# Deep adversarial reinforcement learning for object disentangling

Master-Thesis von Melvin Laux aus Darmstadt
Tag der Einreichung:

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: Dr. Joni Pajarinen
3. Gutachten: M.Sc. Oleg Arenz

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Deep adversarial reinforcement learning for object disentangling

Vorgelegte Master-Thesis von Melvin Laux aus Darmstadt

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: Dr. Joni Pajarinen
3. Gutachten: M.Sc. Oleg Arenz

Tag der Einreichung:

# Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 17. Dezember 2019

(Melvin Laux)

# Abstract

Deep learning in combination with improved training techniques and high computational power have led to recent advances in the field of reinforcement learning. Deep reinforcement learning methods have been used to teach agents how to play Go and Atari video games with human level performance. However, modern RL methods often fail to generalise to environments that differ from the one they were trained in or to transfer from simulation to real-world applications. These problems are caused by the fact that it is often impossible to train an agent in the entire environment, it is intended to be deployed in. For example, autonomous cars can not be trained for every possible scenario it might ever possibly encounter. Thus, generalisation from training to test scenarios often fails due to overfitting and a lack of exploration.
In this thesis, we present the *adversarial reinforcement learning* (ARL) framework, which utilises an adversarial agent, the *adversary*, which is trained to steer the original agent, the *protagonist*, to unknown and difficult states. The the protagonist and the adversary are trained jointly in order to allow them to adapt to the changing policy of their respective opponont. We show that our method is able to generalise by training an end-to-end system for robot control to solve the challenging object disentangling task for robotic arms. We perform an ablation study to investigate the effects of our method's hyperparameters, which shows that the our method is robust to changes to most hyperparameters. Our extensive experiments demonstrate that our method is indeed able to learn more robust policies that improve generalisation from training in simulation to both modified test scenarios as well as real world environments. As learning adversarial conditions may lead to a sparse distribution of positive rewards, we additionally propose a new form of prioritised experience replay for off-policy RL algotihms, *advantage-based experience replay* (ABER). This novel method of prioritising samples that are likely to increase the current policy's performance can increase learning speed and thereby reduce sample complexity.

# Zusammenfassung

Deep Learning in Kombination mit verbesserten Trainingsmethoden und extremer Rechenkraft führte in den letzten Jahren großen Fortschitten im Bereich des Reinforcement Learning. Deep Reinforcement Learning Algorithmen konnten dazu genutzt werden um Systeme zu trainieren die in der Lage sind das Brettspiel Go oder Atari Videospiele auf einem ähnlich hohen Level wie menschliche Experten zu spielen. Allerdings scheitern moderne Reinforcement Learning Methoden häufig in Umgebungen, die sich von denen Unterscheiden in denen sie trainiert wurden oder daran Aufgaben die in Simulationen gelernt wurden in der Realität durchzuführen. Diese Probleme entstehen durch den Fakt, dass es oft unmöglich ist ein RL System in der kompletten Umgebung zu trainieren in der es letztendlich agieren soll. Selbstfahrende Autos können nicht explizit auf jedes Szenario vorbereitet werden, das sie möglicherweise antreffen könnten. Aus diesem Grund scheitern RL Systeme häufig aufgrund von zu großer Fokussierung auf Trainingsszenarios und mangelnder Erkundung der Umgebung.
In dieser Masterarbeit stellen wir die Methode *Adversarial Reinforcement Learning* (ARL) vor, die einen Gegenspieler, den *Antagonist*, trainiert, um den ursprünglichen Agenten, den *Protagonist* in unbekannte oder schwierig zu lösende Situationen zu bringen. Der Protagonist und der Antagonist werden gemeinsam trainiert um es ihnen zu ermöglichen sich dem sich verbessernden Verhalten ihres jeweiligen Gegenspielers anzupassen. Wir zeigen in unseren Experimenten, dass unsere Methode in der Lage ist besser zu Generalisieren, indem wir ein System trainieren, das in der Lage ist, die herausfordernde Aufgabe lösen kann mit Hilfe eines Roboterarms Gegenstände zu entwirren. Wir untersuchen die Effekte und Eigenschaften der äußeren Parameter unserer Methode, und zeigen, dass unsere Methode robust bezüglich Änderungen dieser Parameter ist. Unsere ausgieben Experimente zeigen, dass ARL tatsächlich in der Lage ist robuste Strategien zu lernen die Generalisierung von Simulation zu Realität verbessern kann. Lernen in 'feindlichen' Umgebungen, d.h. in der Gegenwart eines Gegenspielers, kann dazu führen, dass es positive Bestärkungen nur selten zu erreichen sind. Aus diesem Grund präsentieren wir außerdem eine neue Methode um gesammlte Erfahrungen priorisert für den Lernprozess wieder zu verwenden: *Advantage-based experience replay* (ABER). Diese neuartige Methode priorisiert vergangene Erfahrungen anhand der Wahrscheinlichkeit, dass sie die aktuelle Verhaltensstrategie des Agenten verbessern kann. Weiter kann die Lerngeschwindigkeit erhöht werden, was zu einer reduzierten Menge an benötigten Daten führt.

# Acknowledgments

First, I would like to thank my thesis supervisors Dr. Joni Pajarinen and M.Sc. Oleg Arenz for their providing me with this interesting research topic, their time and insightful discussions. By allowing me the opportunity follow my own ideas and as well as always readily providing helpful feedback, they have made the time I spent on the preparation of this thesis highly educational and informative. I also want thank Prof. Dr. Jan Peters for providing me with the opportunity to work in the Intelligent Autonomous Systems research group at TU Darmstadt and sparking my interest in machine learning and robotics through his excellent lectures on statistical machine learning and robot learning. Furthermore, I want to thank the members of the IAS group for creating a very pleasant atmosphere and working environment during my time working on this thesis. Last, but not least, I want to thank my family and friends for their love and support which helped me to overcome the challenges during this year, both in relation to this thesis and all other matters.

# Contents

# Figures and tables

## List of Figures



## List of Tables

# Abbreviations, symbols and operators

## List of Abbreviations

| Notation | Description |
|---|---|
| ABER | Advantage-based experience replay |
| ANN | Artificial neural network |
| ARL | Adversarial reinforcement learning |
| CEM | Cross entropy method |
| CMA-ES | Covariance matrix adaption evolution strategy |
| CNN | Convolutional neural network |
| DDPG | Deep deterministic policy gradient |
| DQN | Deep Q-Network |
| FNN | Feedforward neural network |
| GAIL | Generative adversarial imitation learning |
| GAN | Generative adversarial network |
| i.i.d. | independently and identically distributed |
| KL | Kullback-Leibler divergence |
| MC | Monte-Carlo |
| MDP | Markov decision process |
| PER | Prioritised experience replay |
| RADE | Robot-arm disentangling environment |
| RARL | Robust adversarial reinforcement learning |
| REPS | Relative Entropy Policy Search |
| RL | Reinforcement learning |

| | |
|---|---|
| RNN | Recurrent neural network |
| SAC | Soft actor-critic |
| TD | Temporal difference |
| TRPO | Trust region policy optimisation |

## List of Operators

| Notation | Description | Operator |
|---|---|---|
| KL | the Kullback-Leibler divergence between two probability distributions | $\mathrm{KL}(\bullet \parallel \bullet)$ |
| ln | the natural logarithm | $\ln(\bullet)$ |
| $\pi$ | a behaviour policy | |
| T | the transpose of a matrix | $(\bullet)^T$ |

# 1 Introduction

## 1.1 Introduction

Over the last decades, machine learning and intelligent systems have had a great impact on various fields of applications due to significant breakthroughs. In 2016, DeepMind's AI system, AlphaGo, made a big impact by defeating the South Korean Lee Seedol, one of the world's leading Go players, when the board game Go was still considered a difficult problem for machine learning algorithms [1]. The robotics company Boston Dynamics have shown their humanoid robot Atlas, to have the ability to perform a variety of complex movement tasks such as running, jumping, parkour and gymnastics, showing the huge potential of robotics applications in the area of emergency rescue [2]. IBM's question-answering system Watson gained the public's attention in 2011, when it competed in the popular TV game show Jeopardy, defeating two of the most successful contestants of the show's history [3]. Even in everyday life, autonomous systems such as self-driving lawnmowers, driver-assistance systems or personal AI assistants like Apple's Siri and Amazon Echo have become a common sight. These and other examples show the ever-growing importance of autonomous systems in today's world as emerging fields such as autonomous driving, robotic control, computer vision and many more are poised to demonstrate the vast capabilities of machine learning in various fields of application.

Handcrafting the agent's behaviour for every possible situation is both infeasible and undesirable for many real-world applications like, e.g., robot control or autonomous driving. Using the approach of reinforcement learning, an AI system is trained to learn from its own experience through interactions with its environment. Thus, it is no longer necessary to manually program the agent's behaviour. As real-world applications present themselves as high dimensional problems, classical reinforcement learning algorithms are generally not sufficient to solve the task at hand unless they are combined with powerful function approximators such as deep neural networks. While the combination of deep learning and reinforcement has already resulted in successful applications such as TD-Gammon [4] in the early 90s, the idea has only regained more attention in recent years. The problems and difficulties in training deep networks caused by their high complexity computation and memory as well as the low sample efficiency resulted in a long drop in popularity of neural networks. However, a series of breakthroughs in deep learning such as the concepts of weight sharing and replay buffers in addition to the greatly improved computational power has led to the re-emergence of deep reinforcement learning (DRL).

Despite the recent advancements in the field of DRL, various problems remain during the training of DRL policies. In the field of robotic control, it is a common practice to learn an initial policy from expert demonstrations using imitation learning approaches, such as behavioural cloning, which in turn is used as a starting point for RL training to improve upon. This method allows to learn good policies, even from imperfect demonstrations. However, the availability of such demonstrations may be limited and very expensive. Furthermore, as most RL methods penalise large changes in each iteration of training, the final policy generally will not differ significantly from the one learnt initially by imitation learning and may perform considerably well in the training environment, yet fail to achieve to solve their task when applied in less controlled situations due to a lack of generality in the learned behaviour. In order to deal with this issue, it is possible to use so-called *transfer learning* methods to learn how to successfully apply a policy that learnt in a simulated environment to an different environment. This new environment may either be another simulation that uses a modified model, e.g. different friction parameters, or the real-world setting of the simulation the policy was trained on. [5, 6, 7]

Another issue in reinforcement learning for robot control is that it is difficult, if at all possible, to train the agent on the entire environment that it will be deployed in. If the final agent is to be used in a controlled environment such as manufacturing factories, creating an training scenarios for all possible situation to which the robot may be confronted with, is feasible. On the other hand, if the agent is intended to be deployed in an uncontrolled environment, e.g. a self-driving car in inner city traffic, the issue requires consideration. While pure RL training in real traffic is undesirable for obvious safety reasons, it is also difficult, if not impossible, to model every possible situation in either a laboratory or a computer simulation for training purposes. The behavior of an autonomous agent in an unfamiliar situation is, thus, hard to predict. To deal with this problem, a significant amount of research aims to create risk averse RL agents that go into a fail-safe state to minimise the possibility of catastrophic failure. However, finding a way to present as many critical situations as possible during training will improve overall performance in these situations. An approach to automatically generate as many of these situations, without the necessity of having to rely on manually designed examples from human

experts, is to use a second, adversarial policy. This adversary's task is to generate increasingly difficult situations for the original agent. This idea has shown great success in the field of supervised learning with generative adversarial nets (GANs) [8]. Since then, there have been various successful applications of the same concept in other fields of machine learning, such as imitation learning (GAIL) [9] or reinforcement learning (RARL) [5].
In this thesis, we present a novel adversarial learning framework of off-policy reinforcement learning algorithms. This framework *Adversarial reinforcement learning* (ARL) helps learning RL policies that are able to generalise to different situations by employing an adversarial policy to steer the agent to unknown or difficult situations. Through this adversary driven exploration of the state space, the adversary is more likely to learn a more general policy than by using purely random exploration. Additionally, we propose a new type of experience prioritisation which prioritises to replay samples that are likely to increase performance for off-policy RL methods.

## 1.2 Outline

This thesis will begin by summarising some required foundations in the fields of reinforcement learning, adversarial learning and deep learning in Chapter 2. The next chapter will present and explain the proposed adversarial reinforcement learning method. Subsequently, in Chapter 4, we will see the definition of a way to prioritise sample transitions based on their advantage during experience replay. In the next chapter, we will present our experiments. In the final chapter, we will draw a conclusion from the conducted experiments and discuss potential future works.

# 2 Foundations

In this chapter, we will first give an introduction into the fundamental concepts of reinforcement learning based on current literture. This introduction will include the general intuiton of reinforcement learning as well as its mathmatical foundations by formally modeling it as a Markov Decision Process. We will use definitions, content structure and figures based on various popular textbooks and articles [10, 11, 12, 13, 14, 15, 16]. In the next section, we will provide a brief summary of the basic concepts of deep learning, including a short overview of the history of neural networks, how they can be trained with the backpropagation algorithm and one the most commonly used type of network: feedforward neural networks. Subsequently, we will describe how the two concepts of reinforcement learning and deep learning can be conbined by providing an overview of popular state-of-the-art deep reinforcement learning algorithms such as DQN, DDPG, TRPO, SAC. Finally, we will investigate approaches of adversarial learning, i.e. concepts of how adversaries can be incorporated into the learning process in order to improve performance.

## 2.1 Reinforcement learning

Reinforcement learning (RL) is a machine learning method based on learning through experience. In this setting, an agent interacts with its environment while trying to improve its strategy to select actions based on the observations that are experienced during this process. The historical roots of RL stem from two distinct main threads which remained largely independent until they started to merge in the late 1980s and formed the field of modern RL. The first thread is based in animal behavioural psychology and is known as *trial-and-error learning* which has been investigated since the late nineteenth century, most notably by Edward Thorndike who coined the term "Law of effect" describing how postive and negative stimuli of an individuals experince influence its decision making process. The second thread is largeley based on Richard Bellmans research on the problem of *optimal control* from the 1950s. Bellman contributed the concept of *value functions* in order to solve optimal control problems through *dynamic programming* which are fundamental elements still used in modern RL.
The general RL setup consists of a learning *agent* and an *evironment*. At every given discrete time step $t$, the agent observes the current *state* of the environment and chooses which *action* to perform based on this observation. The agent then receives a *reward* signal depending on the performed action while the environment transitions into a new state. The agents goal is to maximise the expected future reward signal by modifiying its behaviour based on its previous experience. This basic interaction is illustrated in Figure 2.1.

### 2.1.1 Markov decision processes

The RL problem can be formalised using a Markov Decision Process (MDP), named after the Russian mathematician Andrey Markov, which consists of the following elements:

- a set of states $\mathscr{S}$ describes all possible states $\mathbf{s} \in \mathscr{S}$ of the environment
- a set of available actions $\mathscr{A}$ describes all possible actions $\mathbf{a} \in \mathscr{A}$ that are availible to the agent
- a state-transition probability $\mathscr{P}(\mathbf{s}_{t+1}|\mathbf{a}_t, \mathbf{s}_t)$ which models the probability of arriving in state $\mathbf{s}_{t+1}$ after taking action $\mathbf{a}_t$ in state $\mathbf{s}_t$
- a reward function $r(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$ which determines the received reward for taking action $\mathbf{a}_t$ in state $\mathbf{s}_t$ and arriving in state $\mathbf{s}_{t+1}$
- an initial state probability $\mu(\mathbf{s})$ determining the likelihood of a state to be the first in a state sequence of the MDP

In addition, we assume that the *Markov property* $\mathscr{P}(\mathbf{s}_{t+1}|\mathbf{a}_t, \mathbf{s}_t) = \mathscr{P}(\mathbf{s}_{t+1}|\mathbf{a}_t, \mathbf{s}_t, \mathbf{a}_{t-1}, \mathbf{s}_{t-1}, ..., \mathbf{a}_0, \mathbf{s}_0)$ holds for the transition model $\mathscr{P}$, i.e. that the next state solely depends on the previous state and action and is not influenced by any other past state or action within a sequence.
Two further parameters are commonly used to distiguish between types of MDPs and are occasionally included as a defining part of MDPs. First, the decision making process is highly dependent on how many steps are taken in the environment. If the agents time to interact with the environment is limited to a fixed number of time steps $H$, the MDP

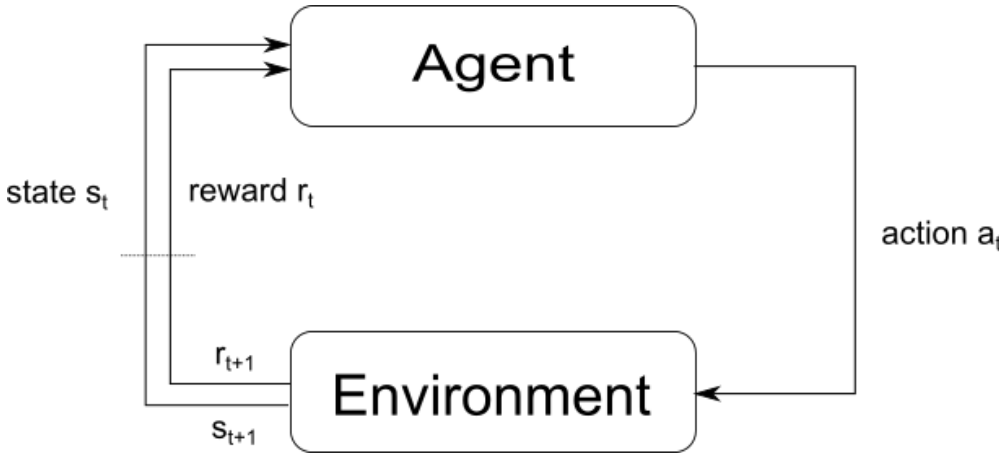



**Figure 2.1.:** At time step $t$, The agent observes the current state $\mathbf{s}_t$ of the environment. Based on this state, the agent's behavioural policy chooses an action $\mathbf{a}_t$, which changes the environment's state based on the system dynamics to the new state. At the next time step $t+1$, the agent observes the new state $\mathbf{s}_{t+1}$ as well as the reward signal $r_t$.

has a *finite horizon*. On the other hand, if no such number exists, i.e. $H = \infty$, the MDP is said to have an *infinite horizon*. MDPs with a finite horizon are also known as *episodic* as a full rollout of $H$ actions $a_t$ is commonly referred to as an $episode$. Infinite, in contrast, are also known as *non-episodic* or *continuing* MDPs. An agents behaviour is heavily impacted by the MDPs horizon since a restricted time frame might require the agent to take greater risks to achieve its goal than if an unlimited amount of time were available. For this thesis, we will generally assume episodic MDPs. The second parameter, which determines how much future rewards are considred in relation to the immediate reward during the decision making process, is known as the *discount factor* $\gamma \in [0, 1]$.

