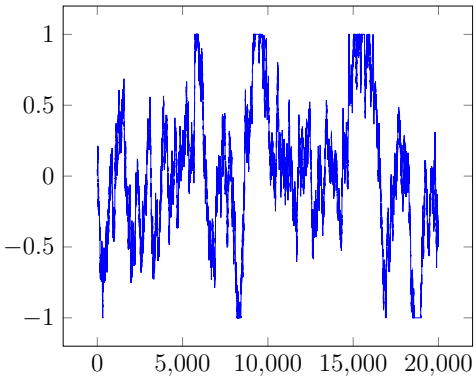
The basic idea of this environment is based on the *bit-flip environment* from the *HER* [AWR+17, pp. 3-4] article. However, minor adjustments had to be made to use this environment for this thesis. Basically, the goal of the bit-flip environment is to flip the bits in a bit vector BV_n of length n in the same way that it matches a target bit vector BVT_n within n tries. The state $S = \{BV_{n_0}, BV_{n_1}, \ldots, BV_{n_{n-1}}\}$ of this environment is the bit vector $BV_n = S$. The action a is the number of the bit to flip and can be chosen from the action space $A = \{0, 1, \ldots, n-2, n-1\}$. For example, an action $a = 2$ means that the bit BV_{n_1} in bit vector BV_n is flipped. To make the algorithms and methods comparable, bit vector BV_n is always reset to zero and the target bit vector BVT_n is taken from a look-up table.

Since *DDPG* outputs continuous values and the bit to be flipped must be an integer number, a conversion from real to natural numbers has to be made. Therefore, the output of *DDPG* is divided into n equal sections, each representing one bit in the bit vector. In addition, *DDPG* does not use an *OU* process for exploration, because the drift-like behaviour of the output values is indeed good for robotic control, but not for a discrete action space such as the bit-flip environment has. Thus, we only used continuous ϵ-greedy exploration for *DDPG*.

This environment uses equation 3.4 as non shaped reward function. The reward for reaching the goal is set to *1* and all other rewards are set to *-1*. This simulates a delayed and sparse rewards problem, since the probability of finding the target bit vector

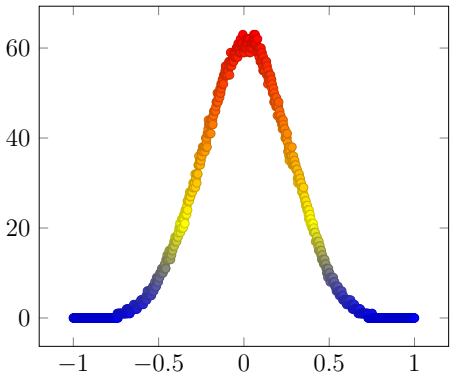


(a) Counted output values.

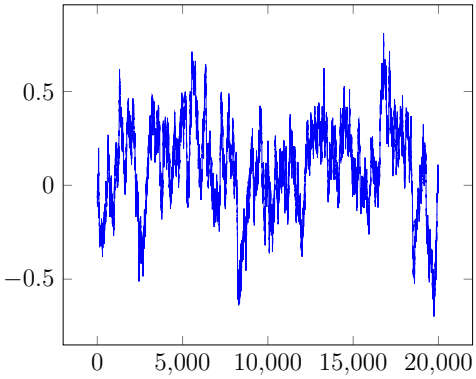


(b) Output value over time.

Evaluation for $\theta = 0.15$ and $\sigma = 0.3$.

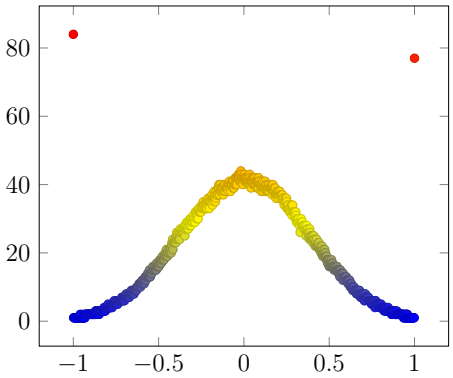


(c) Counted output values.

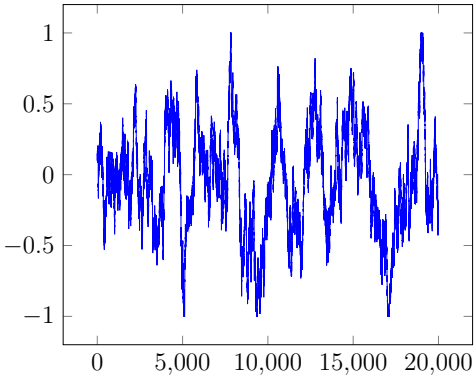


(d) Output value over time.

Evaluation for $\theta = 0.3$ and $\sigma = 0.2$.



(e) Counted output values.



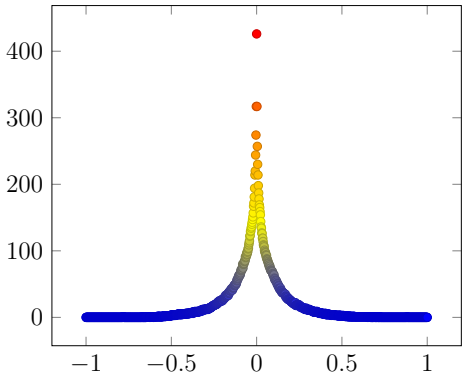
(f) Output value over time.

Evaluation for $\theta = 0.3$ and $\sigma = 0.3$.

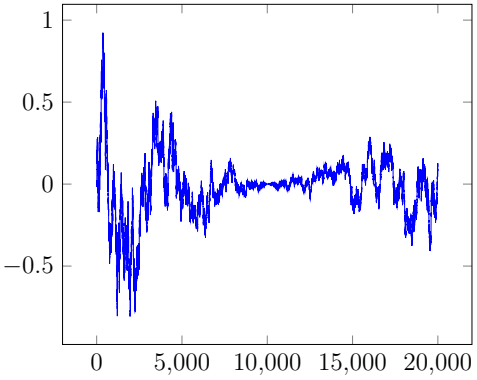
Figure 3.2: Evaluations for the Ornstein-Uhlenbeck (OU) process.

BVT_n is drastically decreasing with the bit vector length n.

$$\mathcal{RF} = \begin{cases} +1, & \text{if } BV_n = BVT_n. \\ -1, & \text{otherwise.} \end{cases} \tag{3.4}$$

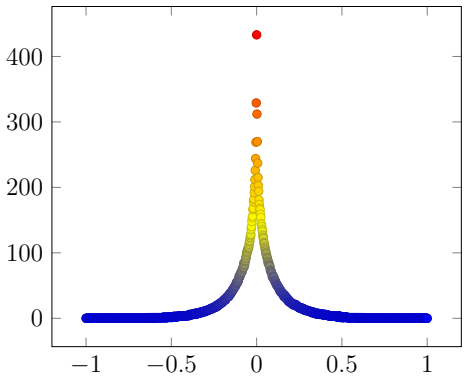


(a) Counted output values.

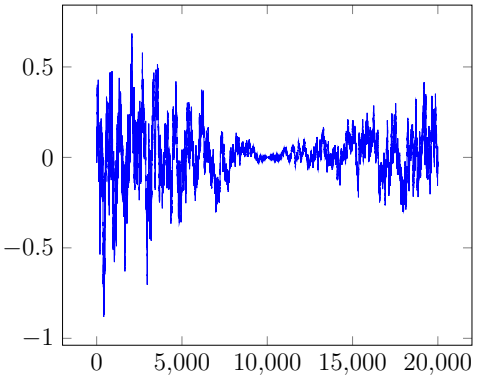


(b) Output value over time.

Evaluation for $\theta = 0.3$ and $\sigma = 0.3$.

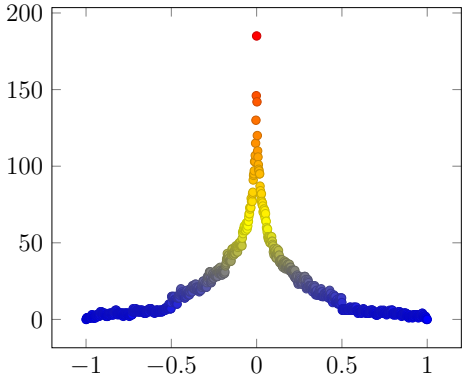


(c) Counted output values.

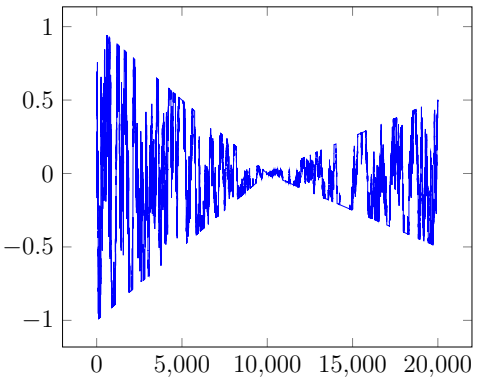


(d) Output value over time.

Evaluation for $\theta = 1.0$ and $\sigma = 0.5$.



(e) Counted output values.



(f) Output value over time.

Evaluation for $\theta = 0.5$ and $\sigma = 1.0$.

Figure 3.3: Evaluations for the annealed Ornstein-Uhlenbeck (OUA) process.

The shaped reward function, observable in equation 3.5, simply counts the equal bits of bit vector BV_n and target bit vector BVT_n. Then the counted number is divided by $n-1$.

$$\mathcal{RF} = \begin{cases} +1, & \text{if } BV_n = BVT_n. \\ count_n(BV_n = BVT_n)/(n-1), & \text{otherwise.} \end{cases} \tag{3.5}$$

The evaluation starts with a bit vector length of $n = 1$. At the end of each episode, which is exactly after *200* bit flips, it is checked if the current $\mathbb{RL}$ method solves the bit vector with length n within n bit flips. If so, the current attempt t is considered as successful. An attempt t is assumed to be failed, if a $\mathbb{RL}$ method fails to solve a certain bit vector within *50* episodes. After determining whether the bit vector length n has been solved successfully or the attempt has failed, the $\mathbb{RL}$ method is reset and the next attempt $t = t+1$ is started. After *5* tries, it is checked if the current $\mathbb{RL}$ method has at least one successful attempt. If this is the case, the next bit vector of length $n = n + 1$ can be performed, otherwise the next $\mathbb{RL}$ method is selected and the evaluation starts again with a bit vector length of $n = 1$.

The success rate of a $\mathbb{RL}$ method can be calculated by dividing the total successful attempts by the number of all attempts. In addition, the average training times and the average bit flips, ranging from *1* to a maximum of *1000*, which are needed to solve a bit vector length n, are measured.

Results

First the evaluation results of the bit-flip environment with a *non shaped* reward function are discussed. The success rate and average training time of this evaluation can be observed in figure 3.4 and 3.5. All methods suffered from the fact, that with raising bit vector lengths n the target bit vector BVT_n is more difficult to discover. In addition, with random exploration methods it could happen that one method experiences the target bit vector more often than the others. Only *HER* has the advantage that the Q-network knows the goal state of the environment right from the beginning. *HERGD* first has the same discovery issue as *PPROP* and *EXPR* until the goal is experienced once. In general, it can be said that using larger *batch sizes* in combination with *HER*, *HERGD* or *PPROP* really helps when dealing with sparse rewards. Comparing *DQN* with *DDQN*, it can be observed that *DDQN* solves more consistently a bit vector length n, even with smaller *batch sizes*. One reason that *DDQN* behaved in this evaluation pretty much like *DQN* is the low number of episodes and bit flips to solve for a bit vector length n. Since the target network used by *DDQN* is updated only slowly to avoid divergence, this method requires more training steps than *DQN*. Unfortunately, due to the low computing power no higher number of episodes or game steps could be executed. Although the *DDPG* results may not seem promising at first glance, they are unexpectedly good. For *DDGP*, the environment was even more difficult to solve because a continuous action space has to be searched, while *DQN* and *DDQN* only have to search a discrete action space with length n. Considering the average training time, observable in figure 3.5, it can be said, that higher *batch sizes* require more training time. In addition, methods using *PPROP* or *DDPG* require a lot more training time. One reason is that the *DDPG* Q-network architecture is more complex than the others since it consists in total of four neural nets. *PPROP* internally uses a sum-tree to store transitions, which lengthens training times.

The evaluation results of the bit-flip environment with a *shaped* reward function can be observed in figure 3.6. First, it can be noticed that *DQN* and *DDQN* were

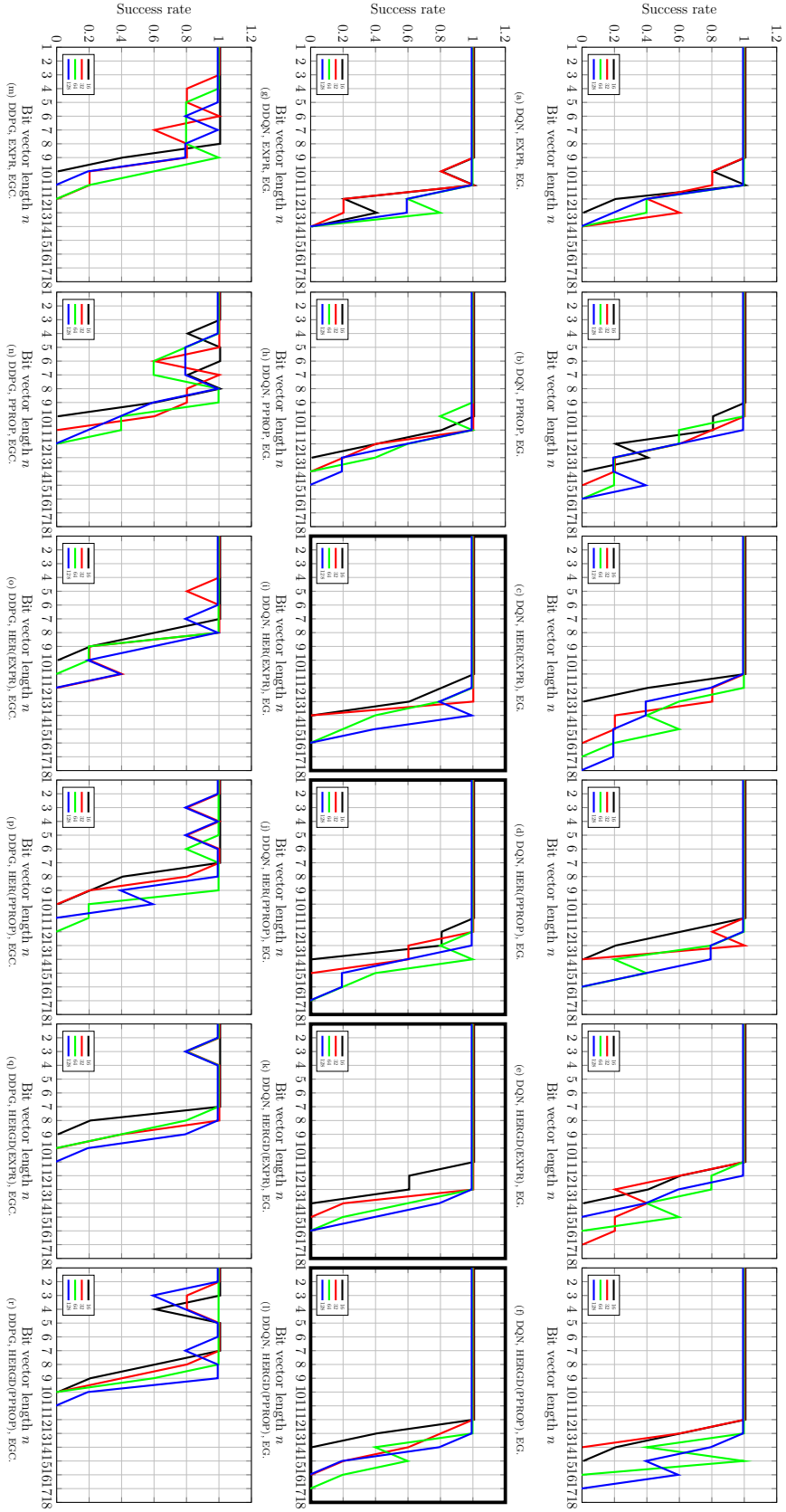


Figure 3.4: Success rate of the bit-flip environment with non shaped rewards.

