Guided Reinforcement Learning Under Partial Observability

Geführtes Reinforcement Learning unter partieller Beobachtbarkeit
Master-Thesis von Stephan Weigand aus Karlstadt
August 2019

Guided Reinforcement Learning Under Partial Observability
Geführtes Reinforcement Learning unter partieller Beobachtbarkeit

Vorgelegte Master-Thesis von Stephan Weigand aus Karlstadt

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: Dr. Joni Pajarinen
3. Gutachten:

Tag der Einreichung:

Erklärung zur Master-Thesis

Erklärung zur Abschlussarbeit gemäß § 23 Abs. 7 APB der TU Darmstadt

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

Mir ist bekannt, dass im Fall eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

In the submitted thesis the written copies and the electronic version are identical in content.

Datum / Date:	Unterschrift / Signature:
____________________	______________________________

Abstract

Due to recent breakthroughs in artificial intelligence, reinforcement learning (RL) gained considerable attention in the last years. Especially combining RL with deep neural networks, which could already achieve great success in other research fields, has contributed to these breakthroughs. In many real-world scenarios it is not always possible to perceive the true and complete state of the environment. These scenarios are known as learning under uncertainty or under partial observability. Formulating such problems as partially observable Markov decision processes (POMDPs) allows us to solve decision making processes under uncertainty. Guided RL approaches address this problem by supporting the RL algorithms with additional state information during the learning process to increase their performance for solving POMDPs. However, these guided approaches are relatively rare in literature and most existing approaches are model-based, meaning that they require to learn an appropriate model of the environment first. For this reason, we will propose a novel model-free guided RL approach, called guided reinforcement learning (GRL). The guidance is mainly based on mixing samples containing full or partial state information while we gradually decrease the amount of full state information during training which ends up with a policy compatible with partial observations. The general formulation of our simple GRL approach allows to combine this approach with a variety of existing model-free RL algorithms as well as a variety of settings where these algorithms can be applied to. We demonstrate that our GRL approach can outperform other baseline algorithms that are trained directly on partial observations on nine different tasks. These tasks include two instances of the well-known discrete action space problem RockSample and the continuous action space problem LunarLander-POMDP which is a partially observable modification of the LunarLanderContinuous-v2 environment. Further, the benchmark tasks also include six partially observable tasks that we have constructed based on continuous control problems, simulated in the MuJoCo physics simulator.

Zusammenfassung

Aufgrund neuster Durchbrüche im Bereich der künstlichen Intelligenz, konnte auch RL für deutliche Aufmerksamkeit sorgen. Vor allem die Kombination von bisherigen Verfahren aus dem RL mit neutralen Netzen, welche bereits in anderen Forschungsfeldern für große Erfolge maßgeblich war, trug dazu bei. In vielen realen Problemstellungen ist es nicht immer möglich, die korrekten und vollständigen Informationen der Umgebung zu erhalten. Diese Szenarien sind auch bekannt unter partieller Beobachtbarkeit oder Unsicherheit. Die mathematische Formulierung solcher Probleme als POMDP erlaubt uns, Entscheidungsprobleme unter Unsicherheiten zu lösen. Des Weiteren gibt es Ansätze, auch guided RL genannt, die RL Algorithmen während des Lernprozesses mit zusätzlichen Informationen unterstützen, um die Performance für die Lösung von POMDPs zu verbessern. Allerdings sind diese Ansätze aktuell nur wenig erforscht und die meisten Ansätze sind Modell-basiert, was das vorherige Lernen einer exakten Darstellung der Umgebung erfordert. Aus diesem Grund veröffentlichen wir in dieser Arbeit einen neuartigen, Modell-freien guided (dt: geführt) RL Ansatz mit dem Namen GRL. Unser Ansatz basiert hauptsächlich auf dem Vermischen von Beispielen, welche sowohl vollständige Beobachtungen als auch nur Teile davon beinhalten. Während der Trainingsphase verringern wir nach und nach die Anzahl an Beispielen vollständiger Beobachtbarkeit, was uns zu Regeln führt, die auch mit partieller Beobachtbarkeit kompatibel sind. Dank der allgemeinen Formulierung unseres Ansatzes ist es möglich, diesen mit einer Vielzahl existierender Modell-freier RL Algorithmen zu kombinieren. Wir zeigen anhand weitreichender Experimente, dass unser Ansatz andere Ansätze, die direkt mit partiell beobachtbaren Beispielen trainiert wurden, in neun Problemstellungen übertreffen kann. Diese Problemstellungen umfassen zwei Instanzen des RockSample Problems und LunarLander-POMDP, eine partiell beobachtbare Modifikation des LunarLanderContinuous-v2 Problems. Weiter, haben wir eigene partiell beobachtbare Aufgaben erstellt, basierend auf neun bekannten Benchmarks im MuJoCo Simulator.