### 2.1.2 Policies and optimal decision making

A learning agent's behaviour can be mathematically formalised as a $policy$ function, denoted as $\pi$. A policy determines the agents next action based on the previous observed state of the environment. For finite state spaces, it is possible for the policy function to an explicit mapping of every possible state $\mathbf{s}$ to an action $\mathbf{a}$. This type of policy is commonly referred to as a *tabular* policy. However, tabular policies are only feasible options for low dimensional, discrete action and state spaces. Continuous control problems, such as controlling a robotoc arm, have an infinite amount possible states and/or actions, making it impossible to create and maintain such a mapping. In these cases it is necessary to use a *parameterised* policy function, whose behaviour is specified by the parameter vector $\boldsymbol{\theta}$. The parameters of the policy can represent various types of policies e.g. a linear combination of the input $\pi_\theta(\mathbf{s}) = \boldsymbol{\theta}^\top \mathbf{s}$ or the weights of a neural network. A policy can either be deterministic or stochastic. A deterministic policy is represented as a function $\pi$ which returns an action for a given input state $\mathbf{a} = \pi(\mathbf{s})$. Stochastic policies return a probabilty distribution over actions rather than a single action $\pi(\mathbf{a}|\mathbf{s}) = Pr(\mathbf{a}_t = \mathbf{a}|\mathbf{s}_t = \mathbf{s})$.

The reinforcement learning agent's goal while interacting with the environment is to find a policy that allows it to make optimal decisions in every state. Such an *optimal policy* $\pi^*(a|s)$ is formally defined as the policy that maximises the the RL objective function, i.e. the expected discounted reward, known as *return* $J(\pi)$, where $J$ is defined as

$$J(\pi) = \mathbb{E}_{\mu(\mathbf{s}_0),\mathscr{P}(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t),\pi(\mathbf{a}|\mathbf{s})}\left[\sum_{t=0}^{H} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})\right] \tag{2.1}$$

It should be noted that in the case of an infinite horizon ($H = \infty$), the choice of $\gamma = 1$ is invalid for Equation (2.1), since $\lim_{H->\inf} J(\pi) - > \infty$. In the case of parameterised stochastic policies $\pi_{\boldsymbol{\theta}}$ with parameter vector $\boldsymbol{\theta}$, finding the optimal policy boils down to the problem of finding an optimal vector $\boldsymbol{\theta}^*$ that maximises its return, that is,

$$\arg\max_{\boldsymbol{\theta}} J(\pi_\theta).$$

In order to simplify notation, it is possible to use the concept of *trajectories*, also known as *rollouts* or *paths*. A trajectory $\boldsymbol{\tau}$ defines a sequence of states, actions and rewards, namely,

$$\tau = (\mathbf{s}_0, \mathbf{a}_0, r_0, \mathbf{s}_1, \mathbf{a}_1, r_1, ..., \mathbf{s}_{H-1}, \mathbf{a}_{H-1}, r_{H-1}, \mathbf{s}_H)$$

Using the notation of trajectories, Equation (2.1) can be reformulated to

$$J(\pi) = \mathbb{E}_{P_\pi(\tau)}[R(\tau)],$$

where $P_\pi(\tau)$ is the probability of trajectories over policies, namely,

$$P_\pi(\tau) = \mu(\mathbf{s}_0) \prod_{t=0}^{H_A-1} \mathscr{P}(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)\ \pi_A(\mathbf{a_t}|\mathbf{s_t}),$$

and $R(\tau)$ is the cumulative reward of a trajectory, that is,

$$R(\tau) = \sum_{t=0}^{H} \gamma^t r_t.$$

In Section 2.2, we will introduce several methods to find optimal policies, i.e. that maximise Equation 2.1.

### 2.1.3 Challenges

Having formally defined the general reinforcement learing problem, we will now take a look at the main challenges of solving MDPs:

1. *The curse of dimensionality:* As Richard Bellman tried to apply optimal control methods to high dimensional problems, he realised that the amount of samples required to cover the state and action spaces grew exponentially. Due this "the curse of dimensionality", high dimensional environments require an enourmous amount of training samples [17].

2. *The exploration/exploitation tradeoff:* During the training of an RL agent, the agent is required to explore the structure of the environment's reward signal. However, it can not be certain whether its current best solution is the true optimal policy or if it needs to continue exploring the environment instead of exploiting the currently best known policy.

3. *The temporal credit assignment problem:* Determining which actions in a single episode were good is difficult in many environments, because the environments transition model remains unknown during learning. For this reason, the long-term effects of an action are not easy to determine. For example, an action may have a low immediate reward but lead to ultimately better overall result than taking the action with a high immediate reward. Hence, the agent needs to be able to learn long-term dependencies of its actions.

## 2.2 Reinforcement learning algorithms

In this section, after having defined the general reinforcement learning problem, we will take a look a the main types of algorithms that can be used to solve these problems. We will first examine *value function* based methods, which evaluate the agent's decisions by assigning numerical values to states or state-action pairs. Next, we take a look at the algorithm class of *policy search* methods. Policy search methods optimise the policy by searching the policies parameter space directly. Finally, we will introduce *actor-critic* methods which try to form a hybrid approach of both value-based and policy-based algorithms.

### 2.2.1 Value function methods

Value function methods originate from control theory and use mappings from states and actions to a numerical quality measure as a key component [10]. In order to be able to evaluate policies, we require a measure of 'how good' a policy will perform when being in a given state. This *state-value* of state $\mathbf{s}$ under policy $\pi$ is defined as the expected return that we receive when following $\pi$ starting in state $\mathbf{s}$. The *state-value function of policy* $\pi$, $V^\pi : \mathscr{S} -> \mathscr{R}$, can be formulated as

$$V^\pi(\mathbf{s}) = \mathbb{E}_{\mathscr{P}(\mathbf{s}_{t+1}|\mathbf{a}_t,\mathbf{s}_t),\pi(\mathbf{a}|\mathbf{s})}\left[\sum_{t=0}^{\infty}\gamma^t r(\mathbf{s}_t,\mathbf{a}_t)|\mathbf{s}_0=\mathbf{s}\right].$$

Similarly, the *action-value functionof policy* $\pi$ – most widely known as *Q-function* – $Q^\pi : \mathscr{S}\times\mathscr{A} -> \mathscr{R}$ is defined as the expected return when beginning from state $\mathbf{s}$, taking action $\mathbf{a}$ and subsequently following policy $\pi$, that is,

$$Q^\pi(\mathbf{s},\mathbf{a}) = \mathbb{E}_{\mathscr{P}(\mathbf{s}_{t+1}|\mathbf{a}_t,\mathbf{s}_t),\pi(\mathbf{a}|\mathbf{s})}\left[\sum_{t=0}^{\infty}\gamma^t r(\mathbf{s}_t,\mathbf{a}_t)|\mathbf{s}_0=\mathbf{s},\mathbf{a}_0=\mathbf{a}\right].$$

It is to obtain the state-value function from the action-value function and vice-versa, namely,

$$\begin{aligned} V^\pi(\mathbf{s}) &= \mathbb{E}_{\pi(\mathbf{a}|\mathbf{s})}\left[Q^\pi(\mathbf{s},\mathbf{a})\right], \\ Q^\pi(\mathbf{s},\mathbf{a}) &= r(\mathbf{s},\mathbf{a}) + \mathbb{E}_{\mathscr{P}(\mathbf{s}'|\mathbf{s},\mathbf{a})}\left[V^\pi(\mathbf{s}')\right]. \end{aligned} \tag{2.2}$$

Furthermore, using the theory of optimal control and dynamic programming, we can reformulate both value functions recursively, namely,

$$\begin{aligned} V^\pi(\mathbf{s}) &= \mathbb{E}_{\pi(\mathbf{a}|\mathbf{s})}\left[r(\mathbf{s},\mathbf{a}) + \gamma\mathbb{E}_{\mathscr{P}(\mathbf{s}'|\mathbf{s},\mathbf{a})}\left[V^\pi(\mathbf{s}')\right]\right], \\ Q^\pi(\mathbf{s},\mathbf{a}) &= r(\mathbf{s},\mathbf{a}) + \gamma\mathbb{E}_{\mathscr{P}(\mathbf{s}'|\mathbf{s},\mathbf{a}),\pi(\mathbf{a}'|\mathbf{s}')}\left[Q^\pi(\mathbf{s}',\mathbf{a}')\right]. \end{aligned} \tag{2.3}$$

Equation (2.3) is best known as the *Bellman equation* and build the foundation of many RL algorithms as they allow us to express the values of states as values of other states.
An additional useful measure is the *advantage function*, which measures how much better taking action $\mathbf{a}$ in state $\mathbf{s}$ instead of following policy $\pi$, that is,

$$A(\mathbf{s},\mathbf{a}) = Q(\mathbf{s},\mathbf{a}) - V(\mathbf{s}).$$

Using the concept of state-value functions, it is straight-forward to formulate the optimal policy, as we can choose the action that maximises the sum of immediate and expected future rewards, that is,

$$\pi^*(\mathbf{s}) = \arg\max_{\mathbf{a}}\ \left(r(\mathbf{s},\mathbf{a}) + \mathbb{E}_{\mathscr{P}(\mathbf{s}'|\mathbf{s},\mathbf{a})}\ \left[V^*(\mathbf{s}')\right]\right). \tag{2.4}$$

Defining the optimal policy is even easier if an optimal action-value function is provided. In any state $\mathbf{s}$, we can simply choose the action $\mathbf{a}$ that maximises its expected future reward, that is,

$$\pi^*(\mathbf{s}) = \arg\max_{\mathbf{a}}\ Q^*(\mathbf{s},\mathbf{a}). \tag{2.5}$$

Knowing that we can create an optimal policy as long as an optimal value function is available, the problem remains to find the optimal value functions $V^*$ and $Q^*$ of the optimal policy $\pi^*$. We will now look at two fundamental algorithms that solve this problem: *policy iteration* and *value iteration*.

#### Policy iteration

Policy iteration finds optimal policies by alternating between two different steps: *policy evaluation* and *policy improvement*. During policy evaluation, the current policy $\pi_k$ is evaluated by estimating its value functions $V^{\pi_k}$ and $Q^{\pi_k}$. Using Equations (2.2) and (2.3), we can obtain these estimates by alternating between calculating the state-value function from the Q-function and vice-versa until convergence. Next, in the policy improvement step, the policy $\pi$ is updated using $Q^\pi$. Similar to Equation 2.5, the updated policy $\pi'$ selects the action $\mathbf{a}$ that maximises $Q^\pi(\mathbf{s})$ in each state $\mathbf{s}$, that is,

$$\pi_{k+1}(\mathbf{s}) = \arg\max_{\mathbf{a}}\ Q^{\pi_k}(\mathbf{s},\mathbf{a}).$$

Next, policy evaluation is performed on the updated policy $\pi'$. This process is repeated until the policy improvement step no longer yields an update, i.e. until $\pi_k = \pi_{k+1}$.

### Value iteration

A major drawback of the policy iteration algorithm is that estimating $V^{\pi_k}$ and $Q^{\pi_k}$ is time-consuming and computationally expensive, especially in high dimensional environments. In each iteration's evaluation step, estimating the two value functions requires iterating over the entire state space multiple times until convergence is reached. The value iteration algorithm replaces the iterative computation of $V^{\pi_k}$ and $Q^{\pi_k}$ with the one step update which terminates after only a single sweep over all states. Using *Bellman's optimality equation*, it is possible combine the policy evaluation step and the policy update step into a single update rule, namely,

$$V^{\pi_{k+1}}(\mathbf{s}) = \max_{\mathbf{a}} \left( r(\mathbf{s}, \mathbf{a}) + \gamma \mathbb{E}_{\mathscr{P}(\mathbf{s}'|\mathbf{s},\mathbf{a})} \left[ V^{\pi_k}(\mathbf{s}') \right] \right).$$

This update is repeated until a convergence threshold is reached, resulting in an optimal state-value function $V^*(\mathbf{s})$, which can be used to a obtain an optimal policy using Equation (2.4).

### Monte-Carlo methods and temporal difference learning

Like optimal control and dynamic programming, policy iteration and value iteration assume knowledge of the system's transition model. However, in the traditional RL problem, the environment's dynamics are unknown. Thus, it is necessary to be able to learn directly from raw experiences without any explicit knowledge of $P(\mathbf{s}'|\mathbf{s}, \mathbf{a})$. RL algorithms can generally be categorised into two groups: *model-based* and *model-free* methods. While model-based approaches try to explicitly learn the system dynamics, model-free algorithms aim to directly learn value functions from the collected data samples. As the focus of this thesis lies on model-free approaches, we will not discuss model-based approaches in more detail.
One approach to model-free RL is to sample episodes from the environment, which in turn are then used to learn value functions and policies. These methods are known as *Monte-Carlo* (MC) methods. Monte-Carlo methods learn from full episode returns and update the estimated value function after each complete rollout using the *Monte-Carlo update*

$$V^{k+1}(\mathbf{s}_t) = V^k(\mathbf{s}_t) + \alpha(R_t - V^k(\mathbf{s}_t)),$$

where $R_t$ is the return after time step $t$ and $\alpha$ the learning rate parameter. As $R_t$ is only known at the end of each episode, it is only logical that value functions can only be performed after complete rollouts.
On the other hand, it is possible to learn directly from single-step rewards, an approach known as *temporal difference* (TD) learning. TD-methods combine the idea of learning from smapled rollouts with the concept of *bootsrapping* from dynamic programming. Bootstrapping describes the general idea of updating an estimate based on other estimates and is a technique, which is used by many reinforcement learning methods. After each time step $t$, a sample of the form $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ with which the so-called *TD-update* can be computed by

$$V^{k+1}(\mathbf{s}_t) = V^k + \alpha(r_t + \gamma V^k(\mathbf{s}_{t+1}) - V^k(\mathbf{s}_t)), \tag{2.6}$$

where the term in the brackets, $r_t + \gamma V^k(\mathbf{s}_{t+1}) - V^k(\mathbf{s}_t)$, is commonly known as the *TD error*, a quantity that appears in many RL algorithms. One of the earliest TD medthods consists of updating the state-value function after every time step $t$ using equation 2.6 and is known as the *TD(O)* algorithm [18]. The *state-action-reward-state-action* (SARSA) algorithm applies the concept updating the value function to the policy's Q-function instead of the state-value function by collecting trasitions of state-action pairs $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1}, \mathbf{a}_{t+1})$. Then, the TD-error of the Q-function can be computed as $\delta = r_t + \gamma Q^k(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - Q^k(\mathbf{s}_t, \mathbf{a}_t)$ [19]. Another major milestone was the famous *Q-learning* algorithm [20], which also uses TD-errors to update the policies Q-function. However, like TD(0), Q-learning works with collected tuples of the form $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ and compute the following TD-error to update the Q-function estimate: $\delta = r_t + \gamma \max_{\mathbf{a}'} Q^k(\mathbf{s}_{t+1}, \mathbf{a}') - Q^k(\mathbf{s}_t, \mathbf{a}_t)$. The only difference between Q-learning and SARSA is the way the action of the $t+1$ timestep is selected. In SARSA, the actually collected sample action $\mathbf{a_{t+1}}$ is used. Since this sample was produced by the same probability distribution, i.e. policy that is being optimised, SARSA is said to be an *on-policy*. Q-learning on the other hand selects the action that maximises the Q-function for the sampled state $\mathbf{s}_{t+1}$, thus updates the Q-function that was not generated by the target policy. Thus, Q-learning is called an *off-policy* algorithm. The question which proability distribution is used to produce training samples for an RL algorithm, i.e. whether an algorithm is on-policy or off-policy, is an important design choice as it comes with significant ramifications in terms of data-efficiency and exploration strategies.

### 2.2.2 Policy search methods

The second major category of RL algorithms are policy search methods. This class of algorithm does not use the concept of value functions, but instead optimise policies by directly searching its parameter space. Considering a parameterised policy $\pi_{\boldsymbol{\theta}}$ with parameter vector $\boldsymbol{\theta}$, the goal of policy search is to find a new vector of parameters $\boldsymbol{\theta}'$ that improves the policy's return, i.e. find a parameter vector, that yields higher return

$$J(\pi_{\boldsymbol{\theta}'}) > J(\pi_{\boldsymbol{\theta}}). \tag{2.7}$$

Once again, we will not discuss any model-based approached, and instead focus on model-free policy search.

Assuming a finite horizon $H < \infty$, it is possible to generalise policy search methods to consist of the following three steps [14]

- *Exploration*: Generate trajectories $\boldsymbol{\tau}^i$ of the form $(\mathbf{s}_0^i, \mathbf{a}_0^i, r_0^i, \mathbf{s}_1^i, \mathbf{a}_1^i, r_1^i, ..., \mathbf{s}_{H-1}^i, \mathbf{a}_{H-1}^i, r_{H-1}^i, \mathbf{s}_H^i)$ by using the current policy $\pi_{\boldsymbol{\theta}}$.
- *Policy evaluation*: Examine the quality of trajectories produced by the current policy.
- *Policy update*: Improve the current policy by computing an updated parameter vector $\theta'$ so that the inequality 2.7 holds.

Like in value-based approaches, policy search algorithms can be subcategorised into *episode-based* and *step-based* methods. While episode-based approaches evaluate complete trajectories in terms of their return $R(\tau^i) = \sum_{t=0}^{H} r_t$, step-based methods evaluate the quality of individual state-action pairs. In the following we will introduce one major type of methods for the policy improvement step: *policy gradient* approaches. While other classes of policy improvement techniques such as derivative-free methods like *simulated annealing* (SA) [21], evolutionary stratgies auch as the *covariance matrix adaptation evolution strategy* (CMA-ES) [22] or information theory-based techniques like the *cross-entropy method* (CEM) [23] exist, a complete survey of policy search methods is not in the scope of this thesis.