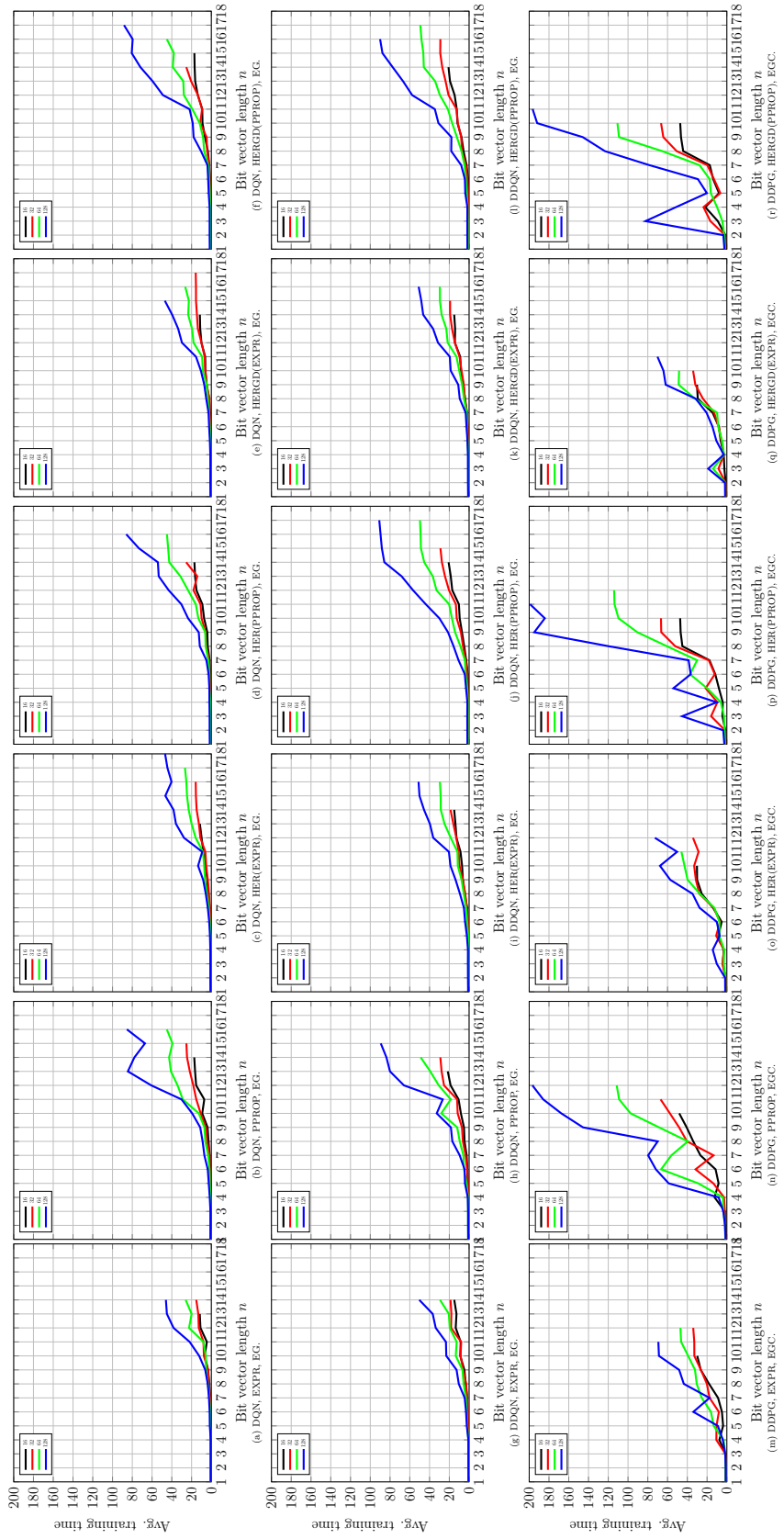


Figure 3.5: Average training times of the bit-flip environment with non shaped rewards.

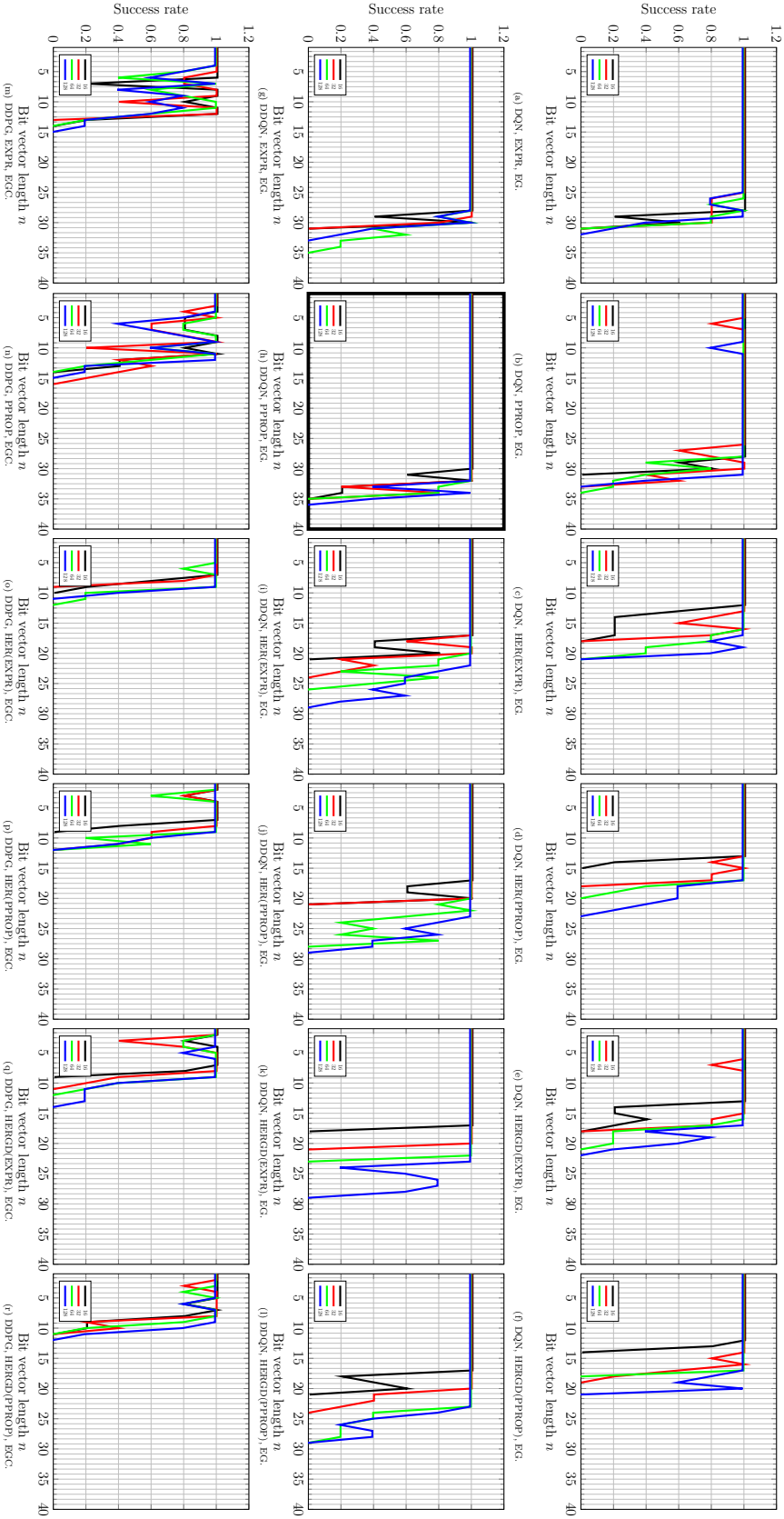


Figure 3.6: Success rate of the bit-flip environment with shaped rewards.

able to solve a much larger number of bit vector lengths n. Reward shaping drastically improved the task solving performance. Only *DDPG* performed not much better with *non shaped* rewards. Comparing *DQN* with *DDQN*, it can be seen that *DDQN* solves more consistently a bit vector length, even with smaller batch sizes, as it was the case with the *non shaped* reward function. *HER* and *HERGD* performed very similar, but worser than the *EXPR* or *PPROP*. That is not suprising, since the *HER* paper [AWR+17, p. 8] mentioned that it performs bad with *shaped* rewards. *PPROP* performed a little bit better than *EXPR*. The average training times do not differ much from the *non shaped* ones and will not be listed here anymore.

3.4.3 Pendulum Environment

To be able to evaluate the $\mathbb{RL}$ methods within a continuous action and state space environment the pendulum environment from open AI Gym [7] was chosen. The goal of this environment is to swing a frictionless pendulum upright, so that it stays vertical, pointing upwards. Image 3.7 shows the pendulum near to the maximum reward position. The perceived state S of this environment is a three tuple $S \in (cos(\phi), sin(\phi), \dot{v})$. This state is generated by the pendulum angle ϕ and vertical velocity $\dot{v}$ of the pendulum. To change pendulum's state a torque $-2 \leq \tau \leq 2$ can be provided as an action a. However, to make this environment more difficult to solve, the torque is limited to $-1 \leq \tau \leq 1$ in our evaluations. Therefore, the pendulum must gain velocity through swinging to reach the rewarding position. After a reset, the pendulum starts in a random position and with a random torque.

Since *DQN* and *DDQN* output for discrete action spaces a conversion from natural numbers to real numbers has to be made. The output layer of *DQN* and *DDQN* has been extended to *21* nodes. Each of them represents a specific torque value, starting from *-1.0*, with a step size of *0.1*, to *1.0* including zero.

For *shaped* rewards this environment uses equation 3.6 with $-\pi \leq \phi_{\text{norm}} \leq +\pi$ and $-8 \leq \dot{v} \leq +8$. Therefore, the reward is in range of $-16.27 \leq R \leq 0$.

$$\mathcal{RF} = -(\phi^2_{\text{norm}} + 0.1 * \dot{v}^2 + 0.001 * \tau) \tag{3.6}$$

For *non shaped* rewards equation 3.7 is used. The reward is set to *1* if the pendulum points upwards and its angle ϕ is in range of $-1° \leq \phi \leq 1°$. If not, the reward is set to *-1*.

$$\mathcal{RF} = \begin{cases} +1, & \text{if } -1° \leq \phi \leq 1°. \\ -1, & \text{otherwise.} \end{cases} \tag{3.7}$$

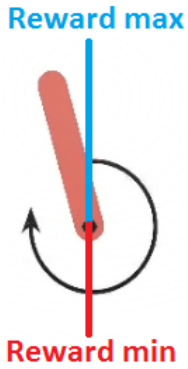


Figure 3.7: Image of the pendulum environment with the pendulum position near the maximum reward position.

Each tested $\mathbb{RL}$ method has exactly *250* episodes to solve the pendulum environment. Each episode consists of *300* steps. At each step, an action is predicted by the Q-network and applied as torque to the pendulum. After every *10th* episode, the learning success is tested. For this purpose, the learned policy $\pi(s)$ is used over *20* episodes and the received rewards are summed up. Then the mean value is calculated from the sum of rewards. In addition, the standard deviation is computed and presented as a transparent background in the graph.

Results

The evaluation results for the *shaped* reward function can be observed in figure 3.8. Since *HER* and *HERGD* in combination with shaped reward functions, already mentioned in Section 3.4.3, performed poorly, these evaluations are discarded on this point. *DDPG* performed very well, while *DQN* and *DDQN* did very poorly in comparison. Suprisingly, *EXPR* delivered a little bit better results than *PPROP*. Concerning the fact that the dimension of a discrete state space is countable, this is not the case for a continuous state space. *PPROP* prioritizes the transitions that are new or surprising. For continuous state spaces, this almost applies to every transition. Especially this affects the learning performance for small *batch sizes*. For the exploration methods it can be said that *EGC* performed better than *OU*. That is because the behavior of *OU* is drift-like. In the pendulum environment it is better to switch the torque repeatedly from positive to negative values and back again. This behavior increases the speed and allows the pendulum to move to a vertical position. Finally, considering the average training time diagrams, it can be said, that *PPROP* increases the training times drastically.