Contents

Figures and Tables

List of Figures

List of Tables

Abbreviations, Symbols and Operators

List of Abbreviations

Notation	Description
BPTT	backpropagation through time
CNN	convolutional neural network
COPOS	compatible policy search
CWPDIS	consistent weighted per-decision importance sampling
DDP	differential dynamic programming
DDPG	deep deterministic policy gradient
DP	dynamic programming
DPG	deterministic policy gradient
DQN	Deep Q-Network
DRQN	Deep Recurrent Q-Network
GPI	generalized policy iteration
GRL	guided reinforcement learning
i.i.d.	independently and identically distributed
IO-HMM	input-output hidden Markov model
KL	Kullback-Leibler
LSTM	Long Short-Term Memory
MC	Monte Carlo
MDP	Markov decision process
MPC	model predictive control
PBVI	point based value iteration
PG	policy gradient

PGAC	policy gradient actor-critic
POMDP	partially observable Markov decision process
PPO	Proximal Policy Optimization
ReLU	Rectified Linear Units
RL	reinforcement learning
RNN	recurrent neural network
SAC	Soft Actor-Critic
s.t.	subject to
SVG(0)	stochastic value gradients
TD	Temporal-Difference
TRPO	Trust Region Policy Optimization

List of Symbols

Notation	**Description**
γ	discount factor parameter
$\boldsymbol{\phi}(\mathbf{s})$	vector of features representing state $\mathbf{s}$
$\mathbf{F}$	Fisher-information matrix
$\boldsymbol{\theta}$	parameter vector
π	policy, decision making rule
$\pi_{\boldsymbol{\theta}}$	parametric policy corresponding to parameter vector $\boldsymbol{\theta}$
$\mathcal{A}$	set of all possible agent actions
$\mathcal{O}$	set of all possible observations
$\mathcal{S}$	set of all possible environment states
$V^{\pi}(\mathbf{s})$	value of state $\mathbf{s}$ under policy π (expected return)
V_t	estimate of V^{π}
$Q^{\pi}(\mathbf{s},\mathbf{a})$	value of taking action $\mathbf{a}$ in state $\mathbf{s}$ under policy π
$\mathbf{a}$	action taken by the agent
$\mathbf{o}$	observation
$\mathbf{s}$	environment's state

List of Operators

Notation	Description	Operator
argmax	Argument at which the function values are maximized	$\mathrm{argmax}(\bullet)$
E	Expected value	$E\{\bullet\}$
log	Natural logarithm	$\log(\bullet)$
T	Transpose of a matrix or vector	$(\bullet)^T$

1 Introduction

1.1 Motivation

Due to recent breakthroughs in artificial intelligence, more and more new techniques find their way into our everyday life. In 2014, Facebook introduced DeepFace, a deep neural network for face recognition that is able to identify faces with an accuracy of 97.35% and is therefore at the brink of human level accuracy [3]. Two years later, Google's AlphaGo becomes the first program to defeat a world champion in the game Go by using tree search in to evaluate positions and using deep neural networks for selecting moves [4]. Last year, Google presented a new language representation model called BERT that redefines the state of the art for 11 natural language processing tasks and is the first deeply bidirectional, unsupervised language representation trained only using a plain text corpus [5]. All these examples use deep learning whose rise has had a significant impact on many areas in machine learning, dramatically improving the state-of-the-art in tasks such as object detection, speech recognition, and language translation [6].

Reinforcement learning (RL) is a special field of artificial intelligence where an agent, e.g. a robot, tries to learn behavior through interaction with an environment. The robot does that by maximizing a reward signal that he receives after each interaction. Combining RL with deep neural networks, which could already achieve great success in other research fields, leads to deep RL. By making use of the powerful function approximation properties of neural networks, deep RL algorithms can approximate optimal policies and/or value functions. Further, deep RL made it possible to solve problems incorporating high-dimensional state and action spaces that were intractable for traditional RL algorithms. Deep RL achieved high attraction in the last recent years due to breakthrough as e.g. by the already mentioned AlphaGo algorithm.