#### Policy gradients

Policy gradient methods are a type of policy search methods that update the policy parameter vector $\boldsymbol{\theta}$ based on the gradient of a given measure of performance with respect to the policy parameters $J(\pi_{\boldsymbol{\theta}})$. The gradient of this measure, $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$ is defined as

$$\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \int_{\tau} \nabla_{\boldsymbol{\theta}}\, p_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) R(\boldsymbol{\tau}) d\boldsymbol{\tau}, \tag{2.8}$$

where $p_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau})$ is the probabilty of policy $\pi_{\boldsymbol{\theta}}$ generating the trajectory $\boldsymbol{\tau}$. The policy gradient determines the direction of each policy update

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_k), \tag{2.9}$$

while the step size $\alpha$ controls the magnitude of the update. This gradient-based maximisation technique is famously known as *gradient ascent* [24]. In order to use policy gradient methods, the parameterised policy $\pi(\mathbf{a}|\mathbf{s}, \boldsymbol{\theta})$ must be differentiable with respect to $\boldsymbol{\theta}$. The gradient $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$ can be estimated with different approaches, e.g. *finite difference* or *likelihood ratio* methods [14]. In the following, we will derive a gradient estimate that is used by likelihood ratio methods including the *REINFORCE* [25] algorithm, one the first policy gradient methods. Likelihood ratio methods are based on the so-called 'likelihood ratio trick', which is expressed through the following identity for any probabilty density function $p$:

$$\nabla p(\mathbf{x}) = p(\mathbf{x}) \nabla \log p(\mathbf{x}) \tag{2.10}$$

By inserting the log likelihood trick (2.10) into equation (2.8), the policy gradient can be rewritten as

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int_{\tau} p_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) \nabla_{\boldsymbol{\theta}} \log p_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) R(\boldsymbol{\tau}) d\boldsymbol{\tau} = \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{\tau})}\left[\nabla_{\boldsymbol{\theta}} \log p_{\pi_{\boldsymbol{\theta}}}(\boldsymbol{\tau}) R(\boldsymbol{\tau})\right], \tag{2.11}$$

where the expectation $\mathbb{E}_{p_{\pi_\theta}(\tau)}\big[\nabla_\theta \log p_{\pi_\theta}(\tau) R(\tau)\big]$ can be approximated through Monte Carlo sampling:

$$\mathbb{E}_{p_{\pi_\theta}(\tau)}\big[\nabla \log p_{\pi_\theta}(\tau) R(\tau)\big] \approx \frac{1}{N}\sum_{i=0}^{N}\big[\nabla \log p_{\pi_\theta}(\tau^{\mathbf{i}}) R(\tau^{\mathbf{i}})\big], \tag{2.12}$$

where $N$ is the number of Monte Carlo rollouts. Furthermore, the probability of trajectory $\tau$ being generated by policy $\pi_\theta$ is defined in the following way:

$$p_\pi(\tau|\theta) = \mu(\mathbf{s_0}) \prod_{t=0}^{H} \mathscr{P}(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)\ \pi_\theta(\mathbf{a_t}|\mathbf{s_t}). \tag{2.13}$$

Using definition (2.13), the term $\nabla \log p_\pi(\tau|\theta)$ can be reformulated as

$$\nabla_\theta \log p_\pi(\tau|\theta) = \nabla_\theta \log\left(\mu(\mathbf{s_0}) \prod_{t=0}^{H} \mathscr{P}(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)\ \pi_\theta(\mathbf{a_t}|\mathbf{s_t})\right) = \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(\mathbf{a_t}|\mathbf{s_t}). \tag{2.14}$$

Finally, equations (2.12) and (2.14) can be inserted into equation (2.11) to get the following policy gradient estimate, that is also used in the REINFORCE algorithm:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(\mathbf{a}_\mathbf{t}^\mathbf{i}|\mathbf{s}_\mathbf{t}^\mathbf{i})\ R(\tau^{\mathbf{i}}).$$

Since the used trajectories are collected via Monte Carlo sampling, which is inherently noisy, the gradient estimate also suffers from large variance. However, a baseline $b$ can be subtracted from the sampled episode reward $R(\tau)$ to reduce the estimates variance:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=0}^{N}\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(\mathbf{a}_\mathbf{t}^\mathbf{i}|\mathbf{s}_\mathbf{t}^\mathbf{i})\ (R(\tau^{\mathbf{i}}) - b).$$

The introduction of a baseline term has generally no influence in the expectation of the gradient, however it can have a significant impact on its variance [10]. Furthermore, using equations (2.11) and (2.14), the policy gradient can be formulated as

$$\nabla_\theta J(\theta) = \mathbb{E}_{p_\theta(\tau)}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(\mathbf{a}_\mathbf{t}^\mathbf{i}|\mathbf{s}_\mathbf{t}^\mathbf{i}) R(\tau)\right].$$

It is possible to turn from this episode-based approach to a step-based approach by using rewards of state-action pairs instead of returns of trajectories:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta(\mathbf{a}|\mathbf{s})}[\nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s}) r_t].$$

Just like for episode-based policy gradients, introducing can reduce the estimate variance.

### 2.2.3 Actor-critic methods

While value function methods learn an explicit value functionwhich is used to create an implict policy by taking actions that maximise the learnt value function, poilcy search approaches learn an explict parameterised policy without using any value functions. Value function methods have the benefit of low variance in the estimated expected returns, but can not efficiently deal with continuous action spaces. On the other hand, policy search methods have the benefit of being able to generate continuous actions by using parameterised policies, but often suffer from slow learning due to a high variance of the estimated gradients [26]. *Actor-critic* methods combine these two concepts by explicitly learning both a policy and a value function. The value function is used as a baseline to the policy gradient estimate in order to reduce its variance. In the context of actor-critc methods, the policy $\pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s})$ is commonly referred to as the *actor*, while the value function $Q_{\boldsymbol{\phi}}(\mathbf{s})$ is called *critic*. Note that the critic is parameterised with parameter vector $\boldsymbol{\phi}$.
It is possible to decompose equation (2.2.2) in the following by,

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\tau)}\left[\sum_{t=0}^{H} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_t^i|\mathbf{s}_t^i) R(\tau)\right] = \mathbb{E}_{p_{\boldsymbol{\theta}}(\tau)}\left[\sum_{t=0}^{H} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_t^i|\mathbf{s}_t^i)\right] \mathbb{E}_{p_{\boldsymbol{\theta}}(\tau)}[R(\tau)]. \tag{2.15}$$

As defined previously, the second expectation in equation (2.15) can be rewritten as the Q-function (or state-value function). Thus, the policy gradient can be formulated using value functions:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\tau)}\left[\sum_{t=0}^{H} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_t^i|\mathbf{s}_\mathbf{t}^i) Q^{\pi_{\boldsymbol{\theta}}}(\mathbf{s}_t^i, \mathbf{a}_t^i)\right].$$

Finally, using the parameterised approximation of the Q-function $Q_{\boldsymbol{\phi}}(\mathbf{s})$ results in the following policy gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{p_{\boldsymbol{\theta}}(\tau)}\left[\sum_{t=0}^{H} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_t^i|\mathbf{s}_\mathbf{t}^i)\, Q^{\boldsymbol{\phi}}(\mathbf{s}_t^i, \mathbf{a}_t^i)\right].$$

In addition to updating the policy, the critic must now also be updated in each iteration by calculating the TD error $\delta$:

$$\delta = r_t + \gamma \max_{\mathbf{a}'} Q^{\boldsymbol{\phi}}(\mathbf{s}_{t+1}, \mathbf{a}') - Q^{\boldsymbol{\phi}}(\mathbf{s}_t, \mathbf{a}_t).$$

Using $\delta$, the Q-functions parameters can be updated with the critic's update function,

$$\boldsymbol{\phi}_{k+1} = \boldsymbol{\phi}_k + \alpha\delta \nabla_{\boldsymbol{\phi}} Q^{\boldsymbol{\phi}}(\mathbf{s}_t, \mathbf{a}_\mathbf{t}). \tag{2.16}$$

Thus, the actor and the critic can be trained simultaeneously by using equations (2.9) and (2.16) to update their respective parameter vectors[10].

## 2.3 Deep learning

After having provided an introduction to reinforcement learing, we will now follow up by providing a brief introduction to the second main ingredient to deep reinforcement learning: *deep learning*. The fundamental concept of deep learning are *artificial neural networks* (ANN), which are based on the idea of finding a mathematical model of biological learning. In 1943, Walter Pitts and Warren McCulloch formulated such a model by representing the human brain as a neural network [27]. A first successful implementation of such a model was the *Perceptron* algorithm, which could train a model of a single neuron and was published by the psychologist Frank Rosenblatt in 1958 [28]. As the perceptron started to gather increased research interest in ANNs, Marvin Minsky and Seymour Papert published a paper in 1969 that proved that it was impossible for the perceptron to learn a function as simple as the *exclusive-or* (XOR) [29]. With the publication of this paper, research interest in neural networks plummeted, beginning a period that is now often referred to as the 'first AI winter'. However, with the emergence of *multilayer perceptrons* and the *backpropagation* algorithm to efficiently train them in the late 1980s [30]. The popularisation of the *convolutional neural network* (CNN) has lead to successes in the field of image classification, e.g. recognition of handwritten digits on the MNIST dataset [31, 32]. During the 1990s, interest slowed down again due to difficulties in training deep network architechtures. Since then, due to the increase of available data and computational power, as well as improved training techniques like unsupervised pretraining of single layers [33, 34], weight sharing [35], improved activation functions (e.g. reLU [36]) and more efficient gradient-descent methods (RPROP [37], RMSprop [38], ADAM [39]) research interest has been ever growing and lead to significant breakthroughs in several fields of machine learning such as computer vision, speech recognition

and natural language processing. The term *deep learning* (DL) generally refers to ANN methods that contain at least one hidden layer and was popularised in the mid-2000s. Deep learning methods have set records on various benchmarks in the area of speech recognition [40, 41, 42], which was one of the first major industrial applications of deep learning [43]. Similarly, in the field of computer vision, DL approaches achieve the best results on the *ImageNet* [44] benchmark task of object recognition [45, 46] and set a record on the *ICDAR Chinese handwriting recognition* benchmark, achieving almost human performance [47]. Furthermore, DL algorithms have become the state-of-the-art in machine translation [48] and question-answering systems [49]. In March 2019, Yoshua Bengio, Geoffrey Hinton and Yann LeCun were awarded with the 2018 ACM Turing Award "for conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing".

### 2.3.1 Feedforward neural networks

One of the most common types of deep neural networks are *feedforward neural networks* (FNNs), also known as *multilayer perceptrons* (MLPs). The goal of a a neural network to approximate a target function $f^*$, e.g. a classifier $y = f^*(\mathbf{x})$ that maps an input vector $\mathbf{x}$ a class $y$. FNNs approximate $f^*$ using a parameterised function $y = f_{\boldsymbol{\theta}}(\mathbf{x})$ by learning optimal parameters $\boldsymbol{\theta}$ yield the best possible approximation [15]. FNNs model the target function using multiple interconnected layers of artificial *neurons*, also called *units*, which are representations of computational units that compute a single output value $y$ from a vector of multiple inputs $\mathbf{x}$. A neuron's output $y$ is computed a weighted sum of all input values using weights $\mathbf{w}$ and the bias $b$. Subsequently, the sum is transformed with an *activation function* $a$, which is typically nonlinear. The output of a single neuron is formulated as

$$z = h\left(\sum_{i=0}^{N} w_i x_i + b\right).$$

Neuron connections are structured as *layers*, such that each layer consist of multiple neurons. Data is processed by passing the output of one layer on to the next one as its input. In FNNs, the outputs of each neuron in a layer $l$ are used as an input for each neuron in the next layer $l+1$, i.e. all neurons in adjacent layers are connected. For this reason, FNNs are also referred to as *fully-connected* neural networks. Using Equation 2.3.1 and matrix formulation of sums, and assuming that all neurons in a layer use the same activation function, the output vector of a layer $l$ with $K_l$ units can be represented as

$$\mathbf{z} = h^k({\mathbf{W}^k}^T\mathbf{x} + \mathbf{b}^k), \tag{2.17}$$

where $\mathbf{W}^k$ is the matrix containing the weights of all neurons in layer $l$, i.e. $W_{ij}$ is the i-th weight of the j-th neuron. Vector $\mathbf{b}$ represents the biases of all neurons in layer $l$, i.e. $\mathbf{b} = (b_1, b_2, ...b_l)^T$. The output vector contains $\mathbf{z}$ the outputs of all neurons, i.e. $\mathbf{z} = (z_1, z_2, ..., z_l)^T$. Using the notation of Equation 2.17, complete neural networks can be represented as a single equation. For example, a three-layer FNN can be written as

$$\mathbf{y} = f^3({\mathbf{W}^3}^T(f^2({\mathbf{W}^2}^T(f^1({\mathbf{W}^1}^T\mathbf{x} + \mathbf{b}^1)) + \mathbf{b}^2)) + \mathbf{b}^3).$$

The first layer of a neural network is commonly referred to as the *input layer*, while the last layer is called the *output layer*. All other layers are generally known as *hidden layers*. Note that the parameters $\boldsymbol{\theta}$ of a neural network are its weights $\mathbf{W}^l$ and biases $\mathbf{b}^l$. Furthermore, FNNs have been shown to be universal function approximators, i.e. can approximate any smooth function to any desired accuracy given a sufficient number of neurons and layers.
Other than FNNs, the most common types of neural networks are *convolutional neural networks* (CNNs), which are heavily used in computer vision and image processing, and *recurrent neural networks* (RNNs), which can be used for processing sequential data, such as written texts, speech or videos. CNNs can be viewed as a specialisation of FNNs since they also process data in only one direction, but are more structured in the way neurons are connected. RNNs on the other hand are a generalisation of FNNs as they allow cyclic connections between neurons. However, a more in-depth introduction of CNNs and RNNs are out of the scope of this thesis.

### 2.3.2 Backpropagation

In order to update the parameters $\boldsymbol{\theta}$ of a neural network $\mathbf{y} = f_{\boldsymbol{\theta}}(\mathbf{x})$, gradient-based updates can be performed with respect to a given loss function $L$, e.g. the squared error, that is,

$$L(\boldsymbol{\theta}) = \sum_{i=0}^{N} L_n(\boldsymbol{\theta}) = \sum_{i=0}^{N} \frac{1}{2} [f_{\boldsymbol{\theta}}(\mathbf{x}_i) - \mathbf{t}_i]^2 .$$

The labeled training examples $(\mathbf{t}_i, \mathbf{t}_i)$ are provided by a set of training data points $D = \{(\mathbf{x}_0, \mathbf{t}_0), (\mathbf{x}_1, \mathbf{t}_1), ..., (\mathbf{x}_N, \mathbf{t}_N)\}$. The gradient of $L$, $\nabla_{\boldsymbol{\theta}} L(\theta)$ can subsequently be used to update the neural networks weights by,

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta}_{old} + \alpha \nabla_{\boldsymbol{\theta}} L(\theta),$$

where $\alpha$ is the step size of the gradient descent algorithm. The *backpropagation* algorithm is a method to perform gradient descent on neural networks in a computationally efficient way. While the term 'backpropagation' and its use for neural networks has been popularised in 1986 by Rumelhart et al. [30], the algorithm has been discovered and rediscovered several times since the 1960s [16, 50, 51].

By interpreting the bias as an additional weight for a constant input, the output of the j-th of the l-th layer can be computed as,

$$z_j^k = h^l \left( \sum_{i=0}^{N} \mathbf{W}_{ij}^l \mathbf{z}^{l-1} \right) = h^l(a_j^l), \tag{2.18}$$

where $a_j$ is the input into the activation function of the j-th neuron of the k-th layer. Applying the chain rule, the partial derivative of $L$ with respect to a single weight $W_{ij}^l$ can be calculated as ,

$$\frac{\partial L}{\partial W_{ij}^l} = \sum_{i=0}^{N} \frac{\partial L}{\partial W_{ij}^l} = \sum_{i=0}^{N} \frac{\partial L_n}{\partial a_j} \frac{\partial a_j^l}{\partial W_{ij}^l}.$$

The derivative of the loss with respect to the activation is called the *error*, denoted with $\delta$:

$$\delta_j^l = \frac{\partial L_n}{\partial a_j^l}. \tag{2.19}$$

Using (2.18) and (2.19), the partial derivative of the loss with respect to a single weight is calculated as,

$$\frac{\partial L}{\partial W_{ji}^k} = \delta_j^l \mathbf{z}^{l-1}.$$

As $\mathbf{z}^{l-1}$ is obtained by evaluation of the netowrk, only the error terms $\delta_j^l$ for each layer need to be calculated explicitly in order to obtain the gradient. For the output layer $L$, the error is computed as,

$$\delta_j^L = h'^k(a_j^L)(\mathbf{y} - \mathbf{t}).$$

By application of the chain rule, the errors for all other layers $l$ are computed by,

$$\delta_j^l = h'^l(a_j^L) \sum_{k=0}^{K_{l+1}} W_{jk}^l \delta_j^{l+1},$$

where $K_{l+1}$ is the number of neurons in layer $l+1$. Thus, the errors $\delta_j^l$ can be calculated layer wise, beginning from the output layer. Following this, the errors are propagated back through the network to calculate the errors in previous layers. This approach of backpropagation of errors is the fundamental, name giving feature of this method. Finally, each weight can be updated using the following update:

$$W_{ij,(old)}^l = W_{ij,(old)}^l + \alpha \delta_j^l \mathbf{z}^{l-1}.$$

It should be noted that the backpropagation algorithm can be derived in a similar fashion using other loss functions.

## 2.4 Deep reinforcement learning

While the idea of using neural networks as function approximators in reinforcement learning is not new, the recent advancements of deep learning have led to a renewed increase of interest in the field of deep reinforcement learning. Using powerful function approximators such as neural networks, can help to alleviate the lack of scalability in high dimensional problems for RL methods. This advantage is traded off with an increased amount of required data, as deep neural networks generally require a large amount of training examples. In this section, we will introduce several noteworthy examples of DRL methods in order to provide a brief overview DRL approaches. We will hereby follow a similar structure as in Section 2.2.

### 2.4.1 Value function methods

As introduced in Section 2.2, value based RL methods learn to approximate a value function in order to learn an optimal policy. In 2013, Mnih et al. introduced a method to approximate the Q-function using a neural network that consisted of two convolutional layers followed by two fully connected layers. Using this approach, Mnih et al. were able to train an agent to play a set of ATARI 2600 games directly from raw visual data, achieving human performance [52, 53]. The method has since been popularised under the name *Deep Q-Network* (DQN). DQN uses the mean squared TD-error as a loss function for gradient descent. Additionally, the method of *weight freezing* is applied in order to stabilise learning by reducing oscillations in the TD error. The idea of weight freezing is to use a second Q-network $Q_{\boldsymbol{\theta}'}$ that is frozen for several timesteps. The target network is then updated by copying the weights of the actual Q-network $Q_{\boldsymbol{\theta}}$. During training, the target network network is used to calculate the TD-error's target value as

$$L(\boldsymbol{\theta}) = (r + \gamma \max_{\mathbf{a}'} Q_{\boldsymbol{\theta}'}(\mathbf{s}_{t+1}, \mathbf{a}_t) - Q_{\boldsymbol{\theta}}(\mathbf{s}_t, \mathbf{a}_t)).$$

Additionally, a replay buffer was used for training to break temporal correlations between samples. Replay buffers are a simple method that stores past experiences in memory from which training examples are sampled. We will introduce this technique in more detail in the next chapter. It should be noted that due to the max operator over the action space in the loss function, DQN can only be applied to environments with discrete action spaces.

### 2.4.2 Policy search methods

In 2015, Schulman et al. proposed a policy search method called *Trust region policy optimisation* (TRPO) [54]. TRPO uses the following policy update:

$$\boldsymbol{\theta}_{new} = \arg\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \boldsymbol{\theta}_{old}) = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{(\mathbf{s},\mathbf{a})\sim\pi_{\boldsymbol{\theta}_{old}}} \left[ \frac{\pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s})}{\pi_{\boldsymbol{\theta}_{old}}(\mathbf{a}|\mathbf{s})} A^{\pi_{\boldsymbol{\theta}_{old}}}(\mathbf{s}, \mathbf{a}) \right],$$

where $L(\boldsymbol{\theta}, \boldsymbol{\theta}_{old})$ is the *surrogate advantage*, which measures how well $\pi_{\boldsymbol{\theta}}$ performs relative to the old policy $\pi_{\boldsymbol{\theta}_{old}}$ using data sampled from the old policy and $A(\mathbf{s}, \mathbf{a})$ the advantage function as defined in 2.2.1. Additionally, the change in policy parameters is limited using a constraint on the *Kullback-Leibler* (KL) divergence, which is an information theoretic measure of how much a probability distribution $p(x)$ differs from another probability distribution $q(x)$, namely,

$$KL(p(\mathbf{x})||q(\mathbf{x})) = \int p(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} dt.$$

Note that the KL divergence is not a formal measure of distance since the symmetry requirement is violated as generally $KL(p(\mathbf{x})||q(\mathbf{x})) \neq KL(p(\mathbf{x})||q(\mathbf{x}))$ except for $p(\mathbf{x}) = q(\mathbf{x})$. Using the KL divergence as an upper bound for the change in policy is an approach that has been used successfully before the introduction of TRPO already, e.g. in *Relative entropy policy search* (REPS) [55] and its extensions [56, 57, 58]. Hence, to update the policy parameters, TRPO solves the following optimisation problem:

$$\arg\max_{\boldsymbol{\theta}} \; L(\boldsymbol{\theta}, \boldsymbol{\theta}_{old}) \quad s.t. \quad \mathbb{E}_{\mathbf{s}\sim\pi_{\boldsymbol{\theta}_{old}}} \left[ KL(\pi_{\boldsymbol{\theta}}(\bullet|\mathbf{s})||\pi_{\boldsymbol{\theta}_{old}}(\bullet|\mathbf{s})) \right] \leq \delta. \tag{2.20}$$

In practice, Equation (2.20) is optimised using linear and quadratic approximations on the objective and the constraind respectively. Furthermore, TRPO performs a line search on the found gradient direction in order to find the largest step that improves the objective while satisfying the KL constraint.