The evaluation results for the *non shaped* reward function can be observed in figure 3.9. In general it can be said that the task solving performance was quite bad. Again, as it was the case in the bit-flip environment 3.4.2, it can be seen that with *non shaped* rewards the task is much harder to solve. Only, *DDQN* and *DDPG* in combination with *EXPR* as replay memory and *EG* or *EGC* as exploration method somekind managed to

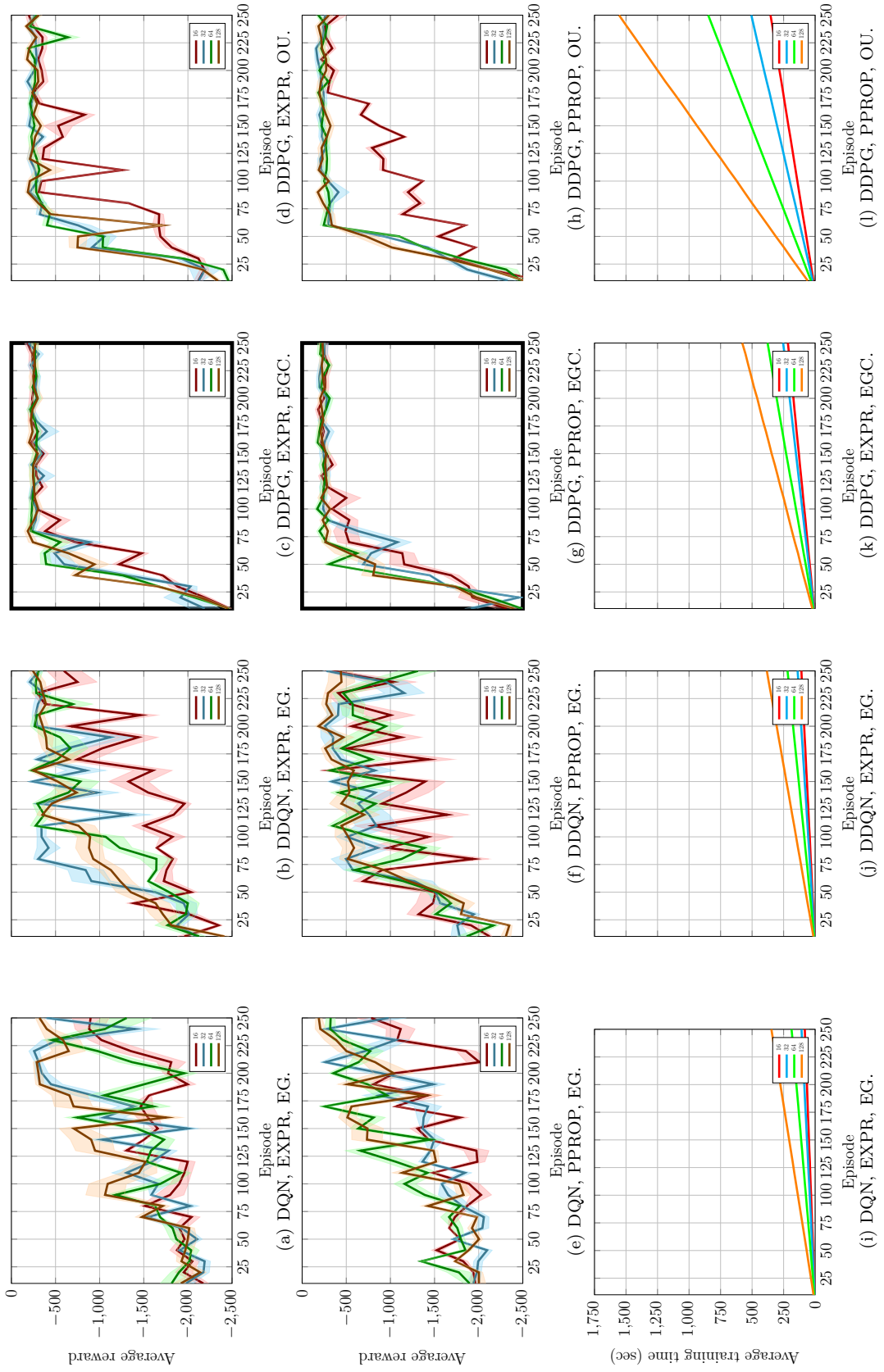
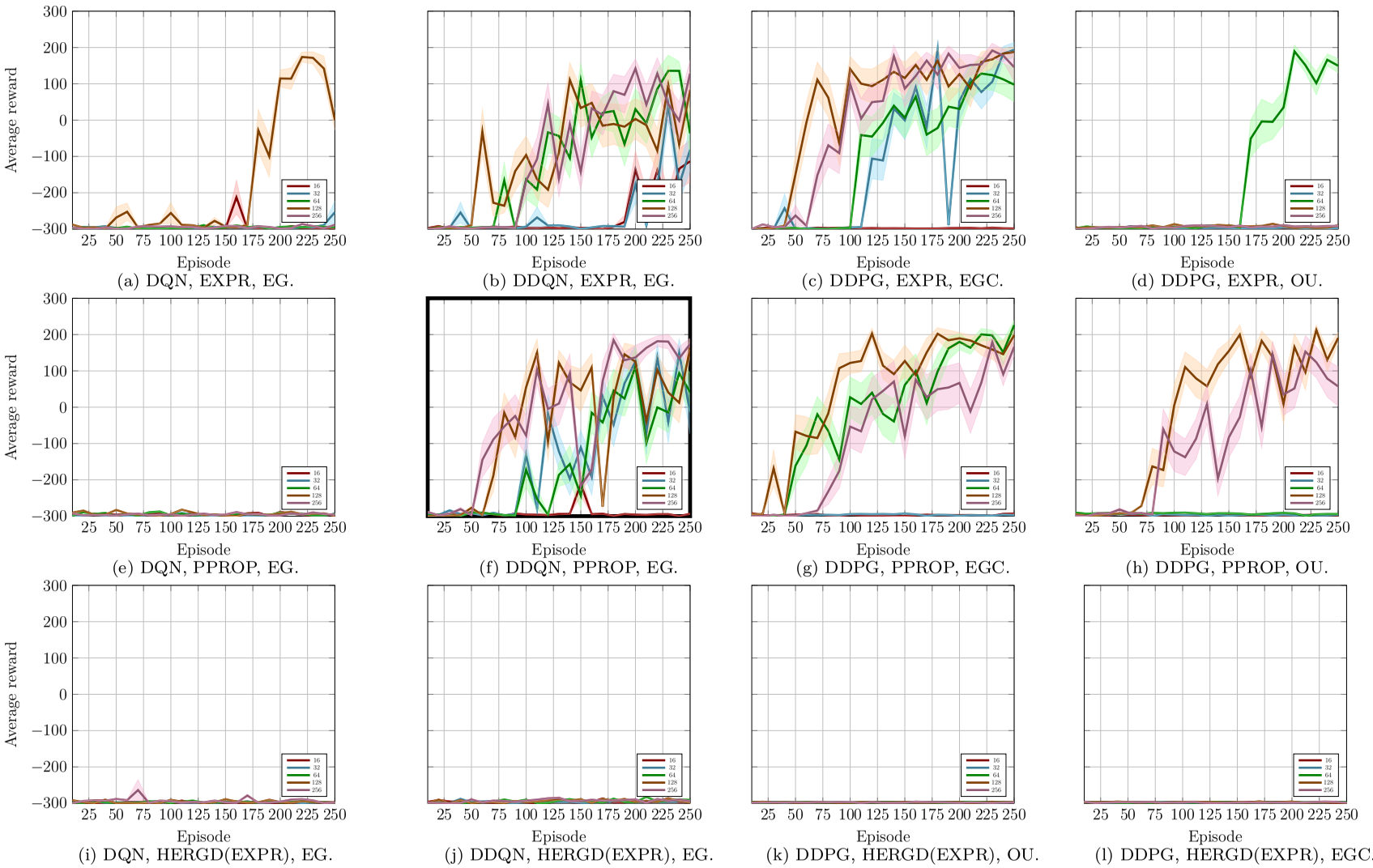


Figure 3.8: Evaluation of pendulum environment with shaped rewards.



(a) DQN, EXPR, EG. (b) DDQN, EXPR, EG. (c) DDPG, EXPR, EGC. (d) DDPG, EXPR, OU.

(e) DQN, PPROP, EG. (f) DDQN, PPROP, EG. (g) DDPG, PPROP, EGC. (h) DDPG, PPROP, OU.

(i) DQN, HERGD(EXPR), EG. (j) DDQN, HERGD(EXPR), EG. (k) DDPG, HERGD(EXPR), OU. (l) DDPG, HERGD(EXPR), EGC.

Figure 3.9: Evaluation of pendulum environment with non shaped rewards.

perform acceptable. The *HER* delivers quite the same result as *HERGD* and is therefore omitted. *HERGD* was not able to solve the environment at all. In continuous state space environments, the main goal can only be defined within a range. In addition, since the state of the pendulum environment consists of the vertical velocity, goal discovery is very bad since the goal is discovered with an non zero velocity. Of course this is hindering, since the goal is to keep the pendulum upright in a vertical position. In the other cases, where *HERGD* is not used, it can be seen that larger *batch sizes* will help to solve the task, but if the rewarding goal of the environment is not experienced often, a lot of non-task-relevant transitions are stored in the replay memory hindering successful learning.

3.5 Measurements

This chapter introduced the adapted interaction model in section 3.1 that is required because the reward can not be perceived by agent's sensors. Subsequently, improvements of the state of the art methods were presented in section 3.2, with the aim of achieving better results. Afterwards, the evaluation setup was introduced in section 3.3, which contains a description and the value set of all important meta parameters. Finally, in the experiments section 3.4 the state of the art methods and improvments are evaluated against the others using different enironments.

Based on the evaluations done with the bit-flip environment and pendulum environment the best combination of the $\mathbb{RL}$ methods can be presented in this section. For discrete action spaces, it is strongly recommended to use *DDQN*, as it works better in such environments than *DQN* and *DDPG*. Obviously, in all cases a *shaped* reward function should be used, since it drastically improves the learning performance. For continuous state space environments it is recommended to use *EXPR* as replay memory, because it performs quite same as *PPROP* and *HERGD*, but requires less training time. On the other hand, for discrete state space environments, where a *non shaped* reward function is used, it is recommended to use *HERGD* in combination with *PPROP*. This will help to successfully solve sparse reward environments. For the *batch size* it is recommended to choose a larger value depending on how sparse the rewards are. The selection of the exploration method has to be tuned based on the environment. For a robotic environment where the decision maker is an robotic arm, *OU* is recommended. For environments, such as the pendulum environment, where drift-like behaviour of the output values is not good, *EGC* is recommended. A summary of the best $\mathbb{RL}$ method combinations can be observed in table 3.3.

		Environment			
		cont. action space cont. state space	dis. action space cont. state space	cont. action space dis. state space	dis. action space dis. state space
Reward function	non shaped	DDPG, EXPR	DDQN, EXPR	DDPG, HERGD(PPROP)	DDQN, HERGD(PPROP)
	shaped	DDPG, EXPR	DDQN, EXPR	DDPG, EXPR	DDQN, PPROP

Table 3.3: Summary of the best $\mathbb{RL}$ method combinations.

4 Simulation

In this section the agent and the environment in combination with the selected $\mathbb{RL}$ method from chapter 3 are simulated. The goal is to ensure that the chosen $\mathbb{RL}$ method operates on the final hardware without major changes. Therefore, agent and environment should be modeled as realistically as possible. Of course, this includes all connected sensors and movements of the agent. Furthermore, special emphasis is placed on the model of the charging station. The simulation environment should be able to model problems as realistically as possible, be fully 3D, open for academic use, easy to learn and use. Unity [12] was chosen because it easily meets all these requirements. Unity is a cross-platform game engine developed by Unity Technologies. It is delivered with an editor and the primary programming language is C# [10].

This chapter starts with a description, in section 4.1, how the communication between agent and the $\mathbb{RL}$ method works. Followed by an introduction of the modelling done in Unity. Then the architecture of the chosen $\mathbb{RL}$ method and the used meta parameters are presented in section 4.3. Finally, with meaningful diagrams and measurements, the performance of the chosen $\mathbb{RL}$ method, in combination with agent and environment, are measured and presented in the evaluation section 4.4.

4.1 Communication

As mentioned in section 1, the SLES is a very small and resource constraint embedded system. Thus, the memory of the attached processor is very limited. Without a drastic and time-consuming reduction of the required memory size of the $\mathbb{RL}$ method used, it would not fit into the memory of the processor. Particular the memory requirement of the Q-network and replay memory should be reduced because they consume most of the memory. This issue may be addressed in the future. Therefore, the $\mathbb{RL}$ method is executed on an external host that communicates with the agent via an UDP interface. For the final hardware, this UDP interface is replaced with a Bluetooth interface. Therefore, the communication architecture, including the protocol and interface, is designed for simple exchangeability. The communication architecture can be observed in figure 4.1. For sake

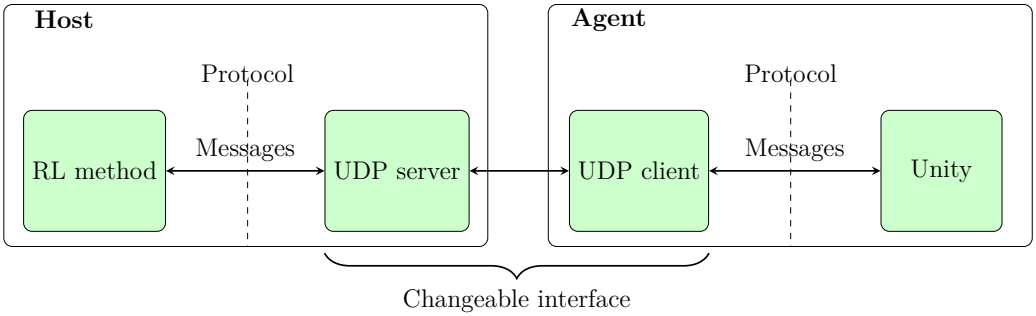


Figure 4.1: Communication architecture.

of simplicity, a blocking communication interface is chosen. That means, the recipient of a message waits until the entire message is received, even if this would take forever. This ensures, that the Q-learning algorithm and the agent are synchronized and that no message is dropped. If one message would not reach the recipient, the execution of both programs would be blocked.

Figure 4.2 visualizes the communication sequence in a sequence diagram. First, the agent sends its perceived state to the host station, which predicts the next action to perform. This action is then sent back to the agent for execution. Obviously, this process is repeated for the entire learning process. In order to ensure that the agent's processor

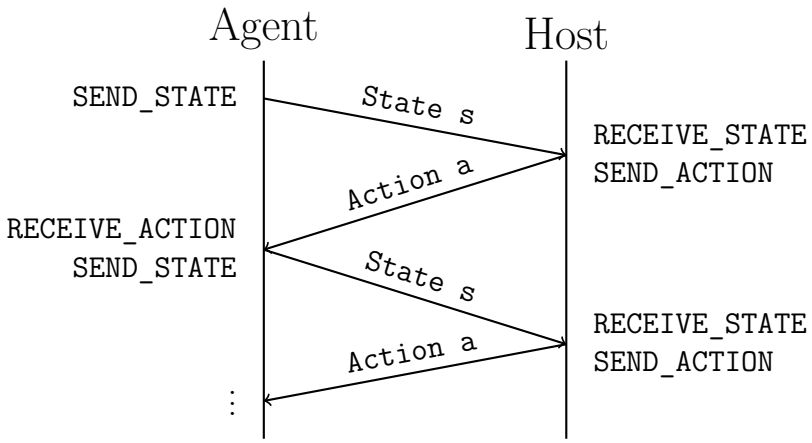


Figure 4.2: Communication sequence diagram.

does not have to spend a lot of time computing the message sent or received, it has been designed with performance in mind. Therefore, each state or action entry is sent as a 32-bit integer.

4.2 Unity Models

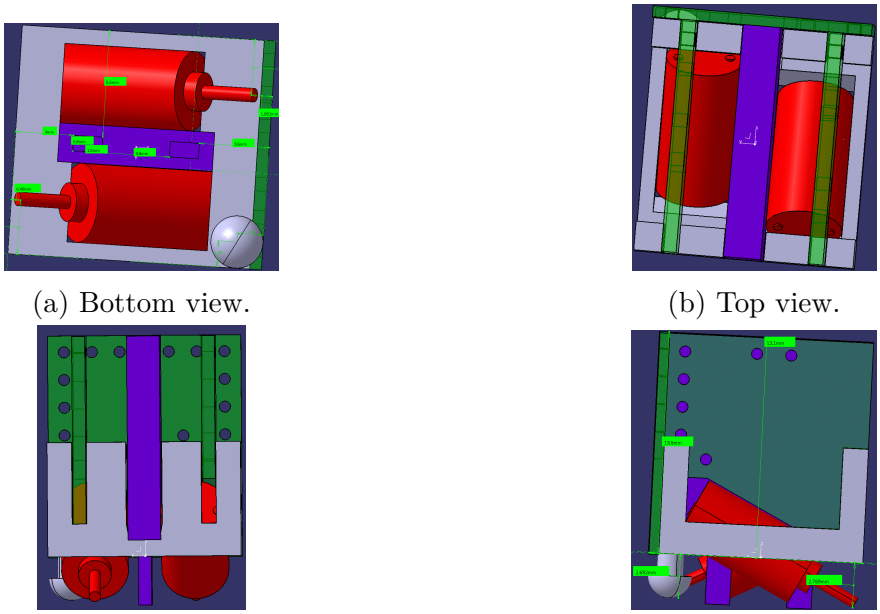
Unity allows to generate quite precise models of real physical objects. Large models can be built by composing smaller sub-models or components. This improves the maintainability and interchangeability of existing components and models. That is why the models in Unity scale very well. Furthermore, Unity also takes many physical parameters, such as gravity, friction and much more, into account. For sub-models which are connected through physical joints, Unity contains joint components. Since the real agent is not yet built, some parameters of the model had to be estimated, but with the future perspective of designing it more and more accurately. Important to note is, that every component: sensors, charging station, actuators, ...; has its own model and script. A script describes the behaviour of the model to which it is attached.

This section describes how the real components work and how they were approximated by models in Unity. First, the model of the agent is presented in section 4.2.1. To measure the performance of the agent's model, a simpler model is also presented in this section. Followed by a description of all sensor models used. Finally, the model of the charging station and environment is introduced.

4.2.1 Robot Model

The construction of the agent can be viewed in figure 4.3. In the bottom view, figure 4.3a, the two actuators and the charging pads (the two rectangles in the middle) can be seen. In addition, the standing pad, located in the bottom right of the bottom view figure 4.3a, which is the third contact point of the agent with the ground, can be observed. This third contact points guarantees stable, not tilting, movement of the agent. These just mentioned components can be observed easier with the help of the front view, figure 4.3d, and the left view, figure 4.3c. The positions of the sensors are not drawn on this building plans.

The Unity model of the agent can be observed in figure 4.4. This first, very simple model of the real agent aims to model the motion and sensor positions as accurately as possible and not the appearance. However, due to the fact, that the robot is not built yet, a lot of parameters had to be estimated. First and foremost, this includes the friction parameter of the actuator's contact point with the ground and the torque that can be transmitted. For simplicity, the standing pad is modelled without friction. The contact points of the actuators with the ground are modelled as simple black spheres, best observable in the bottom and front views of the figures 4.4a and 4.4b. On these spheres, a torque can be applied. As the motor positions are offset longitudinally, this model suffers from sliding. Therefore, forward and sideways frictions are added to the spheres to simulate slippery movements. The actuators are controlled by two vectors. A vector describes the torque applied to an actuator. In the front view figure 4.4b the infrared receiver (large grey rectangle) and the range finder (small grey rectangle) can be seen.



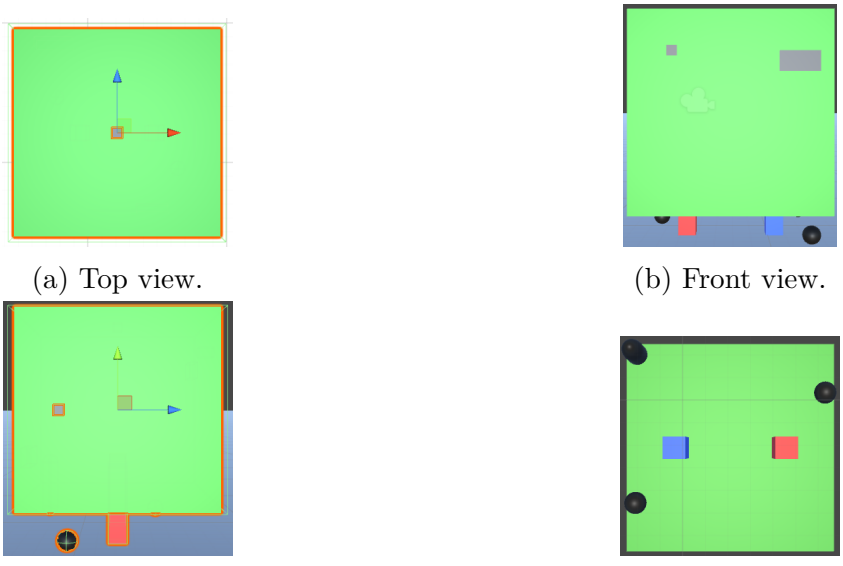
(a) Bottom view. (b) Top view.

(c) Left view. (d) Front view.

Figure 4.3: SLES construction.

The right view figure 4.4c shows another range finder. The infrared receiver in the back view is omitted at this point. The charging pads, red and blue rectangles, can be seen best in the bottom view of the figure 4.4d.

Since the real agent suffers from some impairments such as complex movement due



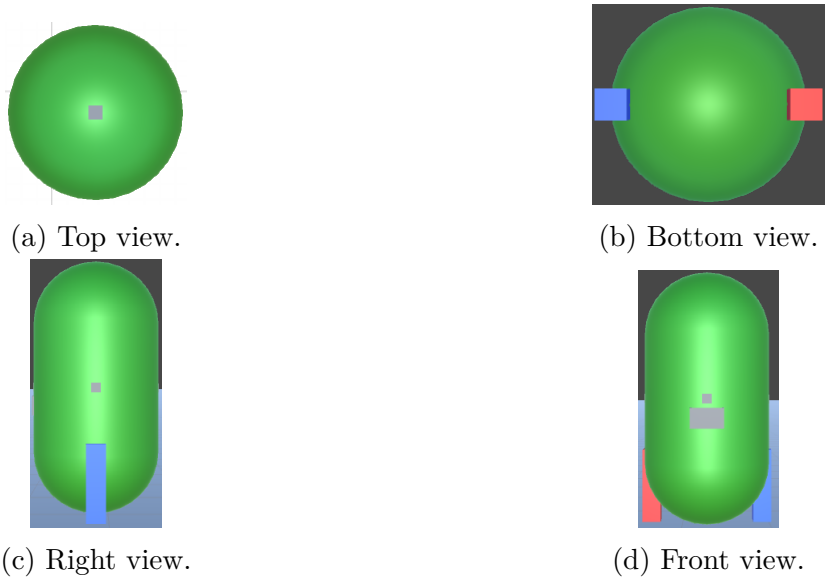
(a) Top view. (b) Front view.

(c) Right view. (d) Bottom view.

Figure 4.4: Unity model of the agent.

to actuator positions, slippery motion, displacement of the sensors and jamming with

walls due to the rectangular construction, a simpler model is additionally implemented. In thesis this model is called simple agent. This simple agent is used for comparison with the model of the real agent. In addition, the simple agent consists of a frictionless and simpler movement model. It is controlled by two vectors. One vector controls the forward or backward movement while the other one is used for turning right or left. The forces are directly applied to the model. In order to make the charging stations more accessible to the simple agent, it is chosen to be round. For better orientation in the state space, four range finders, instead of two, are attached to it. The model of the simple agent can be observed in figure 4.5.



(a) Top view.

(b) Bottom view.

(c) Right view.

(d) Front view.

Figure 4.5: Unity model of the simple agent.

4.2.2 Range Finder Model

To estimate the distance between the agent and an object, Time-of-Flight sensors (TOF) are used. These sensors emit a light pulse that travels to the nearest object and is reflected by it. The flight time of the light pulse, between emitting and returning to the sensor, is measured. Finally, with the help of the flight time, the distance to that object can be estimated.

The range finder model in Unity, observable in figure 4.6, is working similar to the describtion in the previous section. The modelled sensor emits a light ray and the collision of the light ray with the nearest object is measured. The time does not have to be measured because with Unity the light ray emit position and the collision position can be calculated.

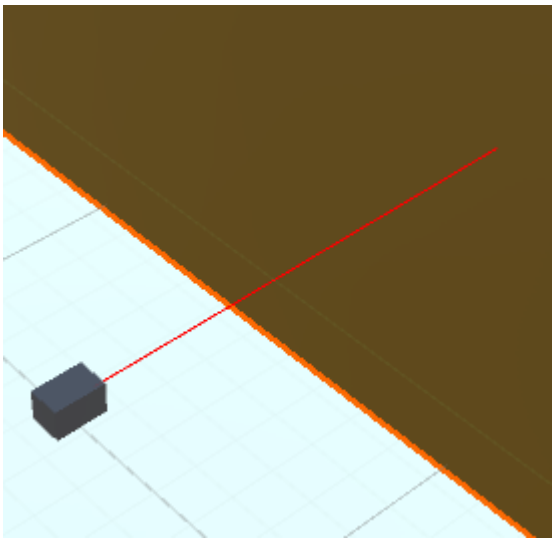


Figure 4.6: Range finder model in Unity.

4.2.3 Infrared Led Model

In order that the agent is able to find the position of the charging station, in particular the position of the charging pads, it is equipped with an infrared LED [3]. The infrared led consists of a narrow directivity and a high brightness. In figure 4.7, the scanning angle and narrow directivity can be seen.

The infrared LED model in Unity, observable in figure 4.8, consist of a sphere

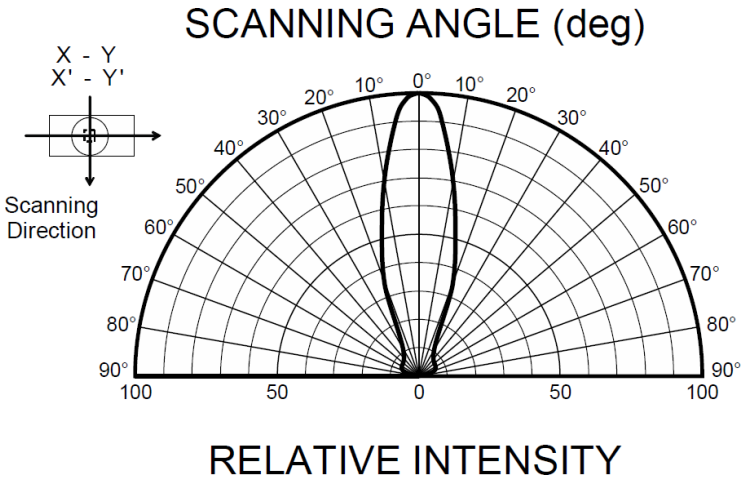


Figure 4.7: Scanning angle and directivity of infrared LED. [3, p. 3]

collider which should model the directivity and angle of the real LED. The red light ray, which is emitted from the LED model, should only show the middle point of the sphere collider.

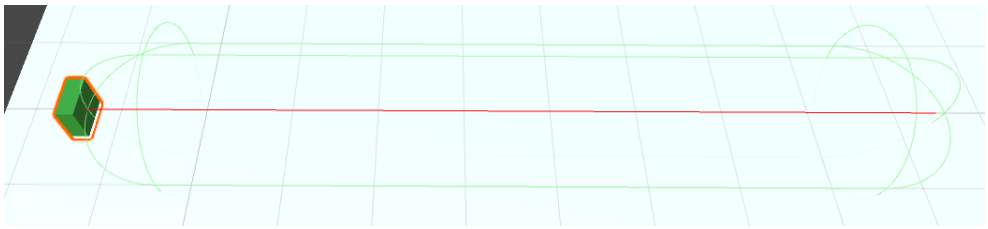


Figure 4.8: Infrared LED model in Unity.