Finally, we will introduce two closely related deep actor-critic methods: *Deep deterministic policy gradients* (DDPG) [59] and *Soft actor-critic* (SAC) [60, 61]. DDPG is a model-free actor-critic algorithm proposed by Lillicrap et al. [59]. The method maintains a deterministic target policy $\pi_{\boldsymbol{\theta}}(\mathbf{s})$ and a Q-function $Q_{\boldsymbol{\phi}}(\mathbf{s},\mathbf{a})$, which are both represented as a neural network. Training episodes are sampled using the deterministic policy in addition with a noise function $N(0,1)$ to allow for exploration, i.e. $a \sim \pi_{\boldsymbol{\theta}}(\mathbf{s}) + N(0,1)$. This method is known as the *reparameterisation trick*. The collected samples $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ are stored in a replay buffer $D$ from which mini-batches are sampled uniformly to calculate the policy and Q-function update. The deterministic policy is updated using the loss function $L(\boldsymbol{\theta}) = -\mathbb{E}_{s\sim D}\left[Q_{\boldsymbol{\phi}}(\mathbf{s}, \pi_{\boldsymbol{\theta}}(\mathbf{s}))\right]$ with the deterministic policy gradient [62]. In order to avoid the max operator over the continuous action space, the Q-function's gradient is computed using a target network of the policy network $\pi'_{\boldsymbol{\theta}}$ that approximately maximises the target Q-network $Q_{\boldsymbol{\phi}'}$. Thus, the Q-network is optimised by minimising the *mean squared bellman error* (MSBE), namely,

$$L(\boldsymbol{\phi}, D) = \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t,r_t,\mathbf{s}_{t+1})\sim D}\left[\left(Q_{\boldsymbol{\phi}}(\mathbf{s}_t,\mathbf{a}_t) - (r_t + \gamma V_{\boldsymbol{\phi}'}(\mathbf{s}_{t+1}, \pi_{\boldsymbol{\theta}'}(\mathbf{s_{t+1}})))\right)^2\right].$$

Furthermore, while DQN updates its target network by copying the actual weights, DDPG performs an update using *Polyak averaging* [63], that is,

$$\boldsymbol{\theta}' = p\boldsymbol{\theta}' + (1-p)\boldsymbol{\theta},$$

where $p$ is a hyperparameter between 0 and 1, usually close to 1.

Haarnoja et al. [60] introduced an extension to DDPG which features an entropy regularisation term in order to increase exploration in later stages of training and to reduce the risk of converging to a local maximum. The *entropy* $H$, is a quantity that measures the randomness of a random variable $\mathbf{x}$ drawn from a probability density distribution $P$. The entropy a variable is calculated by its distribution, namely,

$$H(\mathbf{x}) = \mathbb{E}_{\mathbf{x}\sim P}\left[-\log P(\mathbf{x})\right].$$

In entropy regularised RL, the agent receives a reward after each step. This bonus reward is proportional to the entropy of the policy at that timestep. Adding this term to the standard RL objective results in the following objective function for entropy regularised RL:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t,r_t,\mathbf{s}_{t+1})\sim\pi}\left[\sum_{t=0}^{\inf} \gamma^t \left(r_t + \alpha H(\pi(\bullet|\mathbf{s}_t))\right)\right],$$

where $\alpha > 0$ is the entropy cofficient, which controls the trade-off between entropy and expected returns. In this modified environment, the value functions change to include the entropy bonuses from all timesteps, that is,

$$\begin{aligned} V^{\pi}(\mathbf{s}) &= \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t,r_t,\mathbf{s}_{t+1})\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \left(r_t + \alpha H(\pi(\bullet|\mathbf{s}_t))\right)|\mathbf{s}_0 = \mathbf{s}\right], \\ Q^{\pi}(\mathbf{s},\mathbf{a}) &= \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t,r_t,\mathbf{s}_{t+1})\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t + \alpha \sum_{t=1}^{\infty} \gamma^t H(\pi(\bullet|\mathbf{s}_t))|\mathbf{s}_0 = \mathbf{s}, \mathbf{a}_0 = \mathbf{a}\right]. \end{aligned} \tag{2.21}$$

Using the updated value functions from (2.21), SAC concurrently learns a policy $\pi_{\boldsymbol{\theta}}$ and two Q-functions $Q_{\boldsymbol{\phi}_1}, Q_{\boldsymbol{\phi}_2}$. The Q-functions are updated by minimising the mean squared bellman error using a target value function, that is,

$$L(\boldsymbol{\phi}, D) = \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t,r_t,\mathbf{s}_{t+1})\sim D}\left[\left(Q_{\boldsymbol{\phi}}(\mathbf{s}_t,\mathbf{a}_t) - (r_t + \gamma V_{\boldsymbol{\phi}'}(\mathbf{s_{t+1}}))\right)^2\right],$$

where the state-value function $V$ is implicitly parameterised using the Q-functions parameters by the connection between $Q$ and $V$, that is,

$$V(\mathbf{s}_t) = \mathbb{E}_{\mathbf{s}_t\sim\pi}\left[Q(\mathbf{s}_t,\mathbf{a}_t) - \alpha log\, \pi(\mathbf{a}_t|\mathbf{s}_t)\right].$$

The policy updated using the reparameterisation trick, using $\mathbf{a}_t = f_{\boldsymbol{\theta}}(\boldsymbol{\epsilon}_t; \mathbf{s}_t)$, where $\boldsymbol{\epsilon}_t$ is an input noise vector sampled from a fixed distribution $N$. With this reparameterisation, the policy is optimised with respect to the following objective function:

$$L(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{s_t}\sim D, \epsilon_t\sim N}\left[\alpha log\, \pi_{\boldsymbol{\theta}}(f(\boldsymbol{\epsilon}; \mathbf{s_t})|\mathbf{s}_t) - Q_{\phi}(\mathbf{s}_t, f(\boldsymbol{\epsilon}; \mathbf{s_t}))\right].$$

Furthermore, Haarnoja et al. [61] have introduced a method to automatically the hyperparameter $\alpha$, i.e. the entropy trade-off coefficient.

## 2.5 Adversarial learning

In the final section of this chapter, we will provide a brief introduction to *adversarial learning*. We use the term "adversarial learning" to categorise learning methods that use adversarial training examples or include adversarial agents in order improve an agent's learning process. The main agent, whose policy is meant to be optimised is often referred to as the *protagonist*, while the policy that is used to disrupt the protagonist, is called the *adversary*. We will adopt this terminology for the remainder of this thesis. The general idea of adversarial learning is to learn policies that are robust to potentially malicious inputs. Consider the example of spam filters, whose task is to discriminate between real emails and fake emails. These fake emails are generally sent with malicious intent and intetionally designed to resemble real emails a much as possible in order to be misclassified by the spam filter and the recipient. The creation of such malicious inputs is generally called an *adversarial attack*. In the following, we will introduce two notable machine learning methods that use adversarial learning.

In 2014, Goodfellow et al. proposed an adversarial learning framework to estimate generative models, known as *Generative adversarial nets* (GANs) [8]. In this case, the protagonist is a generative method that aims to create a probability distribution of random variables that matches a given target distribution. For example, the target distribution can be provided in form of a set of images. Samples drawn from the learnt distribution can then be reshaped to form an image. Using this approach, a function is learned to generate images that resemble the ones from the training set. The generator is trained by sampling multiple examples from both the generator and the training data in every iteration. These batches of examples are then used to update the generator's parameters with respect to the difference of the distributions of the two batches, e.g. using the *maximum mean discrepancy* (MMD). The idea of GANs is to train a discriminator, whose task is to determine for a given sample $\mathbf{x}$ whether it was drawn from the true training distribution or the generator. Hence, the goal of the discriminator $D_{\boldsymbol{\theta}}(\mathbf{x})$ is to minimise the classification error for input samples $\mathbf{x}$. On the other hand, the generator $G_{\boldsymbol{\phi}}(\mathbf{z})$ is trained with the goal to produce samples that are likely to be misclassified by the disciminator, where $\mathbf{z}$ is the the random noise input vector to the generator. The generator produces samples by learning a parameteried tranformation from random noise to match the target distribution. Both agents are trained jointly, thus, in every iteration the disciminator's parameters are updated to minimise the classification error, while the generator's parameters are updated to maximise the classification error. Through this competetion between the two agents, the agents are driven to improve until the discriminator is no longer to able improve as the generator has exactly matched the target distribution. In practice, the discriminator is trained to output the probability that the provided sample was drawn from the training data distribution rather than produced by the generator. Formally, the agents play a two-player minimax game, namely,

$$\min_G \max_D L(G, D = \min_G \max_D \mathbb{E}_{\mathbf{x}\sim P_{data}(\mathbf{x})}[logD(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim P_z(\mathbf{z})}[logD(1-G(\mathbf{z}))]),$$

where $P_{data}(\mathbf{x})$ is the probability of sampling $\mathbf{x}$ from the training distribution and $P_z(\mathbf{z})$ the probability of sampling the noise vector $z$ from the used noise distribution $P_z$. Furthermore, it can be shown that if the discriminator is trained until convergence in each iteration and if the generator is updated to improve its loss function in each iteration, then probability of drawing a sample $\mathbf{x}$ from the learnt generator distribution converges to $P_{data}(\mathbf{x})$.

Pinto et al. [5] proposed a reinforcement learning method in order to increase the learnt policy's robustness with repsect to sim2sim and sim2real errors. In this approach, called *Robust adversarial reinforcement learning* (RARL), the protagonist agent is trained in an adversarial environment. In this setting, an adversary, which is able to apply perturbations to the protagonists actions, is trained jointly with the protagonist. This approach can be formalised as a two-player $\gamma$-discounted zero-sum Markov game [64], using the tuple $(\mathscr{S}, \mathscr{A}_P, \mathscr{A}_A, \mathscr{P}, \mathscr{R}, \mu_0, \gamma)$, where $\mathscr{A}_P$ and $\mathscr{A}_A$ define the continuous action spaces of the protagonist and the adversary respectively. The transition probabilty is modeled as a probability dependent on the state and both agents actions $\mathscr{P} : \mathscr{S} \times \mathscr{A}_P \times \mathscr{A}_A -> \mathscr{S}$. The reward signal $\mathscr{S} \times \mathscr{A}_P \times \mathscr{A}_A -> \mathbb{R}$ is identical for both agents. In a $\gamma$-discounted zero-sum game, the protagonist seeks to maxmise the expected discounted return, while the adversary tries to minimise it. In each timestep, both the protagonist and the adversary observe the state $\mathbf{s}$ and choose actions $\mathbf{a}_P$ and $\mathbf{a}_P$ respectively. Next, the state transitions to $\mathbf{s}_{t+1} = \mathscr{P}(\mathbf{s}_t, \mathbf{a}_{P,t}, \mathbf{a}_{A,t})$ and both agents receive the reward signal $\mathbf{r}_t = \mathscr{R}(\mathbf{s}_t, \mathbf{a}_{P,t}, \mathbf{a}_{A,t})$. RARL trains the policies of the two agents in an alternating procedure. First, the protagonist samples from the environment with a fixed adversary. Using the collected rollouts, the protagonist's policy is then optimised. In the case of RARL, the TRPO algorithm is used to update policies. Next, the same procedure is repeated for the adversary: the adversary collects rollouts against a fixed protagonist and subsequently updated using TRPO. This alternating process is then repeated for $N_{iter}$ iterations. RARL in combination with TRPO as its policy optimisation method has been shown to improve generalisation and increase robustness with respect to sim2sim errors in comparison to standard TRPO on several robotic control tasks using the MuJoCo simulator [65].

Having provided a brief survey of the underlying concepts related to our algorithm, we will introduce an adversarial reinforcement learning framework, which is able to learn general policies that are able to transfer from simulation to the real world.

# 3 Deep adversarial reinforcement learning

In this chapter, we will present our method called *Adversarial reinforcement learning* (ARL), which is our approach to efficiently learn robust policies, that generalise well to new or changing situations. While adversarial learning has shown several recent successes [8, 9], it still remains a relatively unresearched field.

## 3.1 Motivation

The performance of reinforcement learning algorithms often depends on finding a trade-off between the explotiation of the best currently known policy and the exploration of the environment to find potentially better policies at the risk of taking suboptimal steps. In many modern RL algorithms, this trade-off is commonly dealt with by allowing a high exploration rate at the beginning of training which is slowly decreased over time. The policy for exploration often consists of simply taking a random action at each time step with a certain probability. In real-world applications, training in the real environment is generally expensive, time-intensive and potentially dangerous. All these factors lead to a scarcity of data, which in turn makes it undesirable to explore the environment in a randomised fashion. Many approaches to exploration use randomly sampled actions in order to explore the environment. Methods that learn deterministic policies $\pi(\mathbf{s})$ commonly employ an $\epsilon$-greedy version of the policy in order to ensure exploration of the environment. This method uses the hyperparameter $0 < epsilon < 1$, also called the *exploration rate*, to trade-off eploitation and exploration. With a probability of $\epsilon$, a random action is sampled from the action space, while the deterministic policy $\pi$ is followed with a probability of $1 - \epsilon$. Typically, training begins with a high exploration rate close to 1, which is gradually decreased as training progresses. Stochastic policies inherently explore the environment as they are probabilty distributions over actions dependent on the state. However, neither of these approaches actively directs exploration to areas of potential interest and are therefore more likely to converge to local optima. Furthermore, the lack of sufficient training scenarios can lead to overfitting, which causes the learned policies not to be able to generalise to test scenarios that differ from ones it was trained on. Another approach to circumvent the scarcity of available training data when learning a policy for real-world applications is to train a policy in a simulator and then transfer it to the real world. However, this approach highly depends how well the simulation is able to model the real world environment. Often the difference between the simulator and reality is too large to successfully transfer the policy if the learned policy is not robust enough to handle modeling errors. Furthermore, random exploration further increases sample complexity. As a result of these issues, we are required to find an approach that is able to find policies that are able to generalise over various different scenarios (e.g. different starting positions) while requiring a significantly smaller amount of training data.

One approach to increase robustness, proposed by Pinto et al. [5], is to model uncertainties as an adversary agent that perturbs the system by applying additional, adversarial forces. This adversary trained together with a second agent, the protagonist, whose goal is to perform the original task while being robust to the perturbations generated by the adversary. This method, called *Robust adversarial reinforcement learning* (RARL), trains both agents in turn against a fixed opponent. In each training iteration, the currently trained agent A collects samples from the environment against a fixed opponent agent B which are the used to update agent A. Subsequently, the roles are reversed and samples are collected in order to update the policy of agent B.

While RARL achieves its goal to increase the robustness of the learnt policy, it does not address the issue of random exploration. Furthermore, RARL only adresses the robustness with respect to the system's transition dynamics and not to its initial probabilities. For example, the task of disentangling two objects may begin from different initial object configurations in the test scenario than anything previously seen during training as exhaustively exploring the task space is too expensive. Hence, we require an approach to direct exploration to areas of the task space in which the current policy performs poorly.

## 3.2 Adversarial reinforcement learning

In this scetion, we will present our adversarial reinforcement learning framework by providing an intiuition of the main approach as well as a formal definition of the algorithm.

### 3.2.1 The ARL framework

The main idea of our apporach is to steer the agent into regions in which the current policy performs poorly and requires further information on. In order to achieve this targeted exploration, we train an adversarial agent that effectively chooses the protagonist's starting position at the beginning of each training episode. The adversary's task is to find areas in the task space which pose difficulties for the protagonist and is rewarded for moving the agent into situations that are expected to yield low returns for the protagonist under its current policy. At the beginning of each episode, the agent follows the adverarial policy for a fixed number of time steps after which the protagonist takes over again. The adversary's and protagonist's policies are trained jointly in order to allow the adversary to adapt to the changing protagonist policy. Our learning framework consists of an MDP enviroment $E$ of horizon $H_P$ and two policies, the *protagonist* $\pi_P$ and the *adversary* $\pi_A$, which both interact with the same environment. The aim of the ARL framework is to learn a protagonist's policy $\pi_P$ that performs well in as many areas of the environment as possible. In order to encourage directed exploration of difficult areas, the original RL framework is extended. At the begining of each training episode, the agent will act for $H_A$ time steps under policy $\pi_A$ and subsequently follow the protagonist policy $\pi_P$ for $H_P$ time steps. We can formulate the protagonist's objective function in the following way:

$$J(\pi_P) = \mathbb{E}_{\pi_P(\mathbf{a}|\mathbf{s}),\mu_P(\mathbf{s}_0)}\left[\sum_{t=0}^{H_P}\gamma^t r(\mathbf{s},\mathbf{a},\mathbf{s}')\right]. \tag{3.1}$$

It should be noted that the initial state probabilty $\mu_P(\mathbf{s}_0)$ in Equation 3.1 is dependent on the adversary's policy $\pi_A$ and formulated as

$$\mu_P(\mathbf{s}) = \sum_{\tau} P_{\pi_A}(\tau|\mathbf{s}_{H_A} = \mathbf{s}). \tag{3.2}$$

The adversary's goal is to find states $\mathbf{s}$ in the MDP in which the protagonist's expected return is low. As defined in Chapter 3, the state-value function $V^\pi$ is a mathematical tool to estimate the expected return of following policy $\pi$ starting in state $\mathbf{s}$, a concept that is used by many RL algorithms. Thus, we can define a reward function $r(\mathbf{s},\mathbf{a},\mathbf{s}')$ that evaluates the adversary's action through the protagonist's negative state-value of the reached state, that is,

$$r_A(\mathbf{s},\mathbf{a},\mathbf{s}';\pi_P) = -V_P s(\mathbf{s}'),$$

where $V_P$ is the current protagonists estimated state-value function. With this choice of reward function, the adversary is encouraged explore the environment in order find states in which the protagonist's policy is expected to perform poorly. We can formulate the adversary's objective as

$$J(\pi_A) = \mathbb{E}_{\pi_A(\mathbf{a}|\mathbf{s}),\mu(\mathbf{s}_0)}\left[\sum_{t=0}^{H_A}\gamma^t r_(\mathbf{s},\mathbf{a},\mathbf{s}';\pi_P)\right].$$

In order to jointly train our agents, we alternate between improving the adversary's policy and improving the protagonist's policy. We begin by training the adversary for $K_A$ episodes. At the beginning of each episode, the environment is reset to one of its initial states sampled from $\mu_0$. Next, at every timestep $t$, the adversary selects an action using its policy $\mathbf{a}_t^{(A)}$ and observes the next state $\mathbf{s}_{t+1}$. Then, the adversary's reward is computed using the current approximation of the protagonist's state-value function, i.e. $r_t^{(A)}$. The tuple $(\mathbf{s}_t, \mathbf{a}_t^{(A)}, r_t^{(A)}, cvecs_{t+1})$ is added to the adversary's replay buffer, from which subsequently a minibatch of transitions is sampled in order to update the adversary's policy. After $H_A$ steps, it is the protagonist's turn to interact with the environment. Note that the environment is not reset before the protagonist begins its interactions, thus the protagonist's initial position is $\mathbf{s}_{H_A}$. The protagonist collects tuples $(\mathbf{s}_t, \mathbf{a}_t^{(P)}, r_t^{(P)}, \mathbf{s}_{t+1})$ for $H_P$ timesteps. The tuples are added to the protagonist's replay buffer $D_A$, however protagonist's policy is not updated during these rollouts. Once the protagonist has completed its interaction with the environment after $H_P$ timesteps, the environments is reset. This process is repeated for $K_A$ episodes in which the adversary's policy is updated at each time step, while the protagonist simply collects experiences to store in its replay buffer $D_P$. After $K_A$ rollouts, it is the protagonist's turn to be improved. At the beginning of each rollout the environment is reset. Next, the adversary interacts with the environment for $H_A$ timesteps in order to generate the protagonist's starting for the protagonist, while storing the experienced transitions in its replay buffer $D_A$. After the adversary has completed $H_A$ actions, it is the protagonist's turn again. For $H_P$ time steps t, the protagonist samples action $\mathbf{a}_t^{(P)} \sim \pi_P(\mathbf{s}_t)$, observes the new state $\mathbf{s}_{t+1}$ and receives the reward $r_t^{(P)}$. Next, the tuple $(\mathbf{s}_t, \mathbf{a}_t^{(P)}, r_t^{(P)}, \mathbf{s}_{t+1})$ is added to the replay buffer $D_P$ and the protagonist's policy $\pi_P$ and state-value function approximation $V_P$ are updated using a minibatch of samples drawn from $D_P$. This sequence of alternating between training the adversary and training the protagonist is repeated for a fixed number of iterations $N$. It should

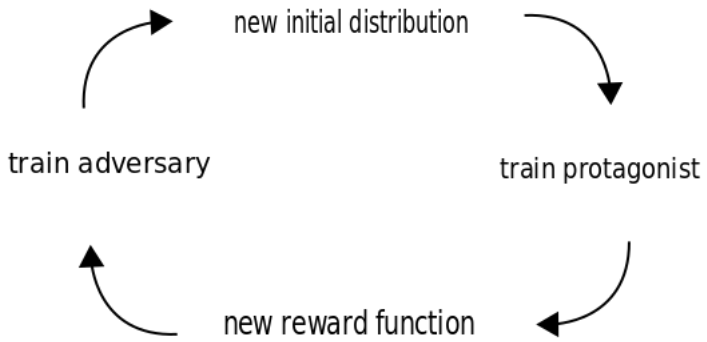



**Figure 3.1.:** ARL training begins by improving the adversary, which implicitly modifies the initial probability of states for the protagonist. Thus, the protagonist needs to improve its policy to learn to deal with this new distibution. As the protagonist changes its policy, the adversary's reward function is modified. Hence, the adversary needs to update its policy again. This process is repeated for a fixed number of iterations.

be noted that the ARL framework introduces the additional hyperparameters $K_A$, $K_P$ and $H_A$. Furthermore, it is possible to use any off-policy RL method can be used in the ARL framwork, as long as the protagonist's state value function $V_P$ is approximated. Algorithm 1 shows the pseudocode of our proposed method, while Figure 3.1 presents a high level visualisation of the ARL learning framework. Additionally, it is also possible to use the ARL framework in combination with on-policy RL methods (see Appendix A).

## 3.3 Prioritised experience replay for ARL

Learning in an adversarial environment such as the one created by our method, positive rewards are likely to be more sparse that in a non-adversarial environment, as training is performed in the presence of an opponent whose sole purpose is to prevent the trained agent from achieving high returns. Hence, policies may converge to suboptimal local extema due to the lack of sufficient positive examples in the learning process. Consider the simple scenario in which an agent is placed in an environment consisting of two adjoining rooms that are connected by a single door. The agent's goal is to simply move from its starting postion to a goal position, which is located in the adjoining room without running into any walls. The agent is rewarded with a large positive reward for reaching its goal and penalised for colliding with the wall. All other actions yield a reward of zero. The main difficulty of this task is to find a trajectory that safely moves the agent through the narrow door. RL methods that do not sufficiently explore the environment may converge to a locally optimal policy of simply staying safely in the starting room as the agent never encounters the second room at all. This problem becomes even more prominent with the presence of an adversary that chooses the agents starting position, as the adversary will attempt to place the agent in positions where it is likely to collide. Hence, the agent is less likely to find the goal position, which, in turn, leads to an increased risk of converging to a suboptimal policy. In the following, we propose a novel approach to prioritise past experiences when using a replay buffer, in order to circumvent this issue.

### 3.3.1 Experience replay

We begin by introducing the general idea of using replay buffers, a method known as *experience replay* (ER) [66]. Instead of using the collected samples immediately, transition tuples $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ can be stored in a *replay buffer* $\mathcal{D}$ of a finite capacity $C$. During every policy improvement step, k transitions are sampled from the replay buffer and used to calulate the policy update. Once the buffer has reached its maximum capacity, new experiences replace the oldest existing experiences in storage in a *first-in-first-out* (FIFO) manner. Standard experience replay then samples stored transitions from the replay buffer using a uniform distribution. One of the main advantages of experience replay is the increased data efficiency as previous experiences can be used in multiple policy updates. This allows to perform several policy

**Algorithm 1:** Off-policy adversarial reinforcement learning

**Input:** Arbitrary initial policies $\pi_A$ and $\pi_P$ with replay buffers $D_A$ and $D_P$, Environment $E$,

**for** $i \leftarrow 0$ **to** $N$ **do**
  **for** $j \leftarrow 0$ **to** $K_A$ **do**
    `reset(`$E$`)` ;
    **for** $t \leftarrow 0$ **to** $H_A$ **do**
      $(\mathbf{s}_t, \mathbf{a}_t^{(A)}, r_t^{(A)}, \mathbf{s}_{t+1}) \leftarrow$ `step(`$E$, $\pi_A$`)`;
      $D_A \leftarrow D_A \cup (\mathbf{s}_t, \mathbf{a}_t^{(A)}, r_t^{(A)}, \mathbf{s}_{t+1})$;
      $\pi_A \leftarrow$ `train(`$\pi_A$, $D_A$`)` ;
    **for** $t \leftarrow 0$ **to** $H_P$ **do**
      $(\mathbf{s}_t, \mathbf{a}_t^{(P)}, r_t^{(P)}, \mathbf{s}_{t+1}) \leftarrow$ `step(`$E$, $\pi_P$`)`;
      $D_P \leftarrow D_P \cup (\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$;
  **for** $j \leftarrow 0$ **to** $K_P$ **do**
    `reset(`$E$`)` ;
    **for** $t \leftarrow 0$ **to** $H_A$ **do**
      $(\mathbf{s}_t, \mathbf{a}_t^{(A)}, r_t^{(A)}, \mathbf{s}_{t+1}) \leftarrow$ `step(`$E$, $\pi_A$`)`;
      $D_A \leftarrow D_A \cup (\mathbf{s}_t, \mathbf{s}_t^{(A)}, r_t^{(A)}, \mathbf{s}_{t+1})$;
    **for** $t \leftarrow 0$ **to** $H_P$ **do**
      $(\mathbf{s}_t, \mathbf{a}_t^{(P)}, r_t^{(P)}, \mathbf{s}_{t+1}) \leftarrow$ `step(`$E$, $\pi_P$`)`;
      $D_P \leftarrow D_P \cup (\mathbf{s}_t, \mathbf{a}_t^{(P)}, r_t^{(P)}, \mathbf{s}_{t+1})$;
      $\pi_P \leftarrow$ `train(`$\pi_P$, $D_P$`)` ;

improvement steps without having to interact with the environment which generally may be very time- or cost-intensive, depending on the task at hand. Furthermore, using experience replay breaks the temporal correlations between sampled transitions and thus restores the i.i.d. assumption of the underlying stochastic gradient descent methods methods that are commonly used for optimisation, i.e., Adam or RMSProp. ER is commonly used in state-of-the-art deep RL algorithms such as SAC or DDPG.

### 3.3.2 Prioritised experience replay

In 2015, Schaul et al. introduced a method called *prioritised experience replay* (PER) [67] which prioritises stored transitions depending on how useful a transition will be for improving the current policy. PER modifies the probability distribution of sampling a transition $t_i$ from the replay buffer, $P_{\mathscr{D}}(\mathbf{t_i})$, by assigining a priority value $p_i$ to each transition $t_i$. The aim of this approach is to assign higher priorities to experiences that contain more useful information than others as training progesses. Some transitions may be less redundant or task-relevant than others and therefore will more likely be more effective during training. To achieve this, PER assigns prioritities to each transition based on a priority function. The assigned priorities are then used to calculate the probability of an experience being replayed by

$$P_{\mathscr{D}}(i) = \frac{p_i^\alpha}{\sum_i p_i^\alpha},$$

where $P_{\mathscr{D}}(i) > 0$ is the probability that the transition $t_i$ will be sampled and the hyperparamter $\alpha$ determines to what extent prioritisation will be used. The choice of $\alpha = 0$ corresponds to uniform sampling as in vanilla experience replay. Schaul et al. propose to use the transitions absolute TD-error as a measure of how useful a transition will be for the policy update as the TD-error of a transition can be interpreted as a measure of how unexpected a transition is [67]. Thus, the priority $p_i$ of transition $t_i$ is computed as

$$p_i = |\delta_i| + \epsilon,$$

where $\delta_i$ is the TD-error of the i-th transiton from the reward buffer. A small $\epsilon > 0$ is added to the absolute TD-error in order to ensure positive priorities for all transitions. Another variant of PER uses a rank-based method to calculate the transitions' probabilities, namely,

$$p_i = \frac{1}{rank(i)},$$

where $rank(i)$ is the rank of transition $t_i$ when the replay buffer is sorted with respect to the absolute TD-error. As updating every transitions in the data buffers with a large capacity in every iteration is computationally expensive, only the priorities that were sampled for each iteration are updated. Note that this approach for updating priorities means that have a low TD-error on the first visit may not be replayed for a long time (or at all if the replay buffer's capacity is low).

#### Importance sampling weights

Since estimating an expected value using stochastic updates requires the updates distribution to correspond to the expectations' distribution. As PER will assign higher probailities to experiences with large priority values, sampling those transitions more frequently than others. This changes the sampling distribtion from the 'original' uniform distribution which is used in vanilla ER, causing the solution to converge to a different estimate. To remedy this bias, Schaul et al. use *Importance Sampling* (IS) weights to compensate for the difference from the uniform distribution of ER. These IS-weights are computed as by

$$w_i = \left(\frac{1}{P(i)} \cdot \frac{1}{C}\right)^{\beta},$$

where C is the size of the replay buffer and $\beta$ is an additional hyperparameter that controls to what extent the weights are used. For a choice of $\beta = 1$, the bias is fully corrected.

### 3.3.3 Advantage-based experience replay

As PER uses the absolute TD-error, not only strong positive rewards are prioritised, but also strong negative rewards, which dampens the desired effect of sampling transitions that highlight positive trajectories. In order to further increase the likelihood of sampling transitions that lead to states with high expected returns, we propose *Advantage-based prioritisation* (ABER). ABER can be seen as an extension of PER, as we only modify the way a transition's priority is computed. As our goal is to increase the probabilities of sampling that improve the current policy, we propose assign high priorities that have lead to improved expected returns compared to following the current policy. Hence, we can prioritise transitions based on their exponential advantage, namely,

$$p_i = exp(A(s_i, a_i)).$$

The advantage value can be approximated by either using learnt approximations of $Q$ and $V$ or by using the transitions TD-error. Taking the exponential of the advantage ensures that transtions that lead to high expected returns. In addition, we have tested a second variant of ABER, which uses the max operator instead of the exponential, that is,

$$p_i = max(A(s_i, a_i), \epsilon),$$

where $\epsilon$ is the minimum priority value. In our experiments, we refer to this method by the name ABER_MAX.

# 4 Experiments

In this chapter, we will first introduce the experimental task that was used to test our approach to ARL, including descriptions of the disentangling environment, implementation details and the actual hardware setup. Next, we will present the results of the three sets of experiments that were carried out in order to evaluate our algorithm. Finally, we will give brief discussions on the results on each experment set.

## 4.1 The disentangling task

We performed all of our experiments in an environment, which we call the *Robot-arm disentangling environment* (RADE). In RADE, the agent is controlling a seven degree-of-freedom (DoF) KUKA LBR iiwa robotic arm, equipped with a SAKE gripper endeffector which is holding an "S"-shaped object. The environment further contains a small styrofoam table on which a second, "O"-shaped object is placed. Figure 4.1 show the experimental hardware setup. The robotic arm is controlled through direct joint actions, i.e. the agent's actions consist of sending joint deltas for all seven robot joints at each time step. These deltas are added onto the robots current joint positions, which will be the destination for the robot in this time step. Actions that would cause the robot to exceed a joint limit are clipped at the limit. Once the robot has finished the move, the next time step begins. In the initial state, the robot is holding the object in such a position that the two objects are entangled and can not "easily" be separated from each other. The agent's goal is to disentangle the objects in a limited amount of steps without causing a collision or leaving its allowed work space. The task is considered as solved if the euclidean distance between the objects passes a threshold of 0.5 meters.

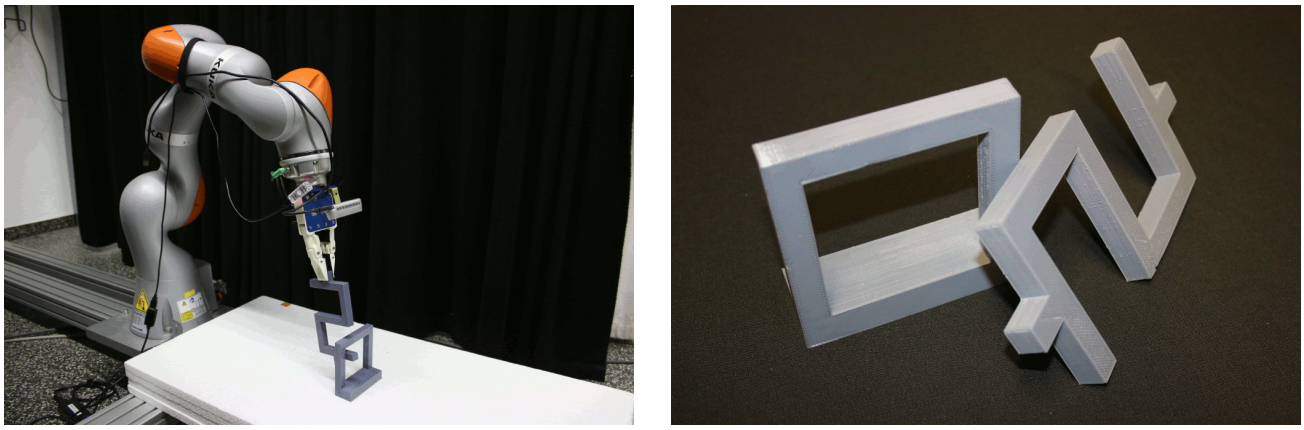


**Figure 4.1.:** The picture on the left shows the KUKA iiwa robot arm with attached SAKE gripper holding the object in an entangled position. Right: The 3D printed disentanling objects: the "O-shape" and the "S-shape".

The agent receives a positive reward for successfully solving the disentangling task. Collisions are penalised with a negative reward that depends on the current time step, where early collisions are punished more severly than later ones. Otherwise a small action penalty is given to the agent in order encourage smaller steps. Thus, the reward function for this task is formalised as,

$$r(\mathbf{s}, \mathbf{a}, \mathbf{s}') = \begin{cases} -1\frac{1-\gamma^{H-t+1}}{1-\gamma^{H-t}}, & \text{if collision detected,} \\ 1, & \text{if } \|\mathbf{s}' - \mathbf{t}\|_2 \geq d_{thresh}, \\ -\|\mathbf{a}\|_2, & \text{otherwise.} \end{cases}$$

The experiments with both, the simulated and the real robot setup, were designed with the aim to answer the following research questions:

1. Does ARL learn more general policies than vanilla SAC?
2. Can advantage-based experience replay significantly improve learning speed for ARL?
3. How robust is ARL with relation to its hyperparameters?

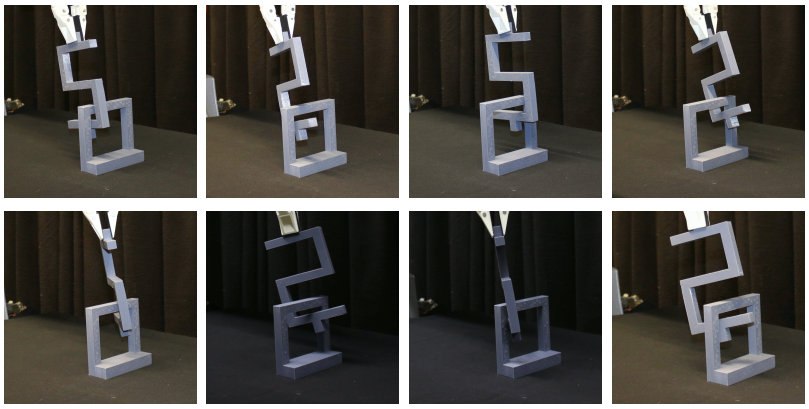


**Figure 4.2.:** The top row of images shows the 4 initial positions from the training set, the bottom row the entaglements from the test set. Note for both sets the S-shape is positioned in the two top corners of the O-shape, once from each side. The test set consists of slight variations of the positions from the training set.

## 4.2 Evaluation methods

The learned protagonist polices were evaluated in two distinct ways to measure and compare their respective performances. The first method evaluates a policy in simulation on a "test set" of initial positions. This test set consists of four different joint configurations in which the objects are entangled. These configurations have not been used to pose as the environments initial state for training. To measure how well an agent is able to generalise to new situations, we evaluate the protagonists' performance over 100 episodes while sampling the initial position uniformly from this test set. Figure **??** shows possible initial position from both the training set and the test set.
Secondly, we evaluate the learned policy's performance by running it in the real robot setting. The starting robot configuration is the same test set as in the simulation evaluation.