4.2.4 Infrared Receiver Model

The infrared receiver [4], which is mounted on the agent, is used to detect the infrared light, emitted by the charging station. This receiver is modelled similar to the diagram presented in figure 4.9, which simply states out that the sensitivity of the receiver depends on the angle of the incoming infrared light. Based on the sensitivity diagram, it can be said, that

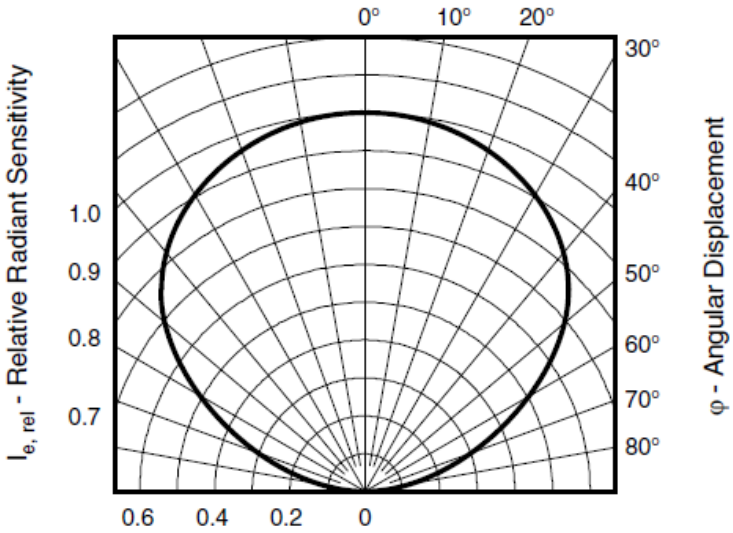


Figure 4.9: Sensitivity of the infrared receiver based on the angle of the incoming infrared light. [4, p. 3]

the sensory output value of the infrared receiver depends on the angle of the receiver to the LED, the distance between them and the displacement of the receiver normal to the emitted light. For better visualisation these three dependencies are shown in figure 4.10 The infrared receiver model in Unity uses these three dependencies to calculate the sensory output value. The output is zero if the angle between the LED to the receiver exceeds 70° degrees. If not, the sensory output value is calculated with the equation 4.1

$$value = cos(angle) - displacement * 0.75 - distance/10.0 \tag{4.1}$$

Figure 4.10: Visualization of the angle of the receiver to the LED, the distance between them and the displacement of the receiver normal to the emitted light.

4.2.5 Compass Model

To enable, that agent's perceived state is unique, a compass module [1] is used. Without this compass module, the agent would perceive the same state for different environmental positions. This would drastically hinder learning performance or even make it impossible. The compass module outputs the magnetic field force and acceleration in all three directions, observable in figure 4.11. The Unity model of the compass simply measures

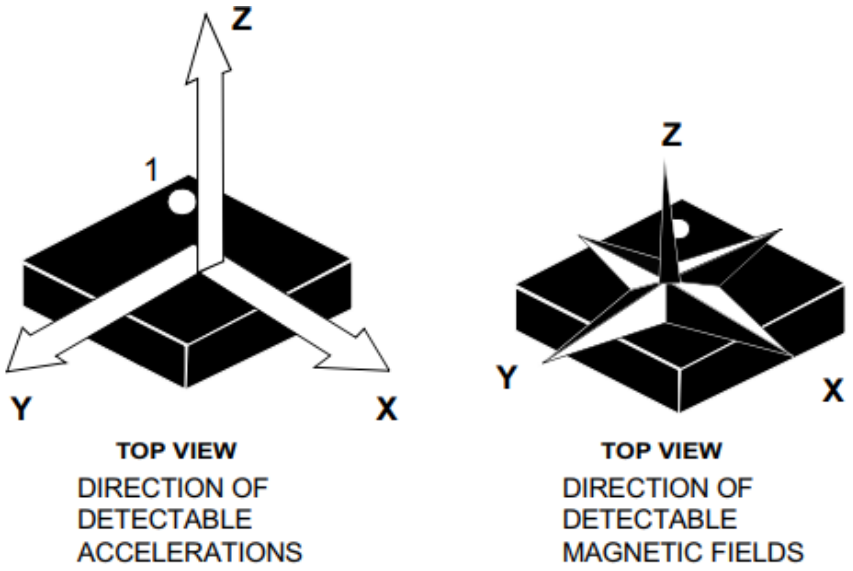


Figure 4.11: Acceleration and magnetic field measurement directions. [1, p. 11]

the accelerations in all three directions. Since Unity automatically measures the velocity of an object in all three directions, the values calculated by Unity are used. In addition, Unity provides the angles of an object relative to world coordinates. These angles, in combination with trigonometry, are used to estimate the magnetic field values.

4.2.6 Charging Station Model

For easy access, the design of the charging station, observable in figure 4.12, supports retraction. Charging pads, positive electrode drawn in red and negative electrode drawn in blue, are visible on the bottom of the charging station. On the back wall the infrared led can be observed. Charging is only possible, if the agent drives correctly onto the charging pads. In addition, the charging pads are offset to the front so that when the agent fully enters the charging station, no charging takes place.

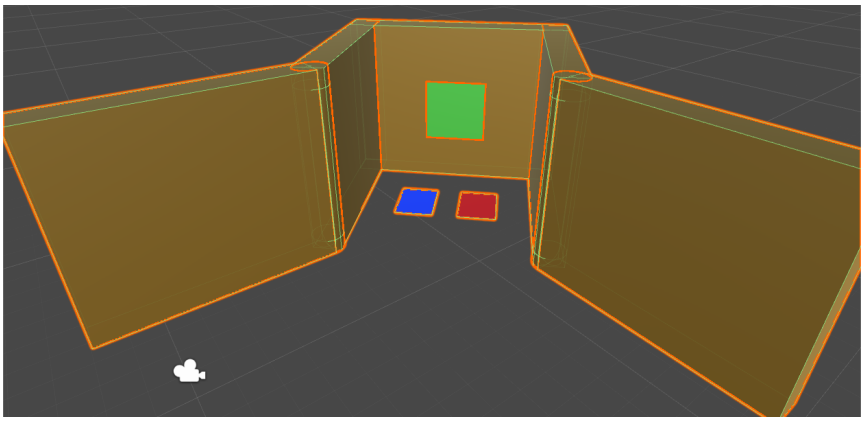


Figure 4.12: Model of the charging station in Unity.

4.2.7 Environment Model

To test the chosen $\mathbb{RL}$ method a quite small and simple environment model was used, observable in figure 4.13. It was chosen small and simple to reduce the training time which is required to find the charging stations. If a larger environment is necessary, the agent can be pre-trained with the smaller environment and subsequently with the larger one to reduce training time.

4.3 $\mathbb{RL}$ Method And Architecture

This chapter introduces the $\mathbb{RL}$ method used, including the architecture of the Q-learning method $\mathcal{M}$. In figure 4.14 the entire architecture, including the interaction of the agent with the $\mathbb{RL}$ method, can be observed. After the environment is reset, the agent perceives its initial state with the help of its sensors and passes it to the Q-learning method $\mathcal{M}$ (DDPG), which predicts the next action a to perform. The exploration method $\mathcal{E}$ adds noise to the predicted action a' to enable exploration of the environment. Then the agent performs the action and perceives the next state s' from the environment. This next state s' is then used to compute the reward r with the help of the reward function $\mathcal{RF}$. In addition, the previous state s of the environment is saved. Eventually, all

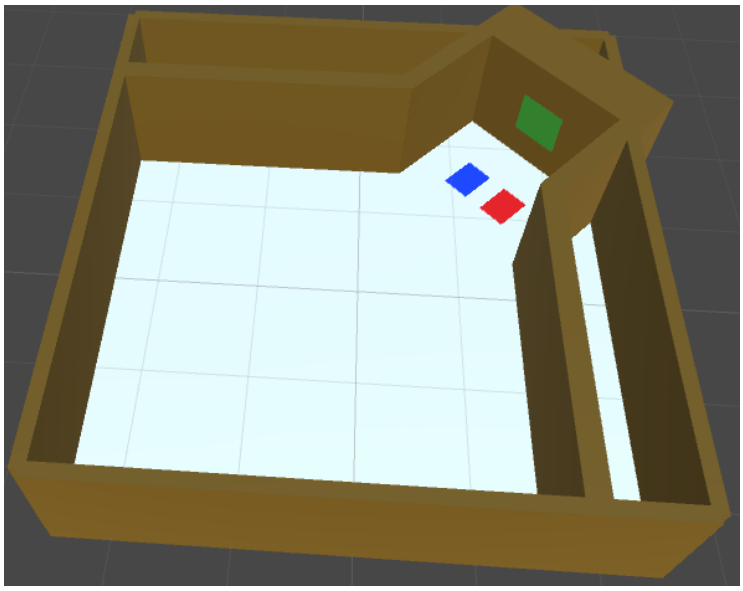


Figure 4.13: Model of the environment in Unity.

information has been gathered to store a transition (s, a', r, s') in the replay memory $\mathcal{RM}$. Finally, after enough transitions are stored in the replay memory $\mathcal{RM}$, the Q-network of the Q-method $\mathcal{M}$ can be trained with a minibatch of transitions. As

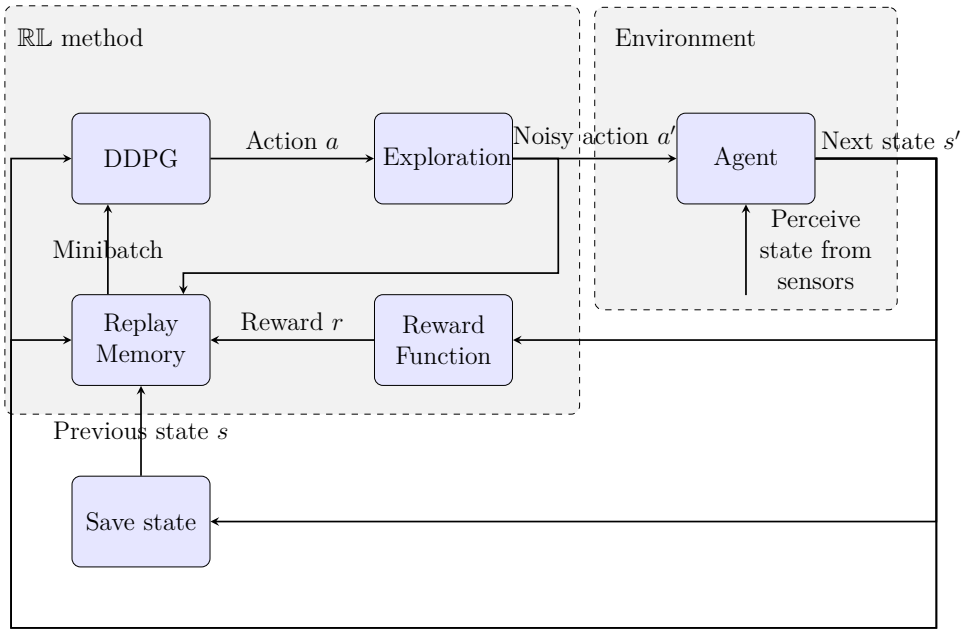


Figure 4.14: Interaction of ℝL method with agent and environment.

suggested by the selection of algorithms chapter 3, as ℝL method the following tuple is used: $(DDPG, OU, EXPR, Hybrid)$. $DDPG$ is applied because it performs best with continuous action and state spaces. As Q-network architecture the proposed architecture with two hidden layers from the DDPG article [LHP+15, p. 11] is used. In addition, the

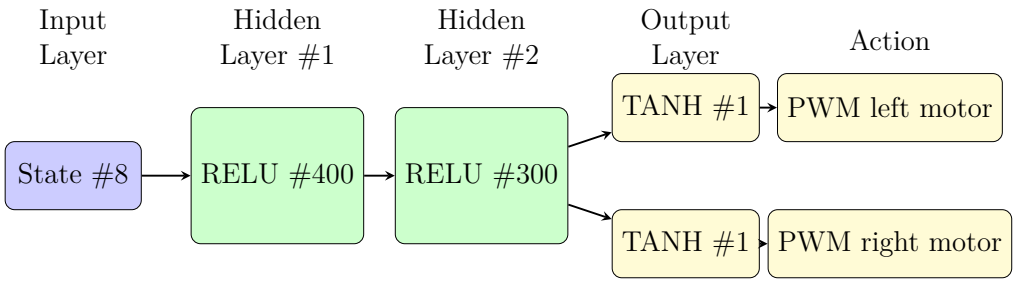


Figure 4.15: DDPG actor architecture.

same number of neurons for the first and for the second hidden layer (*400* and *300*), as mentioned in this article were applied. A *TANH* activation function is used in the output layer to support the output range of $[-1; 1]$. An output of *+1* means full speed of the actuators in clockwise direction, while an output of *-1* means full speed in counterclockwise direction. The states provided to the Q network consists of 8 values. Some of them are measured by the sensors of the agent. Only if agent's battery is currently charged is determined via a measuring circuit. The state consists of the measured value of the front range finder, the right range finder, the front infrared receiver, the velocity in x-direction, the velocity in y-direction, the angle in x-direction, the angle in y-direction and if the battery is currently charged. This state excludes the battery charge and the angle and velocity in z-direction. The battery charge state is omitted because it is not helpful for finding the charging station. The angle and velocity in z-direction are omitted because in the simulator the robot only operates on a plane. The Q-network architecture of the actor can be observed in figure 4.15 and the architecture of the critic in figure 4.16.

OU is selected as exploration method because the drift-like behaviour of the generated noise is a good choice for actuator control. If *EGC* is used instead, the agent would be moving back and forth all the time. This is not the desired exploration behaviour. Since *PPROP* and *HERGD* performed bad in continuous state space environments, *EXPR* is selected as replay memory method.

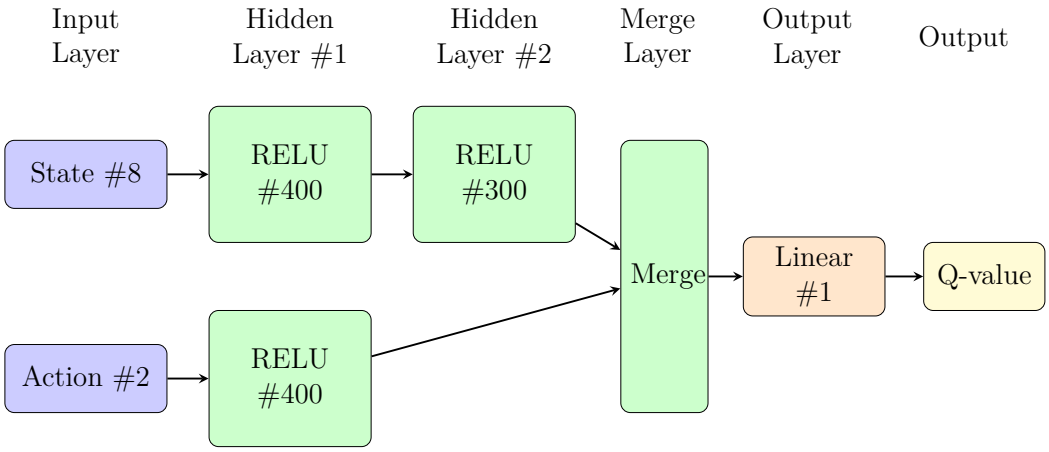


Figure 4.16: DDPG critic architecture.