## 4.3 Implementation details

In order to facilitate experiments, we created a simulated version of the disentangling environment. The existing robot simulator and controller *Simulation Lab* (SL) [68] was used, both, for simulation and to control the real robots. The joint goto-commands were sent from the policy via the *Robot Communication interface* (robcom) [69]. Furthermore, we implemented an extension to SL by creating a headless version of the simulators interface, which does not produce any graphical outputs. Using this headless variant of SL allowed us to run multiple simultated experiments in parallel by utilising the *Lichtenberg High Performance Computer* (HPC). Furthermore, we implemented a RADE gym environment as an extension to the existing *OpenAI gym* environments [70]. Using this standardised environment interface allowed us to easily use readily available implementations of state-of-the-art RL methods like TRPO or SAC from the *OpenAI baselines* [71] and its fork *stable-baselines* [72] projects for our experiments.

## 4.4 Results

We will now present the results of three distinct sets of experiments that have been performed in order to answer the previously stated research questions.

### 4.4.1 Ablation study for adversarial reinforcement learning

In order to determine which variant of ARL shows the best performance and to which hyperparameters ARL is most sensitive to, we carried out a small ablation study, comparing various hyperparameter settings. Our main focus in these experiments were the influence of the number of episodes each agent is trained for per iteration. Another question of interest was how the choice the adversary's horizon $H_A$ affects the learning process. To analyse the effects of the hyperparameters $K_A$, $K_P$ and $H_A$, we trained three different variants of the adversarial SAC algorithm with varying choices of $K_A$ and $K_P$. Hereby, we chose to set $K_A = K_P$ to allow an equal number of training episodes for both protagonist and adversary. Hence, we decided the following three variants of adversarial SAC for our ablation study:

- **ASAC10**: Adversarial SAC with $K_A = K_P = 10$

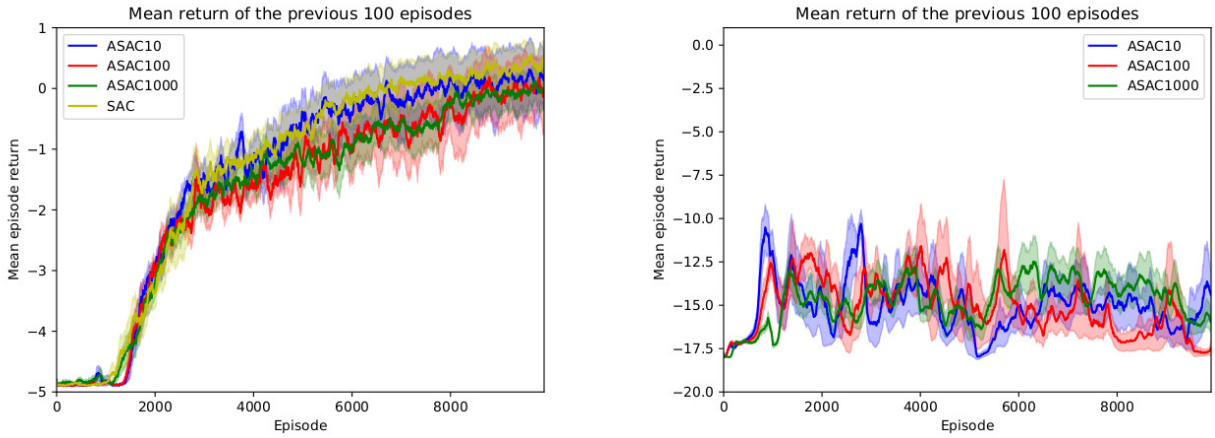
- **ASAC100**: Adversarial SAC with $K_A = K_P = 100$
- **ASAC1000**: Adversarial SAC with $K_A = K_P = 1000$

It should be noted that all policies were trained for the same number of episodes. Thus, the number of training iterations $N$ depends on the choice of $K_A$ and $K_P$, i.e. to train ASAC10 for ten thousand episodes, one thousand iterations were required, while training ASAC1000 for the same amount of episodes only 10 training iterations are required.

We tested all three variants with three different choices for the adversary's horizon and compared them against a baseline of standard SAC. In the following, we will present and discuss the results of this ablation study.

### Twenty-step adversary

We first chose to test the adversarial SAC variants with an adversary, which is allowed 20 interactions with the environment to create difficult situations for the protagonist. All policies consisted of a feedforward neural network with 2 hidden layers of 64 units each. All SAC hyperparameters were set equally, performing 5 optimisation steps for every step in the environment with a learning rate $\alpha$ of 0.0003. The policies were trained for ten thousand episodes each.




**(a)** Protagonist training performance

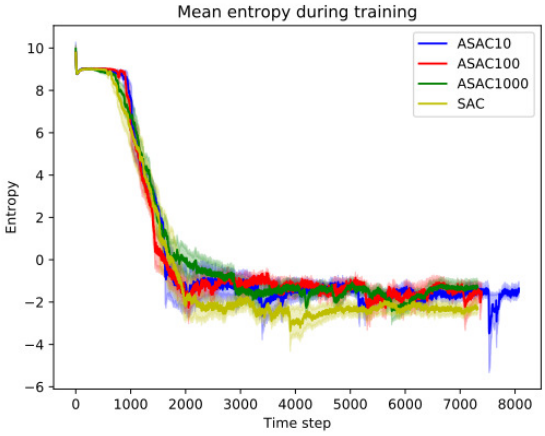
**(b)** Adversary training performance

**Figure 4.3.:** Comparison of ASAC mean return for different numbers of episodes per iteration and a SAC baseline during training in the RADE environment with a 20-step adversary. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.

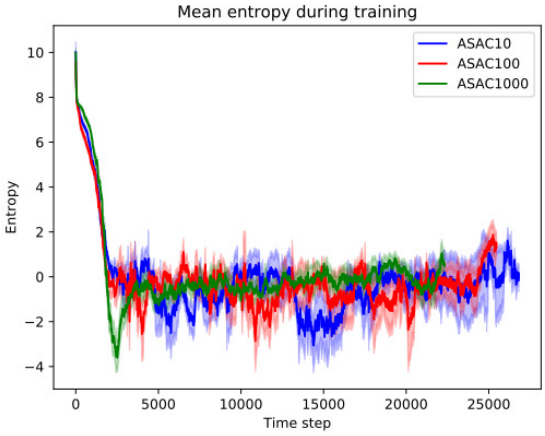
Figure 4.3 shows the learning curves of both the protagonist and the adversary. The protagonist's performance increases in a similar fashion for all three variants of ASAC. The adversary's learning curve appears to alternate between improvement and decline of performance periodically, where larger values for $K = K_A = K_P$ seem to cause less frequent yet larger oscillations. These oscillations are caused by the changing opponent policies. Similar observations can be made in terms of the policies entropies in Figure 4.4, after an initial drop in entropy while the policies learn to avoid collisions. It should be noted that the protagonist's policy entropy remains higher for all variants of ASAC than for SAC, indicating that adversarial training helps to learn more general policies.

To explicitly test how well each of the learnt protagonist policies generalise, we evaluated them on the test set described in Section 4.2.

Table 4.1 shows the performance of the learnt protagonist policies on both the training set and the test set. While the mean return of all ASAC variants on the training set not significantly worse than standard SAC, they all achieved significantly better results on our test set.




**(a)** Protagonist entropy




**(b)** Adversary entropy

**Figure 4.4.:** Comparison of ASAC entropy for different numbers of episodes per iteration and a SAC baseline during training in the RADE environment with a 20-step adversary. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.

| Method | Training | | Test set | |
|---|---|---|---|---|
| | score | success rate (%) | score | success rate (%) |
| SAC | $0.44 \pm 0.29$ | $90.5 \pm 4.84$ | $-3.90 \pm 0.39$ | $17.0 \pm 6.63$ |
| ASAC10 | $0.20 \pm 0.51$ | $86.4 \pm 8.65$ | $-2.56 \pm 0.31$ | $39.7 \pm 5.27$ |
| ASAC100 | $0.07 \pm 0.43$ | $84.2 \pm 7.31$ | $-3.01 \pm 0.28$ | $32.13 \pm 4.78$ |
| ASAC1000 | $-0.42 \pm 0.63$ | $76.0 \pm 10.67$ | $-2.81 \pm 0.36$ | $35.0 \pm 6.39$ |

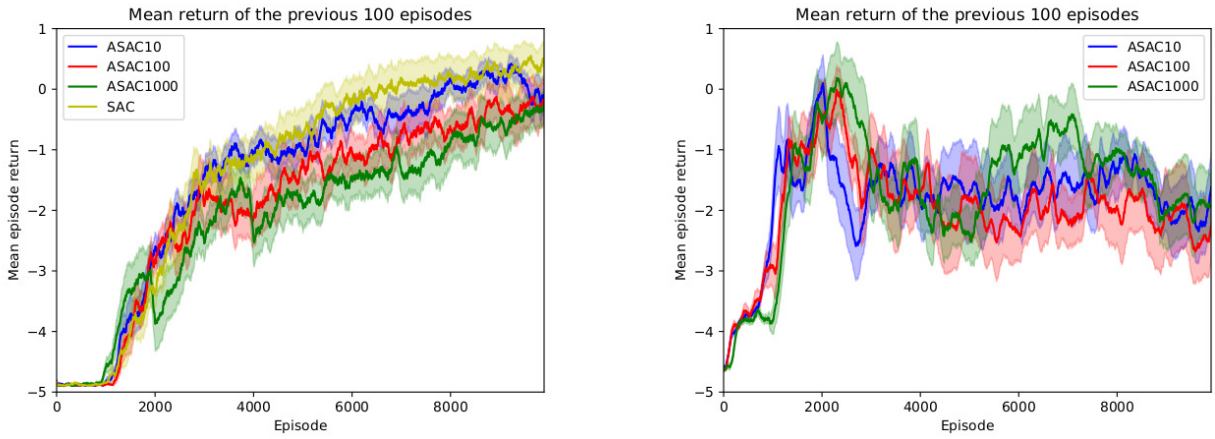
**Table 4.1.:** Comparison of ASAC performance on the training set and the test set for differnet numbers of episodes per iteration during training with a 20-step adversary. The first two columns show the mean scores and success rates of the last 100 episodes of training including the standard error of the mean. The second two columns show each methods performance on the test set over 100 trials. All results are averaged over 10 different runs.

### Five-step adversary

Subsequently, we trained the same set of policies in the RADE environment with an adversary horizon of $H_A = 5$. All other hyperparamters remained the same as in the previous experiment. Once again, each policy was trained for ten thousand episodes and evaluated on the test set in simulation.

Figure 4.5 shows the average return of the last 100 episodes during training for the protagonist and adversary policies. Similarly to the training with a 20-step adversary, the ASAC variants achieve a slightly lower performance than standard SAC. The protagonists' learning curves show the same overall trajectory as SAC while also showing oscillations that once again are stronger and for larger numbers of training episodes per iteration. This observation is caused by the fact that with ASAC10, the policies need to adapt to a changed opponent more frequently than the policies trained with ASAC1000. However, the oscillations magnitude is smaller for variants with less training episodes per iteration because the opponents' policies remain more similar to the ones from the previous iterations. The adversaries' learning curves show the same oscillations. After an initial peak in performance as both policies learn to avoid collisions, the adversaries' mean return slowly decreases as the protagonist's policies improve their policies overall, making it harder for the adversaries to find state the pose difficulties for the protagonists. Figure 4.6 shows the entropy of both the protagonist and the adversary policies' entropies. Once again, we can observe an overall higher entropy for policies that were trained in the adversarial framework.

Table 4.2 shows the training set and test set evaluation of all ASAC variants trained with a 5-step adversary in the RADE setting. Once again, while the ASAC variants show lower mean returns and success rates on the training set, they out-




**(a)** Protagonist training performance

**(b)** Adversary training performance

**Figure 4.5.:** Comparison of ASAC performance on the training set and the test set for different numbers of episodes per iteration and a SAC baseline during training in the RADE environment with a 5-step adversary. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.

| Method | Training | | Test set | |
|---|---|---|---|---|
| | score | success rate (%) | score | success rate (%) |
| SAC | $0.44 \pm 0.29$ | $90.5 \pm 4.84$ | $-3.90 \pm 0.39$ | $17.0 \pm 6.63$ |
| ASAC10 | $-0.18 \pm 0.35$ | $80.0 \pm 6.01$ | $-3.57 \pm 0.41$ | $22.6 \pm 6.96$ |
| ASAC100 | $-0.51 \pm 0.49$ | $74.34 \pm 8.28$ | $-3.03 \pm 0.43$ | $31.78 \pm 7.26$ |
| ASAC1000 | $-0.26 \pm 0.25$ | $78.6 \pm 4.25$ | $-2.74 \pm 0.37$ | $36.7 \pm 6.21$ |

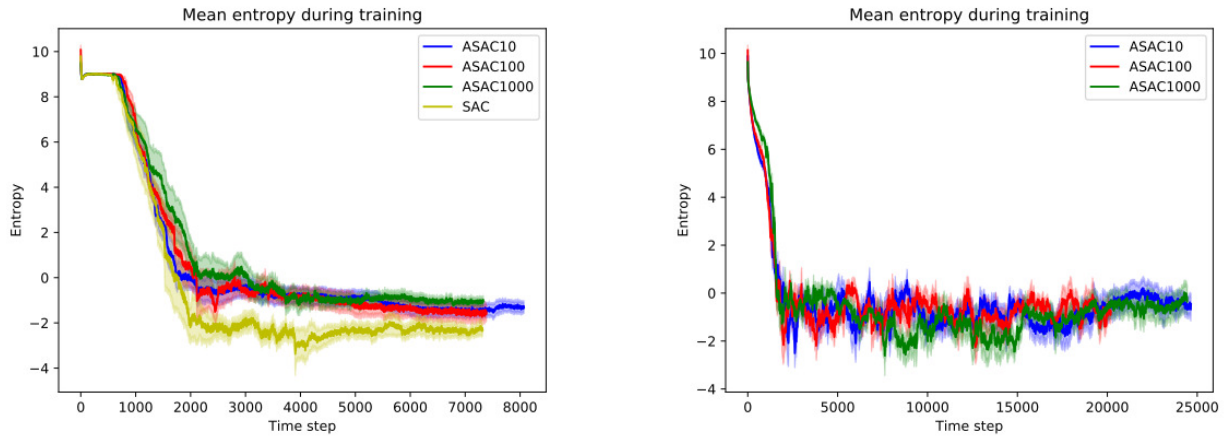
**Table 4.2.:** Comparison of ASAC performance on the training set and the test set for differnet numbers of episodes per iteration during training with a 5-step adversary. The first two columns show the mean scores and success rates of the last 100 episodes of training including the standard error of the mean. The second two columns show each methods performance on the test set over 100 trials. All results are averaged over 10 different runs.

perform SAC on the test set. It should be noted that while ASAC10 achieved the best test set performance for training with a 20-step adversary, ASAC1000 reaches the best results for training with the 5-step adversary.

### Single-step adversary

For our third set of experiments, we evaluated the SAC and ASAC methods with a single-step adversary, i.e. $H_A = 1$. Again, training lasted for ten thousand episodes and subsequently evaluated on the test set in simulation.

Figure 4.7 shows the mean training return during training for both the protagonist and the adversary. We can observe that, once again, the ASAC protagonist learning curves show a trajectory as SAC. The adversaries' learning curves are much more stable than in the previous experiments, however light oscillations can still be detected. This observation can be explained by the adversaries' small horizon. As the only a single action is available to the adversary, only a very limited region state space is reachable for the adversary. Thus, a smaller amount of initial positions for the protagonist can be generated. As the protagonist learns to adapt to these positions, the adversary is unable to further explore the state space for more difficult regions due to its limited horizon. Figure 4.8 shows the policies entropies during training. While the overall trajectory for both the protagonist's and the adversary's entropies look very similar, it should be noted that the adversary's entropy drops to a slightly lower level. This lower entropy of the adversary's policy is explained by the limited horizon with the same reasoning used for the adversary's converging mean return.




**(a)** Protagonist entropy

**(b)** Adversary entropy

**Figure 4.6.:** Comparison of ASAC entropy for different numbers of episodes per iteration and a SAC baseline during training in the RADE environment with a 5-step adversary. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.

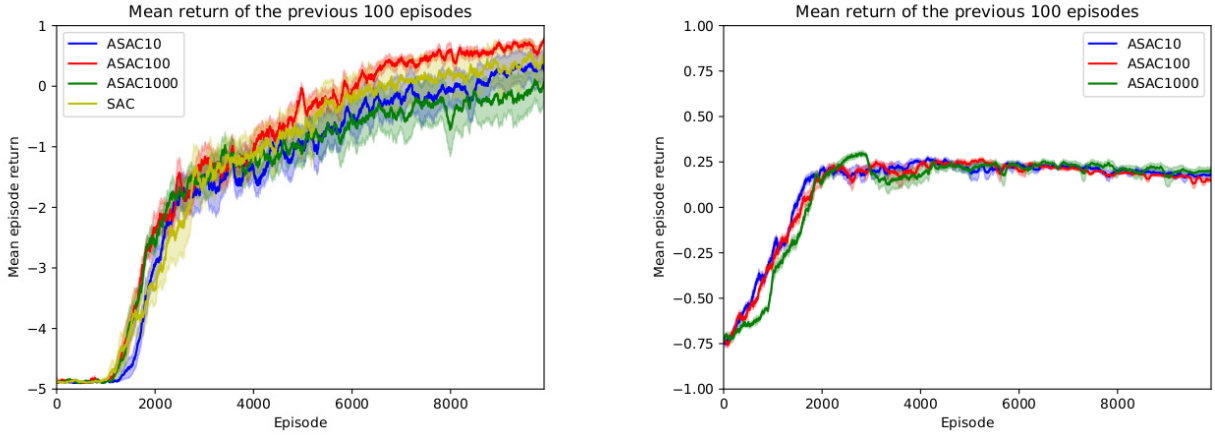
| Method | Training | | Test set | |
|---|---|---|---|---|
| | mean return | success rate (%) | mean return | success rate (%) |
| SAC | $0.44 \pm 0.29$ | $90.5 \pm 4.84$ | $-3.90 \pm 0.39$ | $17.0 \pm 6.63$ |
| ASAC10 | $0.30 \pm 0.29$ | $88.1 \pm 4.99$ | $-2.56 \pm 0.31$ | $39.7 \pm 5.27$ |
| ASAC100 | $0.70 \pm 0.05$ | $94.88 \pm 0.83$ | $-3.01 \pm 0.28$ | $32.13 \pm 4.78$ |
| ASAC1000 | $0.00 \pm 0.33$ | $83.1 \pm 5.60$ | $-2.81 \pm 0.36$ | $35.0 \pm 6.39$ |

**Table 4.3.:** Comparison of ASAC performance on the training set and the test set for differnet numbers of episodes per iteration during training with a 1-step adversary. The first two columns show the mean scores and success rates of the last 100 episodes of training including the standard error of the mean. The second two columns show each methods performance on the test set over 100 trials. All results are averaged over 10 different runs.

Table 4.4 shows the training set and test set evaluation of all ASAC variants trained with a 1-step adversary in the RADE setting. While ASAC10 and ASAC1000 show slightly lower mean returns and success rates than SAC, ASAC100 outperforms SAC on the training set. On the test set, all adversarial variants show significantly better results than SAC, with ASAC10 achieving the best overall mean return and rate of solving the disentangling task.

### Real robot performance

Finally, we evaluated two of the trained policies on the real robot setup in order to investigate how well the policies are able to generalise from simulation to the real-world scenario. As a baseline, we tested the policy learned by standard SAC. We compared this baseline against the best learnt policy from our previous experiments, i.e. ASAC10 with a 20-step adversary. We compared the algorithms by running 3 trials on each starting position from both the training set and the test set. The results were averaged over 5 different runs of each algorithm. Table **??** shows the mean returns and success rates of SAC and ASAC10 for the real robot experiments. We can observe from the results that ASAC not only achieves significantly better results on the test set but also shows a better performance than SAC on the training set. These findings indicate that ARL does indeed improve generalisation from simulation to real-world scenarios as well as unseen test scenarios.




**(a)** Protagonist training performance

**(b)** Adversary training performance

**Figure 4.7.:** omparison of ASAC mean return for different numbers of episodes per iteration and a SAC baseline during training in the RADE environment with a 1-step adversary. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.

| Method | Training | | Test set | |
|---|---|---|---|---|
| | mean return | success rate (%) | mean return | success rate (%) |
| SAC | $-0.44 \pm 0.55$ | $80.0 \pm 9.35$ | $-4.31 \pm 0.36$ | $10.0 \pm 6.12$ |
| ASAC10 | $0.41 \pm 0.59$ | $90.0 \pm 10.00$ | $-2.44 \pm 1.02$ | $41.67 \pm 17.28$ |

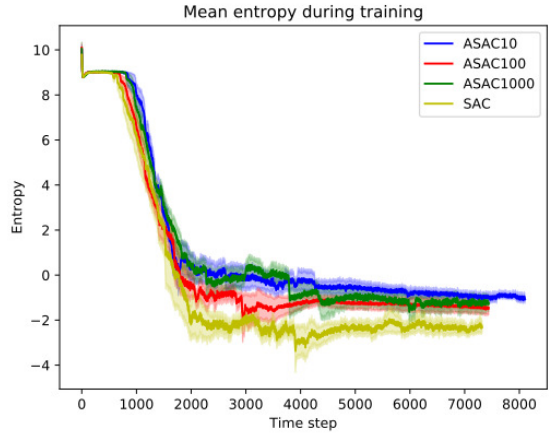
**Table 4.4.:** Comparison of ASAC10 performance on the training set and the test set on the real robot. The first two columns show the mean scores and success rates of the last 100 episodes of training including the standard error of the mean. The second two columns show each methods performance on the test set over 3 trials per scenario. All results are averaged over 5 different runs.

### 4.4.2 Performance of advantage-based experience replay

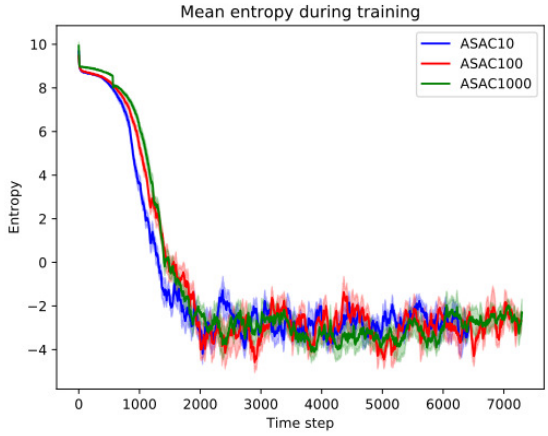
Our second set of experiments was created to investigate whether advantage-based experience replay can boost learning speed and improve overall performance. We trained various policies using the SAC algorithm combined with different versions of experience replay. As a baseline, we used a uniform replay buffer as in 'vanilla' SAC. We then trained policies using SAC in combination with PER, ABER and ABER_MAX, both, with and without including importance weights for bias-correction. In the following, we will refer to the variants of PER, ABER and ABER_MAX that use importance weights as PER_W, ABER_W and ABER_MAX_W. All experiments were performed using the same hyperparameters for ten thousand episodes.

Figure 4.9 shows the mean return of the last 100 episodes during training averaged over five runs on a slightly modified version of the RADE environment in which the magnitude of the collision penalty is not dependent on the time step, namely, a collision results in a penalty of -1. We can observe that both variants of ABER_MAX do not help the learning process, but actually lead to a significantly worse performance. Both variants of PER and ABER yield positive returns faster than standard SAC. However, the PER variants slow down in improvement after roughly 5000 episodes while ABER contiues to improve to solve the training examples in nearly all attempts. Especially ABER_W achieves a big improvement over plain SAC. The entropies of all versions converge to a similar value. Table 4.5 shows the final performance of all trained policies on the training set.

Next, we performed the same set of experiments in the unmodified RADE environment, in which positive rewards are much less sparse due to the time dependent collision penalty. Figure 4.10 shows the mean return and the entropy during training averaged over 10 runs. We can observe once again that the ABER_MAX variants fail to improve learning, causing negative effects instead. ABER does not improve the overall performance as it did in the modified RADE environment. All variants of ABER and PER achieve a similar overall performance as standard SAC. However, it should be noted that

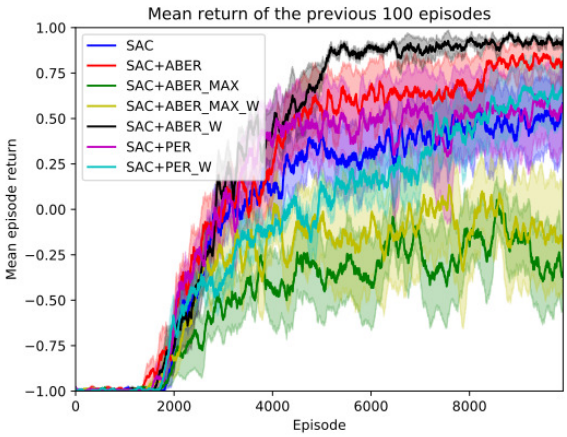



**(a)** Protagonist entropy

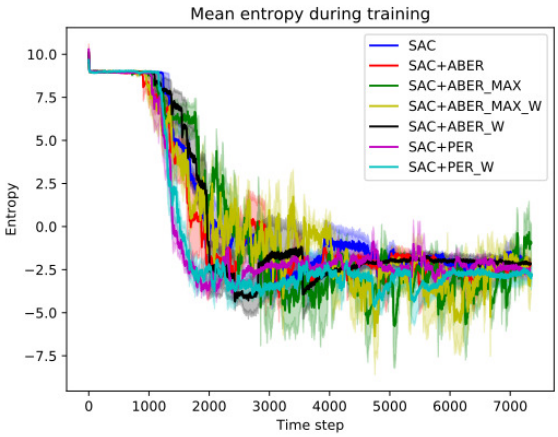



**(b)** Adversary entropy

**Figure 4.8.:** Comparison of ASAC entropy for different numbers of episodes per iteration and a SAC baseline during training in the RADE environment with a 1-step adversary. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.




**(a)** Training performance




**(b)** Entropy

**Figure 4.9.:** Comparison of PER, ABER, ABER_MAX and uniform experience replay mean return and entropy during training. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.

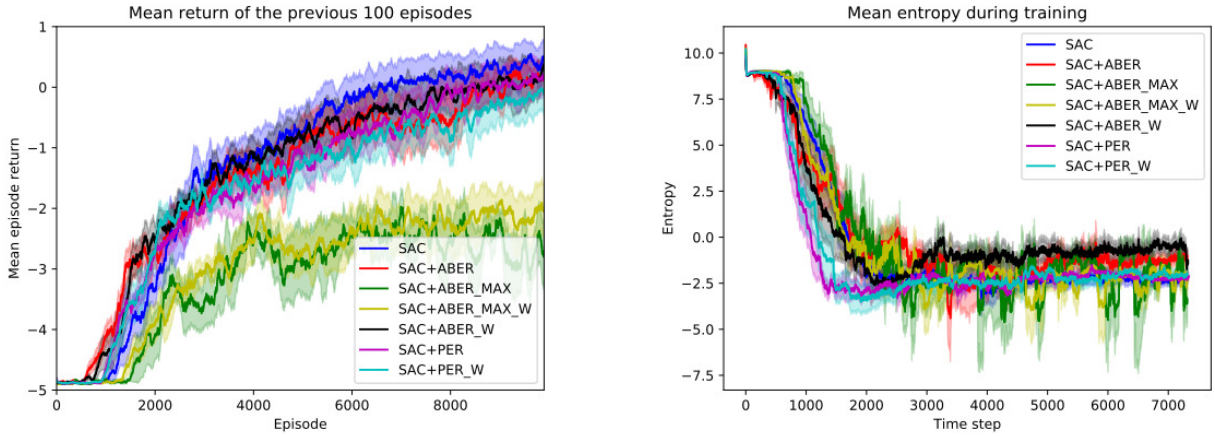
the ABER variants start to increase in performance earlier than SAC and PER. This observation indicates that ABER might indeed be helpful in sparse environments, as the beginning of training in the RADE environment has a relatively sparse reward structure due to the fact that many actions in the initial position lead to collisions.

### 4.4.3 Performance of adversarial reinforcement learning with prioritised experience replay

In our final set of experiments, we evaluated ARL can be improved and stabilised by introducing experience replay prioritisation, namely PER and ABER. We compared the basic ASAC100 variant to versions that use the weighted variants PER and ABER to sample experiences in the RADE setting with a 20-step adversary. Figure 4.11 shows the mean return of the previous 100 episodes during training for both the adversary and the protagonist. We can observe that the protagonists all with a similar improvement rate. It should be noted that ASAC100+ABER and ASAC100+PER both achieve positive more quickly than ASAC. However, the overall performance of all trained protagonists is on an equal level. The adversaries' performance reaches an early peak for ASAC100+ABER but then quickly declines to a very low level. This indicates that the adversary frequently collides early, even in late stages of training. A similar trend can be observed for ASAC+PER.

| Method | Training | |
|---|---|---|
| | mean return | success rate (%) |
| SAC | $0.58 \pm 0.21$ | $78.8 \pm 10.57$ |
| SAC+PER | $0.55 \pm 0.22$ | $77.4 \pm 11.01$ |
| SAC+ABER_MAX | $-0.35 \pm 0.20$ | $32.6 \pm 10.09$ |
| SAC+ABER | $0.81 \pm 0.09$ | $90.4 \pm 4.29$ |
| SAC+PER_W | $0.66 \pm 0.19$ | $77.4 \pm 11.01$ |
| SAC+ABER_MAX_W | $-0.19 \pm 0.29$ | $40.4 \pm 14.41$ |
| SAC+ABER_W | $0.93 \pm 0.03$ | $96.4 \pm 1.57$ |

**Table 4.5.:** Comparison of PER, ABER, ABER_MAX and uniform experience replay on sparse RADE performance. The two columns show the mean scores and success rates of the last 100 episodes of training including the standard error of the mean. All results are averaged over 10 different runs.




**(a)** Training performance

**(b)** Entropy

**Figure 4.10.:** Comparison of PER, ABER, ABER_MAX and uniform experience replay mean return and entropy during training. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.

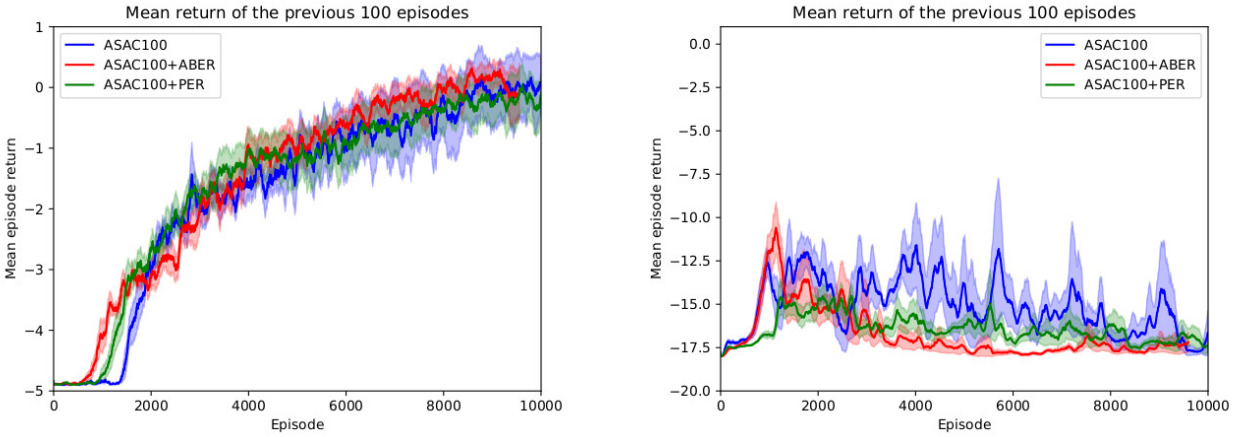
Figure 4.12 shows the entropy during training for both the adversaries and the protagonists. While all curves follow a similar trajectory, ASAC100+ABER learns a policiy with the highest entropy. Furthermore, it should be noted that the reason for the longer entropy curve for ASAC100 compared is caused by the fact that the adversary of this variant takes more timesteps overall du to later collisions.

Table 4.7 shows the results of ASAC100, ASAC100+PER and ASAC100+ABER on the training and test sets. We can observe that neither ABER nor PER are able to improve the results of standard ASAC.

Next, we repeated the previous experiment using ASAC1000 in combination with PER and ABER in the RADE environment with a 5-step adversary. Figure 4.13 shows the performance of ASAC1000, ASAC1000+PER and ASAC1000+ABER during training averaged over 10 runs each. We can observe a similar trend as in the previous experiment for the agents' performances. The incorporation of ABER causes the adversary to reach an early peak and then continually decline to a very low level as training progresses. Once again, the bad performance is likely caused by many early collision in later stages of training. This claim can be supported by noticing the very short entropy curve of ASAC1000+ABER in Figure 4.14. However, ASAC1000+ABER achieves the best performance on the test scenarios, despite the bad performance of the adversary in late training episodes. The performance of ASAC1000, ASAC1000+PER and ASAC1000+ABER is shown in Table 4.8.

| Method | Training | |
|---|---|---|
| | mean return | success rate (%) |
| SAC | $0.44 \pm 0.29$ | $90.5 \pm 4.84$ |
| SAC+PER | $0.12 \pm 0.31$ | $85.1 \pm 5.22$ |
| SAC+ABER_MAX | $-2.75 \pm 0.54$ | $36.0 \pm 9.68$ |
| SAC+ABER | $0.12 \pm 0.20$ | $85.11 \pm 3.47$ |
| SAC+PER_W | $-0.11 \pm 0.37$ | $81.2 \pm 6.33$ |
| SAC+ABER_MAX_W | $-2.04 \pm 0.42$ | $48.4 \pm 7.20$ |
| SAC+ABER_W | $0.35 \pm 0.16$ | $89.0 \pm 2.70$ |

**Table 4.6.:** Comparison of PER, ABER, ABER_MAX and uniform experience replay on sparse RADE performance. The two columns show the mean scores and success rates of the last 100 episodes of training including the standard error of the mean. All results are averaged over 10 different runs.

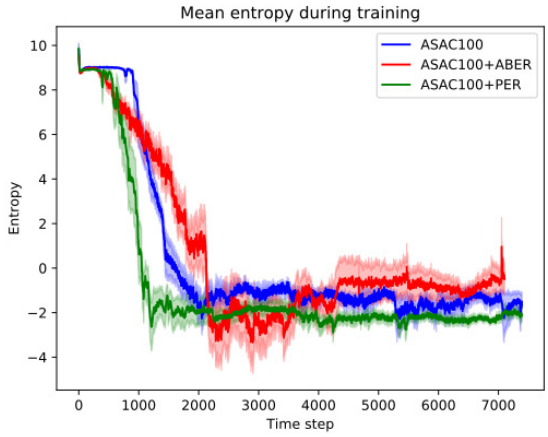



**(a)** Protagonist training performance

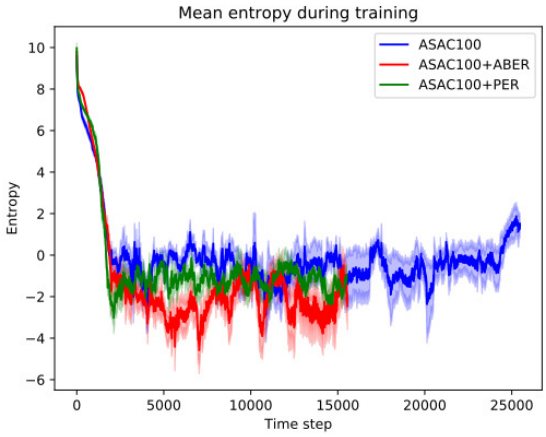
**(b)** Adversary training performance

**Figure 4.11.:** Comparison of ASAC mean return combined with PER and ABER with a 20-step adversary in the disentangling environment. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.

Finally, we evaluated ABER10 in combination PER and ABER in the RADE environment using a 1-step adversary. Figures 4.15 and 4.16 show the adversaries' and protagonists' mean return and the entropy during training. We can observe similar learning curves for all used methods. Table 4.7 shows that neither PER nor ABER are able to improve the test scenario performance of ASAC10 in the RADE environment with a 1-step adversary.




**(a)** Protagonist entropy

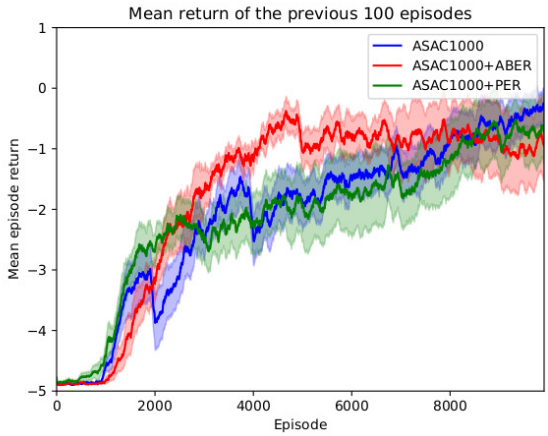



**(b)** Adversary entropy

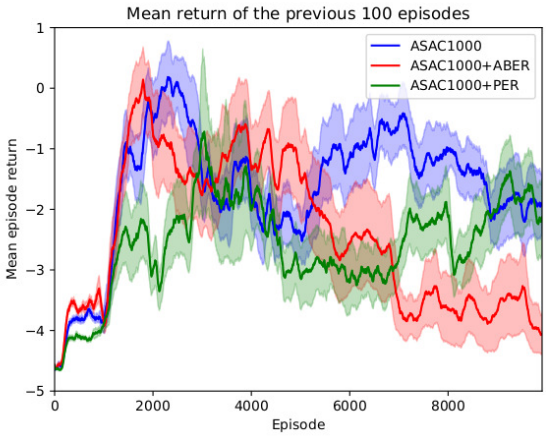
**Figure 4.12.:** Comparison of ASAC entropy combined with PER and ABER with a 20-step adversary in the disentangling environment. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.

| Method | Training | | Test set | |
|---|---|---|---|---|
| | mean return | success rate (%) | mean return | success rate (%) |
| ASAC100 | $0.30 \pm 0.29$ | $88.1 \pm 4.99$ | $-2.56 \pm 0.31$ | $39.7 \pm 5.27$ |
| ASAC100+PER | $-0.13 \pm 0.36$ | $80.78 \pm 6.16$ | $-3.62 \pm 0.26$ | $21.78 \pm 4.41$ |
| ASAC100+ABER | $-0.06 \pm 0.43$ | $79.38 \pm 7.47$ | $-2.87 \pm 0.25$ | $34.5 \pm 4.28$ |

**Table 4.7.:** Comparison ASAC performance on the training set and the test set in combination with PER and ABER. The first two columns show the mean scores and success rates of the last 100 episodes of training. The second two columns show each methods performance on the test set over 100 trials. All results are averaged over 10 different runs.