Finally, a *hybrid* reward function is used. It is called *hybrid* because it is a combination of a *shaped* and a *non shaped* reward function. If agent's infrared receiver is in range of the infrared LED, a *shaped* reward function is used to guide the agent towards the charging station. Otherwise, the reward is set to *-10*. For simplicity, the outputted value of the infrared receiver is limited to the range $[0; +1]$. A value of zero means that the receiver is out of range of the transmitter, while a value of *+1* means that the agent's receiver is directly in front of the transmitter. If the agent is charging, which means the charging pads of the agent are in contact with the charging pads of the charging station, the reward is set to *100*. A reward of *100* for the recharging may seems very high at first glance, but considering that the agent needs many steps to reach the charging station and thus the reward has to be propagated backwards many steps, this value is chosen to be large enough. The reward function can be observed in equation 4.2.

$$\mathcal{RF} = \begin{cases} +100, & \text{if agent is charging.} \\ 10 * (-1 + \text{IR receiver output value}), & \text{otherwise.} \end{cases} \tag{4.2}$$

The *shaped* reward function does not generate any positive values because otherwise reaching the infrared LED light bulb could be considered as a subgoal. In this case it could not longer be guaranteed, that the agent's policy $\pi(s)$ will convert to the optimal policy $\pi_*(s)$. In addition, the time to learn the task will increase dramatically.

4.4 Evaluation

For this evaluation, the environment from section 4.2.7 is used. The agent is executed in this environment for *1500* episodes with *500* steps each. After every *50th* episode the learning progress is determined. For this purpose, the learned policy $\pi(s)$ is used *10* times for *500* steps each. For each of them, the total reward is summed up and finally the average reward is calculated. In addition, the standard deviation is determined. After each episode, the agent is reset to a random position and with a random rotation, but not directly into the charging station.

As reward function a hybrid function (equation 4.2) is used. The minimal reward per episode is exactly $\mathcal{RF}_{min} = steps * (-10) = -5000$. This can happen if the agent was not charged or was not within range of the infrared LED during an entire episode. The theoretical maximum reward achievable by the agent is $\mathcal{RF}_{max} = steps * 100 = 50000$. For that, the agent has to be reset in correct position on the charging pads of the charging station after an episode ends. Since this is not possible because the charging station is not a correct reset position, this reward can not be achieved.

The Q-learning method, as mentioned in section 4.3, is used. The learning rates of the actor and target network are selected to be small to enable slow adaptation. Since past experiences are worthy in this environment, the discount factor is chosen to be close to one. Since the reward function is *hybrid* a quite large batch size is used. In addition,

two new factors are used for this evaluation. First, the *L2 regularization* which addresses the issue of overfitting. The same value as proposed in article [LHP+15, p. 11] is used. Second, the *warm-up time*, which determines after how many transitions the training begins. Otherwise, training would begin immediately after a minibatch of transitions is available. From then on, a minibatch is sampled from the replay memory after each game step. The first transitions would be sampled many times. This can severely affect learning performance and is therefore avoided with the help this warm up time.

All the used $\mathbb{RL}$ method parameter settings can be observed in table 4.1.

Environment	Figure 4.13
Episodes	1500
Steps	500
Tests	10
Q-learning method $\mathcal{M}$	DDPG
Optimizer	Adam
Learning rate actor α_A	0.0001
Learning rate critic α_C	0.001
Target update factor τ	0.001
L2 regularization	0.001
Discount factor γ	0.99
Hidden layer 1 neurons	400
Hidden layer 2 neurons	300
Hidden layer activation function	*RELU*
Output layer activation function	*TANH*
Batch size	128
Warm-up time	20 Episodes
Replay memory $\mathcal{RM}$	EXPR
Size	1000000
Exploration method $\mathcal{E}$	OU
θ	0.3
μ	0.0
σ	0.3
Reward function $\mathcal{RF}$	Hybrid equation 4.2

Table 4.1: Parameter settings for simulation evaluation.

Result

The results of the agents can be observed in figure 4.17. Considering the simple agent, whose line is drawn in red, it can be observed that after ≈*400* episodes the average reward becomes positive. This means that it sometimes managed to find the charging station and move properly onto the charging pads. From then on, it can be seen, that the simple agent is constantly improving its task solving performance. Slowly it learns from every reset position and angle the optimal way to the charging station. After ≈*1000* episodes the improvement stops. From then on the simple agent has learned to find the charging station from almost every state of the environment.

In figure 4.18 a successful attempt of the simple agent finding the charging station

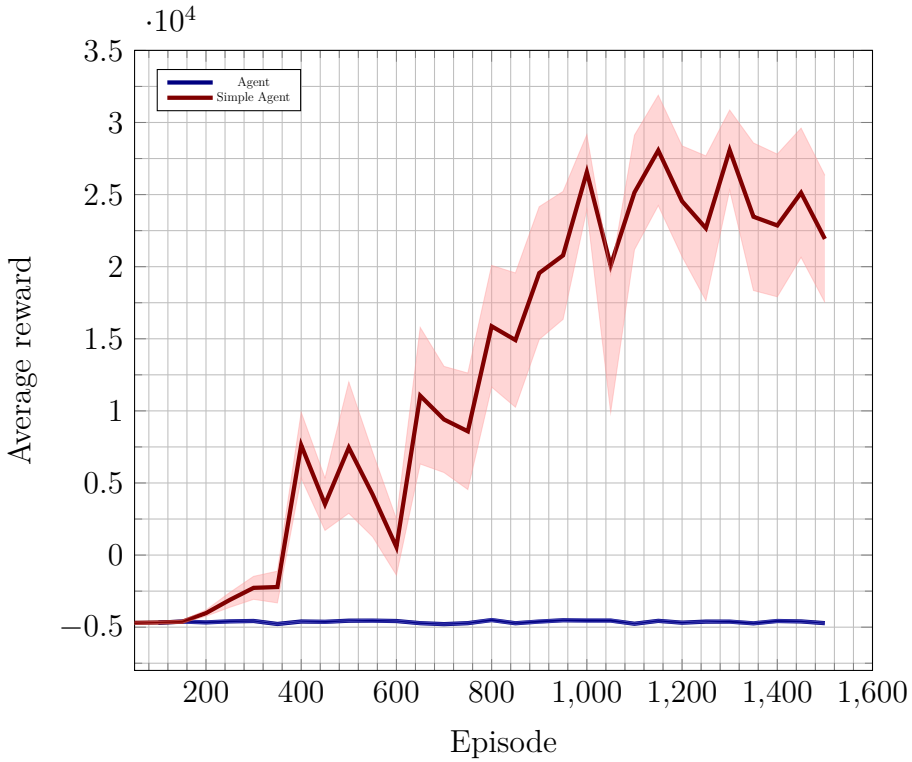


Figure 4.17: Simulation results of different agents.

can be observed. Although the simple agent starts with its back to the charging station, it managed to find the charging station and move properly onto the loading pads. However, looking at the real agent model, drawn in blue, it can be said that no learning success was made. The reason why the agent does not reliably manage to find the charging station is due to the interaction of several sub-problems. On the one hand, this is due to the two actuators and the associated complex movements. The robot moves forward or backward only when both actuators are turning in the same direction. Otherwise it turns in a circle. Since random exploration of large state spaces is a difficult task per se, but in combination with agent's complex movements, this gets much worse. Due to the drifting behaviour of *OU*, in most cases the agent only learns to spin around its own axis. The agent has

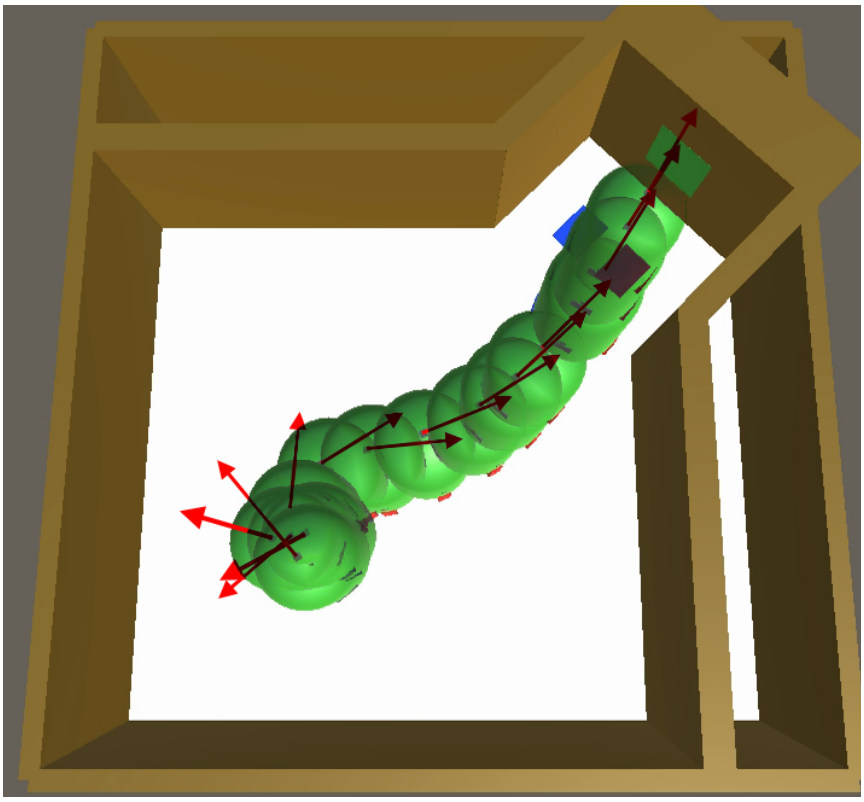


Figure 4.18: Trajectory of the simple agent moving towards the charging station.

learned that if his infrared receiver is aiming towards the charging station's infrared LED for some time, it will result in less negative rewards overall. Obviously, with this rotating behaviour, good exploration of the state space, which is crucial for *RL*, is therefore not achieved. Further, this causes the replay memory to be filled with many more transitions that do not contribute to the task's resolution. This contributes to *catastrophic forgetting*. Even if the agent has managed to reach the charging pads properly within an episode, it will not learn to repeat this behaviour after training ends, because the Q-network is trained with too many non-task-relevant transitions. One workaround for this would be to simplify agent's movements. For example, the movement of the simple agent can be imitated. A vector used for forward or backward movement and another for left or rightward movement can be applied. Apart from that, the movements can be discretized to forward, backward, left and right. However, this is not recommended because of the lower scalability and because the movements need to be simplified for each other robot construction.

On the other hand, the robot is equipped with only a few sensors resulting in a very poorly resolved state of the environment. First, this leads to the fact, that the shape of the robot is difficult to learn for the Q-network. As a consequence, the robot often jams in the charging station or wedges with a wall. This makes it difficult to properly reach the charging pads. Furthermore, the limited sensors make it difficult to define a reward function that motivates the robot to solve the task. Figure 4.19 shows the reward

in dependence of agent's current state. The charging station is located in the upper right corner and is marked with a thick red border. The dashed red diagonal line indicates the light emitted by the infrared LED. Figure 4.19a shows the reward for the case if agent's infrared receiver is not pointing towards the infrared LED of the charging station. Therefore, the reward for all states is *-10*. The next figure 4.19b shows the other case; Agents's infrared receiver is directly pointing towards the infrared LED of the charging station. This environment can be considered as very difficult low reward environment because the agent only gets a reward other than *-10* when pointing towards the infrared LED. In addition, the area of influence of the infrared LED is small, so the reward for most states in the environment is *-10*. With more sensors equipped, perhaps a more motivating reward function could be defined. It would be ideal if the agent receives a reward depending on the distance to the charging station, as can be observed in the figure 4.19c.

-10	-10	-10	-10	-10	-10	-10	-10	-10
-10	-10	-10	-10	-10	-10	-10	-10	-10
-10	-10	-10	-10	-10	-10	-10	-10	-10
-10	-10	-10	-10	-10	-10	-10	-10	-10
-10	-10	-10	-10	-10	-10	-10	-10	-10
-10	-10	-10	-10	-10	-10	-10	-10	-10
-10	-10	-10	-10	-10	-10	-10	-10	-10
-10	-10	-10	-10	-10	-10	-10	-10	-10
-10	-10	-10	-10	-10	-10	-10	-10	-10

(a) Reward if agent is pointing away from charging station.

-10	-10	-10	-10	-10	-10	-10	-10	100
-10	-10	-10	-10	-10	-10	-10	-1	-10
-10	-10	-10	-10	-10	-10	-2	-10	-10
-10	-10	-10	-10	-10	-3	-10	-10	-10
-10	-10	-10	-10	-4	-10	-10	-10	-10
-10	-10	-10	-5	-10	-10	-10	-10	-10
-10	-10	-6	-10	-10	-10	-10	-10	-10
-10	-7	-10	-10	-10	-10	-10	-10	-10
-8	-10	-10	-10	-10	-10	-10	-10	-10

(b) Reward if agent is pointing towards charging station.

-8	-7	-6	-5	-4	-3	-2	-1	100
-8	-7	-6	-5	-4	-3	-2	-1	-1
-8	-7	-6	-5	-4	-3	-2	-2	-2
-8	-7	-6	-5	-4	-3	-3	-3	-3
-8	-7	-6	-5	-4	-4	-4	-4	-4
-8	-7	-6	-5	-5	-5	-5	-5	-5
-8	-7	-6	-6	-6	-6	-6	-6	-6
-8	-7	-7	-7	-7	-7	-7	-7	-7
-8	-8	-8	-8	-8	-8	-8	-8	-8

(c) Optimal reward function.

Figure 4.19: Reward depending of agent's current state.