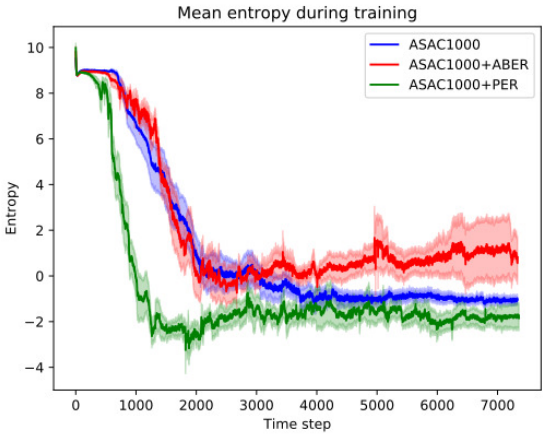



**(a)** Protagonist training performance

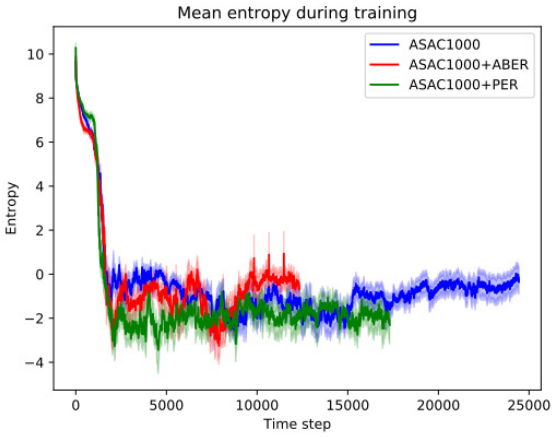



**(b)** Adversary training performance

**Figure 4.13.:** Comparison of ASAC mean return combined with PER and ABER with a 5-step adversary in the disentangling environment. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.




**(a)** Protagonist entropy

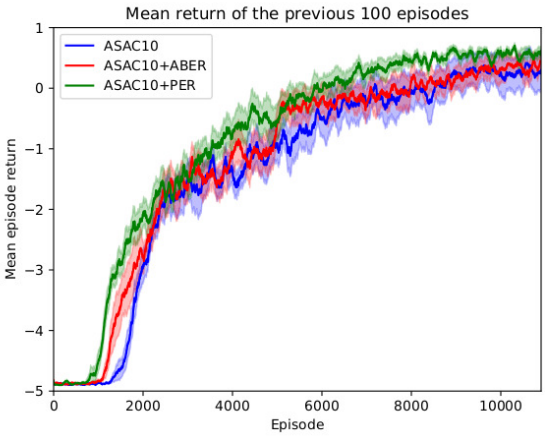



**(b)** Adversary entropy

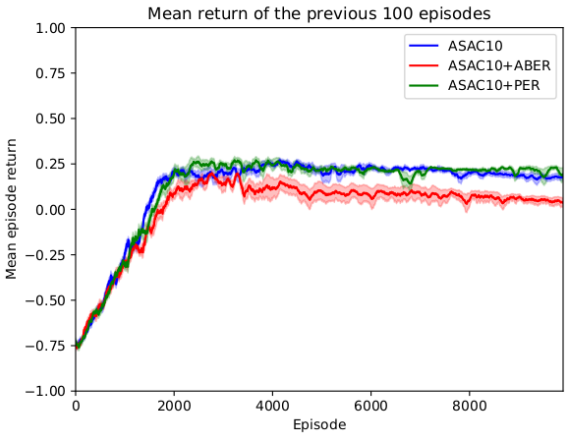
**Figure 4.14.:** Comparison of ASAC entropy combined with PER and ABER with a 5-step adversary in the disentangling environment. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.

| Method | Training | | Test set | |
|---|---|---|---|---|
| | mean return | success rate (%) | mean return | success rate (%) |
| ASAC1000 | $0.30 \pm 0.29$ | $88.1 \pm 4.99$ | $-2.56 \pm 0.31$ | $39.7 \pm 5.27$ |
| ASAC1000+PER | $-0.70 \pm 0.53$ | $71.2 \pm 9.05$ | $-4.11 \pm 0.22$ | $13.40 \pm 3.78$ |
| ASAC1000+ABER | $-0.95 \pm 0.59$ | $66.89 \pm 10.02$ | $-2.38 \pm 0.25$ | $42.78 \pm 4.28$ |

**Table 4.8.:** Comparison ASAC performance on the training set and the test set in combination with PER and ABER. The first two columns show the mean scores and success rates of the last 100 episodes of training. The second two columns show each methods performance on the test set over 100 trials. All results are averaged over 10 different runs.

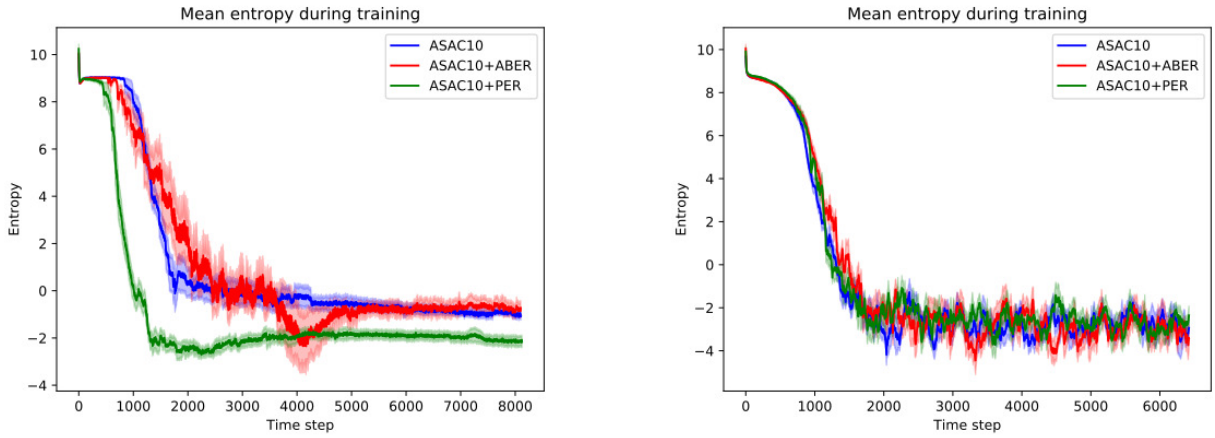



**(a)** Protagonist training performance




**(b)** Adversary training performance

**Figure 4.15.:** Comparison of ASAC mean return combined with PER and ABER with a 1-step adversary in the disentangling environment. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.




**(a)** Protagonist entropy

**(b)** Adversary entropy

**Figure 4.16.:** Comparison of ASAC entropy combined with PER and ABER with a 1-step adversary in the disentangling environment. Episode returns are averaged over 10 runs. Shaded areas show the standard error of the mean.

| Method | Training | | Test set | |
|---|---|---|---|---|
| | mean return | success rate (%) | mean return | success rate (%) |
| ASAC10 | $0.30 \pm 0.29$ | $88.1 \pm 4.99$ | $-2.56 \pm 0.31$ | $39.7 \pm 5.27$ |
| ASAC10+PER | $0.59 \pm 0.07$ | $93.2 \pm 1.17$ | $-3.07 \pm 0.52$ | $31.0 \pm 8.73$ |
| ASAC10+ABER | $0.44 \pm 0.13$ | $90.44 \pm 2.14$ | $-3.60 \pm 0.30$ | $22.0 \pm 5.08$ |

**Table 4.9.:** Comparison ASAC performance on the training set and the test set in combination with PER and ABER. The first two columns show the mean scores and success rates of the last 100 episodes of training. The second two columns show each methods performance on the test set over 100 trials. All results are averaged over 10 different runs.

# 5 Conclusion and future work

In this thesis, we introduced a novel adversarial learning framework, ARL, that trains agents to learn more general policies by introducing an adversary to the learning process in order to steer exploration to unseen and difficult regions of the environment. As the adversary is implicitly generating new training scenarios for the protagonist, the protagonist's policy is less likely to overfit to the set of training scenarios. In order to be able to evaluate our method on the challenging object disentangling with a robotic arm, we extended the existing robot simulation tool, SL, by implementing a headless version. This headless version of SL allowed us to run multiple simulated experiments in parallel on the in order to perform an extensive evaluation of the proposed ARL training method. Furthermore, as learning in an adversarial environment can lead to sparsity of positive rewards, we proposed a new method of prioritising past experiences from a replay buffer, based on a transition tuple's advantage estimate. The advantage-based experience replay method samples transitions that are likely to improve the current policy more frequently than others.

To evaluate both ARL and ABER, we performed multiple experiments both in simulation and on the real robot. Our experiemts included an ablation study in order to investigate the effects of changing hyperparameters of ARL, namely the adversary's horizon $H_A$ and the number of training episodes per iteration $K_A$ and $K_P$ for each policy. The results of the ablation study show that the performance of ARL is robust to the choice of $K_A$ and $K_P$ as all tested choices led to a significantly improved performance on the test scenarios. The adversary's horizon does impact the learning process as adversaries with too short horizons $H_K$ lack the possibility to explore larger parts of the environment's state space. However, even a single step adversary was able to improve the protagonist's ability to generalise. Furthermore, ARL is indeed able to generalise from training in simulation to deployment in the real world. Additionally, we investigated the how well ABER can improve learning in comparison to standard ER and PER. Our experiments show that ABER can improve learning speed and data-efficiency in environments with sparse distributions of positive rewards. However, a more extensive evaluation of ABER is required in order to draw a more meaningful conclusion about its potential use cases.

The results of our experiments create several natural starting points for possible future works. First, an interesting question remains how ARL performs with different base algorithms other than SAC. Also on-policy methods can potentially be used in the ARL framework (for pseudocode of on-policy ARL see Appendix A). Furthermore, as an evaluation on other task than robotic object disentangling was out of the scope of this thesis, investigating the performance of ARL in other environments remains a topic for future research. Another interesting question is to investigate the possibility incorporating shared critic that evaluates both the actions of the protagonist and the adversary. With this extension, learning speed and data-efficiency might be further improved. Also, as during this thesis only FNN policies were used, an exploring how using other types of neural networks such as CNNs or RNNs could further improve performance on the disentangling task is an additional possible future extention of our approach. Finally, as mentioned above, a more exhaustive investigation of ABER is required in order to be able to draw a more conclusive evaluation of the potential of advantage-based experience replay.

# Bibliography


[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[2] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Permenter, T. Koolen, P. Marion, and R. Tedrake, "Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot," *Autonomous Robots*, vol. 40, 07 2015.

[3] D. Ferrucci, A. Levas, S. Bagchi, D. Gondek, and E. Mueller, "Watson: Beyond jeopardy!," *Artificial Intelligence*, vol. 199-200, pp. 93–105, 07 2013.

[4] G. Tesauro, "Temporal difference learning and td-gammon," *Commun. ACM*, vol. 38, pp. 58–68, Mar. 1995.

[5] L. Pinto, J. Davidson, R. Sukthankar, and A. Gupta, "Robust adversarial reinforcement learning," *ICML*, 2017.

[6] P. F. Christiano, Z. Shah, I. Mordatch, J. Schneider, T. Blackwell, J. Tobin, P. Abbeel, and W. Zaremba, "Transfer from simulation to real world through learning deep inverse dynamics model," *CoRR*, vol. abs/1610.03518, 2016.

[7] A. A. Rusu, M. Vecerík, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, "Sim-to-real robot learning from pixels with progressive nets," *CoRR*, vol. abs/1610.04286, 2016.

[8] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 2672–2680, Curran Associates, Inc., 2014.

[9] J. Ho and S. Ermon, "Generative adversarial imitation learning," in *Advances in Neural Information Processing Systems 29* (D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, eds.), pp. 4565–4573, Curran Associates, Inc., 2016.

[10] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[11] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2009.

[12] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[13] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[14] M. P. Deisenroth, G. Neumann, and J. Peters, "A survey on policy search for robotics," *Foundations and Trends® in Robotics*, vol. 2, no. 1–2, pp. 1–142, 2013.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[16] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, 04 2014.

[17] R. E. Bellman, *Adaptive control processes: a guided tour*, vol. 2045. Princeton university press, 2015.

[18] R. Sutton, "Learning to predict by the method of temporal differences," *Machine Learning*, vol. 3, pp. 9–44, 08 1988.

[19] G. Rummery and M. Niranjan, "On-line q-learning using connectionist systems," *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

[20] C. Watkins and P. Dayan, "Technical note: Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 05 1992.

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *SCIENCE*, vol. 220, no. 4598, pp. 671–680, 1983.

[22] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," *Evol. Comput.*, vol. 9, pp. 159–195, June 2001.

[23] P.-T. de Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of Operations Research*, vol. 134, pp. 19–67, Feb 2005.

[24] A. Cauchy, "Méthode générale pour la résolution des systemes d'équations simultanées," 1847.

[25] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, pp. 229–256, May 1992.

[26] I. Grondman, L. Busoniu, G. A. D. Lopes, and R. Babuska, "A survey of actor-critic reinforcement learning: Standard and natural policy gradients," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, pp. 1291–1307, Nov 2012.

[27] W. S. McCulloch and W. Pitts, "Neurocomputing: Foundations of research," ch. A Logical Calculus of the Ideas Immanent in Nervous Activity, pp. 15–27, Cambridge, MA, USA: MIT Press, 1988.

[28] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, pp. 65–386, 1958.

[29] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, Oct. 1986.

[31] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, pp. 193–202, Apr 1980.

[32] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, pp. 541–551, Dec 1989.

[33] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, pp. 1527–1554, July 2006.

[34] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[35] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov 1998.

[36] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, (USA), pp. 807–814, Omnipress, 2010.

[37] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: the rprop algorithm," in *IEEE International Conference on Neural Networks*, pp. 586–591 vol.1, March 1993.

[38] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[39] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.

[40] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, pp. 82–97, Nov 2012.

[41] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," pp. 6645–6649, 01 2013.

[42] H. Sak, O. Vinyals, G. Heigold, A. W. Senior, E. McDermott, R. Monga, and M. Z. Mao, "Sequence discriminative distributed training of long short-term memory recurrent neural networks," in *INTERSPEECH*, 2014.

[43] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature Cell Biology*, vol. 521, pp. 436–444, 5 2015.

[44] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR09*, 2009.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (USA), pp. 1097–1105, Curran Associates Inc., 2012.

[46] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[47] D. Cireşan and J. Schmidhuber, "Multi-column deep neural networks for offline handwritten chinese character classification," 09 2013.

[48] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014.

[49] A. Bordes, S. Chopra, and J. Weston, "Question answering with subgraph embeddings," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 615–620, Association for Computational Linguistics, Oct. 2014.

[50] H. J. Kelley, "Gradient theory of optimal flight paths," 1960.

[51] A. E. Bryson, "A gradient method for optimizing multi-stage allocation processes," in *Proc. Harvard Univ. Symposium on digital computers and their applications*, 1961.

[52] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 12 2013.

[53] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[54] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 1889–1897, PMLR, 07–09 Jul 2015.

[55] J. Peters, K. Mülling, and Y. Altün, "Relative entropy policy search," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pp. 1607–1612, AAAI Press, 2010.

[56] A. Abdolmaleki, R. Lioutikov, J. R. Peters, N. Lau, L. Pualo Reis, and G. Neumann, "Model-based relative entropy stochastic search," in *Advances in Neural Information Processing Systems 28* (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), pp. 3537–3545, Curran Associates, Inc., 2015.

[57] C. Daniel, G. Neumann, O. Kroemer, and J. Peters, "Hierarchical relative entropy policy search," *Journal of Machine Learning Research*, vol. 17, no. 93, pp. 1–50, 2016.

[58] R. Akrour, G. Neumann, H. Abdulsamad, and A. Abdolmaleki, "Model-free trajectory optimization for reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning* (M. F. Balcan and K. Q. Weinberger, eds.), vol. 48 of *Proceedings of Machine Learning Research*, (New York, New York, USA), pp. 2961–2970, PMLR, 20–22 Jun 2016.

[59] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, 09 2015.

[60] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *ArXiv*, vol. abs/1801.01290, 2018.

[61] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," *ArXiv*, vol. abs/1812.05905, 2018.

[62] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," *31st International Conference on Machine Learning, ICML 2014*, vol. 1, 06 2014.

[63] B. Polyak, "New stochastic approximation type procedures," *Avtomatica i Telemekhanika*, vol. 7, pp. 98–107, 01 1990.

[64] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Proceedings of the Eleventh International Conference on International Conference on Machine Learning*, ICML'94, (San Francisco, CA, USA), pp. 157–163, Morgan Kaufmann Publishers Inc., 1994.

[65] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control.," in *IROS*, pp. 5026–5033, IEEE, 2012.

[66] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, vol. 8, pp. 293–321, May 1992.

[67] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *CoRR*, vol. abs/1511.05952, 2015.

[68] S. Schaal, "The sl simulation and real-time control software package," 2009.

[69] R. Lioutikov, O. Kroemer, G. Maeda, and J. Peters, "Learning manipulation by sequencing motor primitives with a two-armed robot," vol. 302, pp. 1601–1611, 01 2016.

[70] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[71] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines." `https://github.com/openai/baselines`, 2017.

[72] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines." `https://github.com/hill-a/stable-baselines`, 2018.

# A Pseudocode for on-policy ARL

**Algorithm 2:** On-policy adversarial reinforcement learning

**Input:** Arbitrary initial policies $\pi_A$ and $\pi_P$, Environment $E$,
**for** $i \leftarrow 0$ **to** $N$ **do**
  **for** $j \leftarrow 0$ **to** $K_A$ **do**
    `reset(`$E$`)` ;
    **for** $t \leftarrow 0$ **to** $H_A$ **do**
      $(s_t, a_t^{(A)}, r_t^{(A)}, s_{t+1}) \leftarrow$ `step(`$E$, $\pi_A$`)`;
      $\pi_A \leftarrow$ `train(`$\pi_A$`)` ;
    **for** $t \leftarrow 0$ **to** $H_P$ **do**
      $(s_t, a_t^{(P)}, r_t^{(P)}, s_{t+1}) \leftarrow$ `step(`$E$, $\pi_P$`)`;
  **for** $j \leftarrow 0$ **to** $K_P$ **do**
    `reset(`$E$`)` ;
    **for** $t \leftarrow 0$ **to** $H_A$ **do**
      $(s_t, a_t^{(A)}, r_t^{(A)}, s_{t+1}) \leftarrow$ `step(`$E$, $\pi_A$`)`;
    **for** $t \leftarrow 0$ **to** $H_P$ **do**
      $(s_t, a_t^{(P)}, r_t^{(P)}, s_{t+1}) \leftarrow$ `step(`$E$, $\pi_P$`)`;
      $\pi_P \leftarrow$ `train(`$\pi_P$`)` ;