5 Conclusion

The main objective of this thesis is to address the question how a robot can learn to solve tasks completely independently. For scalability, the task should not be tied to any assumptions such as the environment or the design of the robot. Therefore, the robot should have as little prior knowledge about the task, the environment and the meaning of its in- or outputs. This should help to solve tasks, which can not be completely solved by algorithms because of their complexity and enormous state space. The big vision is, that any kind of robot can independently learn to solve an intended task, eliminating the time-consuming and error-prone writing of algorithms and programs. This will result in higher production yield and throughput. The task addressed in this thesis is to learn to survive as long as possible. Therefore, the robot should find the charging station and charge its battery.

Since this task setting involves robotics, the chosen algorithm should handle following issues: continuous state and action spaces because sensor inputs and actuator outputs are continuous; poorly resolved environmental state, since robots are often equipped with fewer sensors; sparse rewards due to the large state space and should require as little knowledge about the task and the environment.

The robot possesses several inputs from sensors for state observation and outputs to actuators for movement. To determine the boundaries of the environment, for orientation and for state observation, two rangefinders and a compass are mounted on it. More on, it is equipped with two photodiodes to facilitate locating the charging stations as they emit ultraviolet light. Additional hardware has been installed to determine the battery charge status. To enable movements, two actuators are placed at the bottom of the robot.

Since machine learning is usable for prediction tasks, methods from this field are applied. However, since one requirement is to learn to solve the task in the best possible way, the method used must be from the field of Reinforcement learning, especially Q-learning algorithms are applied. Q-learning calculates Q-values that show how much reward to expect by performing a particular action from a certain state. However, Q-learning requires other methods to work. A reward function is used to reward or

punish the robot's actions. To enable learning from past experiences and to guarantee the convergence of the Q-network a replay memory has to be used. Finally, an exploration method that allows the robot to explore the environment and gather information about the task to be learned is needed.

To familiarize ourselves with Q-learning and to be flexible about which Q-learning algorithms and required methods are used on the finished robot, several of these methods are implemented and evaluated against each other. In addition, new techniques and improvements done to the state of the art methods with the aim of achieving better results overall, such as *Hindsight Experience Replay with Goal Discovery*, ϵ*-greedy Continuous* and *Ornstein-Uhlenbeck Annealed*, are implemented. With the aid of these evaluations, it was possible to determine which methods should be used for the robot. As Q-learning method, *Deep Deterministic Policy Gradient* is used because it performs best with continuous action and state spaces. Standard *Experience Replay* is used as replay memory because it provides quite the same results compared to *Proportional Prioritized Replay Memory* for continuous state spaces, but requires less training time. As reward function a combination of a *shaped* and a *non shaped* reward function was used. In evaluations the *shaped* reward functions performed better than the *non shaped* ones, but due to the fewer sensors of the robot, only a combination of both was possible.

To ensure that the selected methods are executed on the final hardware without major changes, the robot, the environment, and the methods are simulated. This includes the modelling of all sensors, the charging station and the environment. These simulations showed that a simpler model of the robot could learn to solve the problem while the model of the real robot failed. First, the complex and slippery movements of the robot caused by the positioning of the actuators, is not easy to learn. In addition, this causes also the exploration method to struggle, since random exploration of large state spaces is a difficult task per se, but in combination with the complex movements, it is getting even worse. Also the rectangular construction of the robot leads to several issues, such as it often jams in the charging station or wedges with a wall. Furthermore, the limited sensors make it difficult to define a reward function that motivates the robot to learn to solve the task. This causes the goal state to be under explored. The Q-network is trained with too many non-task-relevant transitions.

This work has shown that robots using Q-learning are, under certain circumstances, able to learn to solve tasks independently. Still a lot of little things have to be considered in order to enable a successful learning of a task. Such as that the task to be learned has to be very simple and special attention must be paid to the sensors used in order to be able to resolve the environmental state well enough and to be able to define a motivating reward function. Furthermore, the construction of the robot movements should be defined with simplicity in mind. If small mistakes or sloppiness are made to the just mentioned points, this can impair the learning success or even make it impossible. To make the usage of Q-learning handier, it is necessary to work on better exploration methods, as the learning time and success depend drastically on them. Furthermore, improvements to the Q-Learning algorithms must be met to better deal with sparse rewards. If these issues are

solved, Q-learning has the potential to address many unsolved robotic and algorithmic problems and help to shape the future.

5.1 Future Work

As mentioned in section 4.1, the $\mathbb{RL}$ method is executed on an external host because of the limited memory of the agent. In the future, it is planned to compute more and more parts of the $\mathbb{RL}$ methods in the agent's processor. Most of the memory is consumed by the Q-network architecture and the replay memory. In order to reduce the memory usage of the Q-network, it is necessary to find the minimal architecture which is still capable of learning the task. By decreasing the number of neurons in each layer, fewer weights and biases are needed to be stored, which drastically reduce memory requirements, since each neuron is connected to all of the neurons in the previous layer. Then, the bit length of the weights and biases can be shortened to further reduce the memory requirements. In order to reduce the replay memory size, the minimum number of transitions to be stored in the replay memory, which is needed to solve the problem, can be searched. Or the replay memory size could be adjusted adaptively, for example with an algorithm such as mentioned in article [LZ17, p. 2]. This approach has the advantage of reducing development time and scaling better.

A big problem is that an entire training run takes a lot of time. For each action the agent performs, data must be exchanged between the Unity program and the Python script, transitions need to be stored in the replay memory, the next action needs to be computed by the Q-network and new weights and biases have to computed with the aid of a minibatch. These are all the reasons why a whole training run takes about 4 to 6 hours, depending strongly on the architecture of the Q-network, the replay memory and exploration method used. Obviously, this dramatically increases the development time. To achieve faster results faster, these processes should be parallelized. This means that several agents have to be simulated and the data is gathered by one Q-network which is trained with the experiences of all agents. Such approaches already exist, for example, A3C mentioned in article [MBM+16, p. 8].

With sparse reward tasks, one problem is that a transition with a positive reward had to be sampled from replay memory and had to be propagated back by repeatedly sampling the predecessor states. The Q-network slowly learns which actions to perform from certain states in order to reach the rewarding state. Is the rewarding state perceived only few times, this process is disturbed. Considering catastrophic forgetting, successful learning of the task becomes unlikely. In order to accelerate the back propagating of Q-values, an improvement could be made to the Q-learning algorithm. For example, the Q-function could be applied to the transitions before saving an episode to the replay memory. Therefore a positive reward is present throughout an entire episode. Sampling a transition with positive reward becomes more likely. A disadvantage of this change is that the algorithm would convert more slowly to the ideal policy. This approach can be

combined with the n-step loss mentioned in the article [VHS+17, p. 3], which should help to propagate the Q-values along the trajectories.

A great influence on the learning time and the learning success would be the improvement of the exploration methods. As mentioned in section 2.1.10, only random exploration methods are possible because the robot should know as little as possible about the task and the environment. Count-based exploration methods [THF+16, pp. 1-2] that store a counter, how many times a state has been visited, and what actions are taken to exit the state, are not applicable for this work because memory on an embedded system is a limited resource. An interesting idea is presented in article [PHD+17, pp. 1-9]. Instead of applying noise only to the output neurons, it is applied to the entire Q-network. Another approach is to let the decision maker adaptively learn the exploration policy in DDPG, presented in article [XLZP18, p. 1]. Advantage is that this approach is scalable and yields to a better global exploration. Disadvantage is that this approach consumes more memory than *OU* or *EGC*.

As mentioned in section 4.2.1, the robot is not built yet and therefore a lot of parameters had to be estimated. This includes the friction parameters of the actuators contact point to the ground, the transmittable torque and a more accurate modelling of the sensors used. Is the final robot available, these parameters can be redefined for a more realistic simulation environment.

When survival has been successfully resolved, other tasks can be tackled. For example, the Q-network that solves survival can be stored. Another Q-network is used to learn an new task. After the battery charge is critical, the agent can switch to the Q-network that has learned to survive. After the battery has been charged successfully, learning continues with the other Q-network. Even the heuristics, when to switch to other Q-networks, which are learning different tasks, can be learned by a Q-network.

5.2 Outlook

One reason why this thesis is important and contributes to current research is that it shows what to consider when using Q-learning for real robots. In addition, the strengths and weaknesses of Q-learning and the methods required are demonstrated. Firstly, it is shown which building blocks (exploration method, replay memory and reward function) are required for Q-learning. For each of these required building blocks, various state of the art methods are presented, evaluated and discussed. In addition, useful parameters are revealed for each of them. These explanations, supported by diagrams, make it easier for researchers, who are not yet familiar with this field, to properly select the methods and parameters for their own problem to be solved. Secondly, the adapted interaction model, presented in section 3.1, shows for the first time how and where the reward function has to be calculated for real robots. In contrast to environments that are only executed in software, the reward for real robots must be calculated by the robot itself based on the

sensor values. Thirdly, it is shown how important the selection of sensors is, in order to be able to define a motivating reward function. This selection must also be adjusted to the problem to be solved in order to enable a successful design of the reward function. Fourthly, this work has shown that the more complex the movement of the robot is, the less successful the learning with the current exploration methods is. This indirectly refers to one of the biggest weaknesses of Q-learning, the lack of good exploration methods.

Enabling robots to learn complex tasks through experience allows us to take a big step into the future. The applications for such self learning robots are limitless. Writing of complex algorithms to control these robots is completely eliminated because they learn to control themselves. In addition, through repetition they are able to constantly optimize their behaviour. Changes in the environment do not affect them because they can adapt to them automatically. It is conceivable that these robots share already acquired knowledge with each other. This can drastically reduce the time required to learn a completely new task. In medicine, they could perform complicated operations, for example on the brain, where very subtle movements are needed. They can even be used to enter human-unfriendly environments after an environmental crisis or even other planets.

Literature

[AWR+17] Andrychowicz, Marcin ; Wolski, Filip ; Ray, Alex ; Schneider, Jonas ; Fong, Rachel ; Welinder, Peter ; McGrew, Bob ; Tobin, Josh ; Abbeel, Pieter ; Zaremba, Wojciech: Hindsight Experience Replay. In: *CoRR* abs/1707.01495 (2017)

[Bar12] Barber, David: *Bayesian Reasoning and Machine Learning*. New York, NY, USA : Cambridge University Press, 2012. – ISBN 0521518148, 9780521518147

[Bel54] Bellman, Richard: The theory of dynamic programming. In: *Bull. Amer. Math. Soc.* 60 (1954), 11, Nr. 6, S. 503–515

[Bel57] Bellman, Richard: A Markovian Decision Process. In: *Indiana Univ. Math. J.* 6 (1957), S. 679–684. – ISSN 0022–2518

[Day92] Dayan, Peter: The Convergence of TD(λ) for General λ. In: *Machine Learning* 8 (1992), May, Nr. 3, S. 341–362. – ISSN 1573–0565

[FAS10] Fuchida, Takayasu ; Aung, Kathy T. ; Sakuragi, Atsushi: A study of Q-learning considering negative rewards. In: *Artificial Life and Robotics* 15 (2010), Sep, Nr. 3, S. 351–354. – ISSN 1614–7456

[Gas02] Gaskett, Chris: *Q-Learning for Robot Control*, The Australian National University, Diss., 2002

[HGG18] Hanai, Tuka A. ; Ghassemi, Mohammad M. ; Glass, James R.: Detecting Depression with Audio/Text Sequence Modeling of Interviews. In: *Interspeech*, 2018

[HGS15] van Hasselt, Hado ; Guez, Arthur ; Silver, David: Deep Reinforcement Learning with Double Q-learning. In: *CoRR* abs/1509.06461 (2015)

[HMLIR07] Heidrich-Meisner, Verena ; Lauer, Martin ; Igel, Christian ; Riedmiller, Martin A.: Reinforcement learning in a nutshell. In: *ESANN*, 2007

[HSSH17] Hwangbo, J. ; Sa, I. ; Siegwart, R. ; Hutter, M.: Control of a Quadrotor With Reinforcement Learning. In: *IEEE Robotics and Automation Letters* 2 (2017), Oct, Nr. 4, S. 2096–2103. – ISSN 2377–3766

[KB14] Kingma, Diederik P. ; Ba, Jimmy: Adam: A Method for Stochastic Optimization. In: *CoRR* abs/1412.6980 (2014)

[KP12] Kober, J. ; Peters, J.: *Reinforcement Learning in Robotics: A Survey*. Bd. 12. Berlin, Germany : Springer, 2012, S. 579–610

[KPR+17] Kirkpatrick, James ; Pascanu, Razvan ; Rabinowitz, Neil ; Veness, Joel ; Desjardins, Guillaume ; Rusu, Andrei A. ; Milan, Kieran ; Quan, John ; Ramalho, Tiago ; Grabska-Barwinska, Agnieszka ; Hassabis, Demis ; Clopath, Claudia ; Kumaran, Dharshan ; Hadsell, Raia: Overcoming catastrophic forgetting in neural networks. In: *Proceedings of the National Academy of Sciences* 114 (2017), Nr. 13, S. 3521–3526. – ISSN 0027–8424

[LHP+15] Lillicrap, Timothy P. ; Hunt, Jonathan J. ; Pritzel, Alexander ; Heess, Nicolas ; Erez, Tom ; Tassa, Yuval ; Silver, David ; Wierstra, Daan: Continuous control with deep reinforcement learning. In: *CoRR* abs/1509.02971 (2015)

[LZ17] Liu, Ruishan ; Zou, James: The Effects of Memory Replay in Reinforcement Learning. In: *CoRR* abs/1710.06574 (2017)

[MBM+16] Mnih, Volodymyr ; Badia, Adrià P. ; Mirza, Mehdi ; Graves, Alex ; Lillicrap, Timothy P. ; Harley, Tim ; Silver, David ; Kavukcuoglu, Koray: Asynchronous Methods for Deep Reinforcement Learning. In: *CoRR* abs/1602.01783 (2016)

[MKS+13] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Graves, Alex ; Antonoglou, Ioannis ; Wierstra, Daan ; Riedmiller, Martin A.: Playing Atari with Deep Reinforcement Learning. In: *CoRR* abs/1312.5602 (2013)

[MKS+15] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Rusu, Andrei A. ; Veness, Joel ; Bellemare, Marc G. ; Graves, Alex ; Riedmiller, Martin ; Fidjeland, Andreas K. ; Ostrovski, Georg ; Petersen, Stig ; Beattie, Charles ; Sadik, Amir ; Antonoglou, Ioannis ; King, Helen ; Kumaran, Dharshan ; Wierstra, Daan ; Legg, Shane ; Hassabis, Demis: Human-level control through deep reinforcement learning. In: *Nature* 518 (2015), Februar, Nr. 7540, S. 529–533. – ISSN 00280836

[MP43] Mcculloch, Warren ; Pitts, Walter: A Logical Calculus of Ideas Immanent in Nervous Activity. In: *Bulletin of Mathematical Biophysics* 5 (1943), S. 127–147

[PG17] Patterson, Josh ; Gibson, Adam: *Deep Learning: A Practitioner's Approach*. Beijing : O'Reilly, 2017. – ISBN 978–1–4919–1425–0

[PHD+17] Plappert, Matthias ; Houthooft, Rein ; Dhariwal, Prafulla ; Sidor, Szymon ; Chen, Richard Y. ; Chen, Xi ; Asfour, Tamim ; Abbeel, Pieter ; Andrychowicz, Marcin: Parameter Space Noise for Exploration. In: *CoRR* abs/1706.01905 (2017)

[RNS13] Roy, N. ; Newman, P. ; Srinivasa, S.: *Tendon-Driven Variable Impedance Control Using Reinforcement Learning*. MITP, 2013. – ISBN 9780262315722

[RZL17] Ramachandran, Prajit ; Zoph, Barret ; Le, Quoc V.: Searching for Activation Functions. In: *CoRR* abs/1710.05941 (2017)

[SB98] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction*. MIT Press, 1998

[SLH+18] Shi, H. ; Lin, Z. ; Hwang, K. ; Yang, S. ; Chen, J.: An Adaptive Strategy Selection Method With Reinforcement Learning for Robotic Soccer Games. In: *IEEE Access* 6 (2018), S. 8376–8386. – ISSN 2169–3536

[SLHV18] Serj, Mehdi F. ; Lavi, Bahram ; Hoff, Gabriela ; Valls, Domenec P.: A Deep Convolutional Neural Network for Lung Cancer Diagnostic. In: *CoRR* abs/1804.08170 (2018)

[SLM+15] Schulman, John ; Levine, Sergey ; Moritz, Philipp ; Jordan, Michael I. ; Abbeel, Pieter: Trust Region Policy Optimization. In: *CoRR* abs/1502.05477 (2015)

[SQAS15] Schaul, Tom ; Quan, John ; Antonoglou, Ioannis ; Silver, David: Prioritized Experience Replay. In: *CoRR* abs/1511.05952 (2015)

[SSS+17] Silver, David ; Schrittwieser, Julian ; Simonyan, Karen ; Antonoglou, Ioannis ; Huang, Aja ; Guez, Arthur ; Hubert, Thomas ; Baker, Lucas ; Lai, Matthew ; Bolton, Adrian ; Chen, Yutian ; Lillicrap, Timothy ; Hui, Fan ; Sifre, Laurent ; van den Driessche, George ; Graepel, Thore ; Hassabis, Demis: Mastering the game of Go without human knowledge. In: *Nature* 550 (2017), Oktober, S. 354–

[THF+16] Tang, Haoran ; Houthooft, Rein ; Foote, Davis ; Stooke, Adam ; Chen, Xi ; Duan, Yan ; Schulman, John ; Turck, Filip D. ; Abbeel, Pieter: #Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning. In: *CoRR* abs/1611.04717 (2016)

[Thr92] Thrun, Sebastian B.: Efficient Exploration In Reinforcement Learning. Pittsburgh, PA, USA : Carnegie Mellon University, 1992. – Forschungsbericht

[VHS+17] Vecerik, Matej ; Hester, Todd ; Scholz, Jonathan ; Wang, Fumin ; Pietquin, Olivier ; Piot, Bilal ; Heess, Nicolas ; Rothörl, Thomas ; Lampe, Thomas ; Riedmiller, Martin A.: Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards. In: *CoRR* abs/1707.08817 (2017)

[WD04] Watkins, Christopher J. C. H. ; Dayan, Peter: Technical Note: Q-Learning. In: *Machine Learning* 8 (2004), S. 279–292

[XLZP18] Xu, Tianbing ; Liu, Qiang ; Zhao, Liang ; Peng, Jian: Learning to Explore with Meta-Policy Gradient. In: *CoRR* abs/1803.05044 (2018)

Internet References

[1] *Compass*, accessed 2019-29-03. https://www.st.com/resource/en/datasheet/lsm303agr.pdf.

[2] *Facebook's DeepText*, accessed 2019-29-03. https://code.fb.com/ml-applications/introducing-deeptext-facebook-s-text-understanding-engine/.

[3] *Infrared LED*, accessed 2019-29-03. https://www.rohm.com/datasheet/SML-S13RT/sml-s13rt-e.

[4] *Infrared receiver*, accessed 2019-29-03. https://www.vishay.com/docs/84317/temd7100itx01.pdf.

[5] *Keras*, accessed 2019-29-03. https://keras.io/.

[6] *Open AI Gym*, accessed 2019-29-03. https://gym.openai.com/envs/.

[7] *Open AI Gym; Pendulum*, accessed 2019-29-03. https://gym.openai.com/envs/Pendulum-v0/.

[8] *Reinforcement Learning Course*, accessed 2019-29-03. https://courses.cs.washington.edu/courses/cse571/16au/slides/20-rl-silver.pdf.

[9] *Tensorflow*, accessed 2019-29-03. https://www.tensorflow.org/.

[10] Microsoft. *C#*, accessed 2019-29-03. https://docs.microsoft.com/en-us/dotnet/csharp/.

[11] Python Software Foundation. *Python*, accessed 2019-29-03. https://www.python.org/.

[12] Unity Technologies. *Unity*, accessed 2019-29-03. https://unity3d.com.

A Neural Networks

Neural networks are modelled in a similar way as the human brain. Many simple units (neurons) are connected with each other to form more complex functions. There is no centralized unit which controls the neurons and they are working completely in parallel. [PG17, p. 41]

Each neuron sums up the information coming from the previous neurons (inputs). They consist of an *activation function*, a *bias* and a *weight*. In general, the *weights* are used to store the information. Of course, somehow the *bias* stores information too, but since his task is to additively shift the *activation function*, only the *weights* will be considered as information memory. The *activation function* determines if an neuron is active or not. Therefore, the *bias* allows more freedom to adjust the activity of a neuron. Learning is done by changing these *weights* and *biases*. The structure of an neuron is illustrated in figure A.1. The behaviour of a neural network is determined by its architecture, which describes how the neurons are connected. In this thesis only *feed-forward multilayer neural networks* are used. This kind of network consist of one *input layer*, one *output layer* and multiple *hidden layers*. Each layer contains several neurons. The layers are fully connected, which means that one neuron of one layer has a connection to all the neurons in the next layer. Figure A.2 illustrates an example of such a feed-forward multilayer neural network.

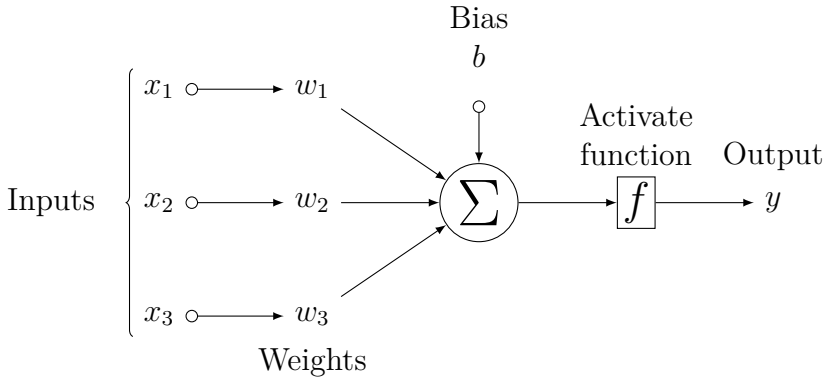


Figure A.1: Structure of an neuron.

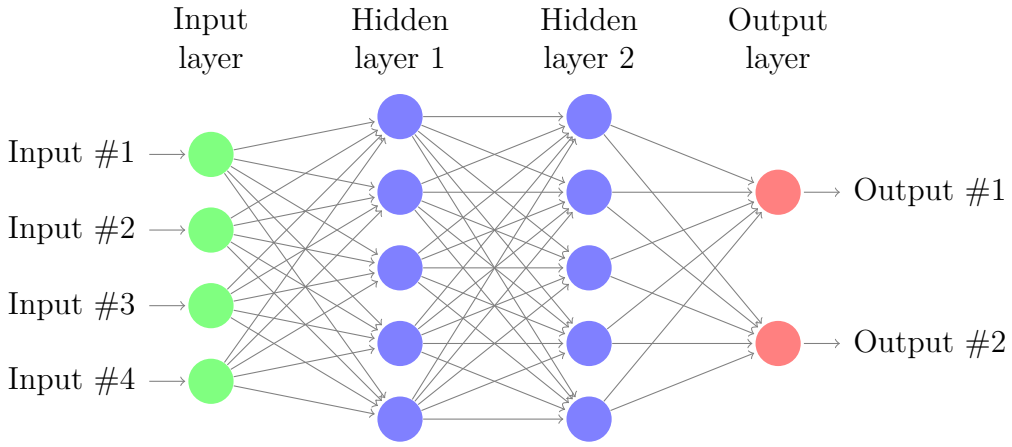
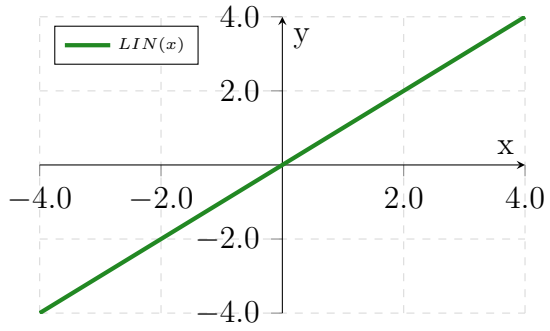


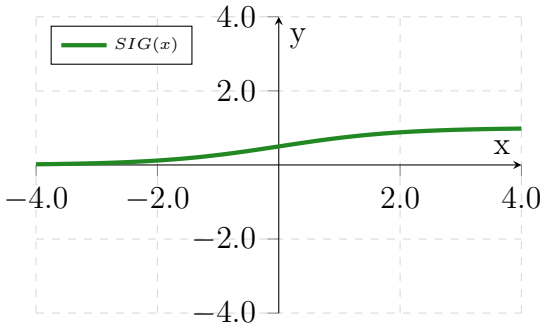
Figure A.2: Feed-forward multilayer network with four input nodes; two hidden layers each with five neurons and two output neurons.

A.0.1 Activation Functions

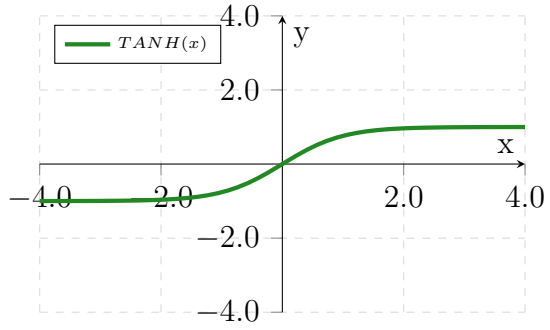
Activation functions in combination of weights, and biases defines the behaviour of a neuron. A neuron is considered to be active if it passes a non zero value (known as forward propagation) to another neuron. Activation functions affect many parameters such as the training dynamics, the tasks performance and the training time [RZL17, p. 1]. In addition, they are used to clip neuron outputs because open intervals may cause the neural network training algorithm to diverge. In the output layer, activation functions can be used for robotic control. For example, if an output is utilized for left or right steering of a car, the $TANH(x)$ A.3c can be used. Full steering to the left is encoded with -1.0 and 1.0 for a full steering to the right. An overview of the most common activation functions is shown in figure A.3.



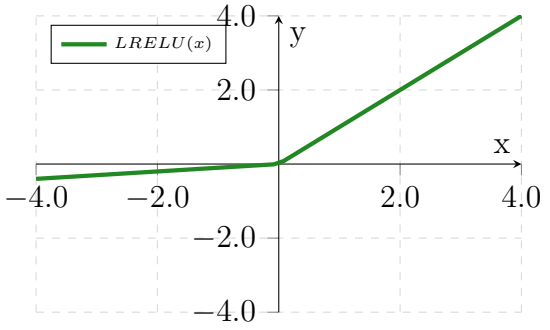
(a) Linear: $LIN(x) = x$



(b) Sigmoid: $SIG(x) = \frac{1}{1+exp(-x)}$

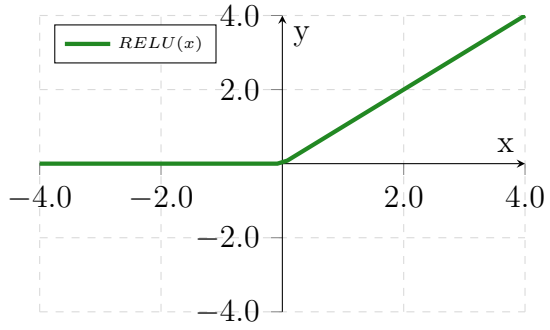


(c) Tanh: $TANH(x) = tanh(x)$



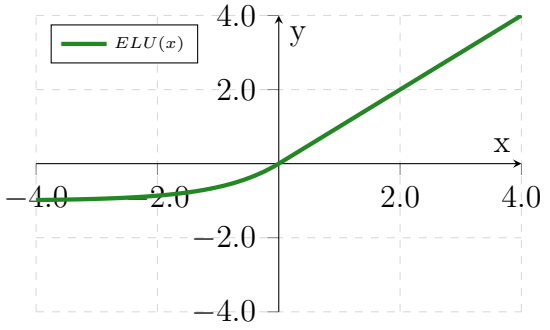
(d) Leaky rectifier linear unit:

$$LRELU(x) = \begin{cases} 0.1 * x, & \text{if } x < 0. \\ x, & \text{otherwise.} \end{cases}$$



(e) Rectifier liner unit:

$$RELU(x) = \begin{cases} 0, & \text{if } x < 0. \\ x, & \text{otherwise.} \end{cases}$$



(f) Exponential linear unit:

$$ELU(x) = \begin{cases} exp(x) - 1, & \text{if } x < 0. \\ x, & \text{otherwise.} \end{cases}$$

Figure A.3: Overview of popular activation functions.

Verfassungserklärung

David Bechtold
1150 Vienna
Austria

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct - Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Vienna, April 4, 2019

David Bechtold