In many real-world scenarios it is not always possible to perceive the true and complete state of the environment. An examples scenario would be driving an autonomous car in bad weather conditions where not all sensor data is available or cameras could only record in poor quality. Other examples are physical simulation models for robots which may capture only some of the physical phenomena. These scenarios are known as learning under uncertainty or under partial observability. Because RL algorithms usually need a correct perception of the whole environment, these algorithms would not perform well on partially observable tasks. Formulating such problems as partially observable Markov decision process (POMDP) allows us to solve decision making processes under uncertainty. Most POMDPs require taking long-term-dependencies into account to be able to learn advanced behavior, meaning to consider previous executed actions as well as received rewards and observations from a long time ago. For example static images of moving elements do not provide information about the velocity of the latter. By keeping track of consecutive image frames, an agent is capable to learn velocities. So-called memory-based approaches use finite memory representations to learn based on previous interactions with the environment. One possible approach for the memory representation is the use of recurrent neural networks (RNNs) which has been well studied in the past, e.g. [7], [8], [9] or [10]. Nevertheless, it is still challenging to learn optimal behavior due to the lack of information. Guided RL approaches address this problem by supporting the RL algorithms with additional state information during the learning process to increase their performance for solving POMDPs. However, these guided approaches are relatively rare in literature and there exits only a few prominent examples like the work from Zhang et al. [11] or from Levine et al. [12] on guided policy search. Most existing guided RL approaches are model based which requires to learn an appropriate model of the environment first. For this reason, we will introduce a model-free guided RL approach where learning of an model is not necessary.

1.2 Contribution

This work focuses on solving RL problems in partially observable domains. We propose a novel guided RL approach, called guided reinforcement learning (GRL), which guides RL algorithms with additional full state information during the learning process to increase their performance solving POMDPs in the test phase. The guidance is mainly based on mixing samples containing full or partial state information while we gradually decrease the amount of full state information during training which ends up with a policy compatible with partial observations.The general formulation of our simple GRL approach allows to combine this approach with a variety of existing model-free RL algorithms as well as a variety of settings where these algorithms can be applied to. We demonstrate the example usage with the compatible policy search (COPOS) algorithm and the Soft Actor-Critic (SAC) algorithm. As learning a model could be very difficult on complex systems, e.g. in the context of robot learning, we focus on model-free approaches that enable us optimizing policies directly through system interactions. With sufficient exploration we should be able to converge to a good policy. Further, we are able to efficiently learn behavior for parts of the problem that are actually fully observable while making learning easier for the algorithms due to using full observability at the beginning of learning.

We evaluated the performance of our GRL approach with other well-known RL algorithms on several challenging POMDP benchmark tasks. These tasks include RockSample [13], a discrete action space problem, and LunarLander-POMDP, a partially observable version of the continuous action space problem LunarLanderContinuous-v2 [14]. Further, we designed and developed partially observable versions of six continuous control problems, simulated in the MuJoCo [15] physics simulator. Also based on the LunarLanderContinuous-v2 environment we built another modification, called Noisy-LunarLander, to address the general case of bad observation quality and to investigate if our GRL approach could help to learn a policy that is more robust to uncertainty in observations. In this task, all observations returned by the environment are noisy. We compare our approach against training directly on noisy state information.

1.3 Overview

We structure this work as follows: First, chapter 2 provides a brief introduction in RL and several basic concepts like policies, value functions and function approximation as well as a definition for POMDPs. In addition, we will also show up possible solution techniques for POMDPs, give a brief insight into deep RL, and introduce model-free deep RL algorithms that are relevant for this work. In chapter 3, we present our GRL approach, refer to some related approaches which have been published recently, and demonstrate the example usage of GRL in combination with the algorithms COPOS and SAC. Chapter 4 covers the environments and tasks for continuous and discrete action spaces that we used to design experiments for the evaluation of our method. Further, we will present at this point the results of our comprehensive experiments. In chapter 5 we summarize this work, draw a brief conclusion and discuss future work.

2 Background

This chapter covers a brief overview and introduction to RL in general as well as to the specific case of partial observability. First, we will describe the typical RL problem as agent-environment interaction and further model this problem in a mathematical framework for the fully observable case. For real-world environments where the agent often has no perfect and complete perception of the real state of the environment, we adapt the mathematical framework to partial observability as POMDP. Beside several basic concepts needed for RL like policies, value functions and function approximation, we discuss some fundamental RL algorithms that can be divided into value-based methods, policy-based methods, and a combination of both called actor-critic methods. In addition, we will also show up possible solution techniques for POMDPs, that could be categorized into model-based methods and model-free methods. We concentrate mainly on model-free methods while having a look on related POMDP approaches from literature. As many current approaches for solving fully and partially observable RL problems have been scaled up to high-dimensional problems that make use of the powerful function approximation properties of neural networks, we provide a brief insight into deep RL. At the end of this chapter, several model-free deep RL algorithms that are relevant for this work will be introduced.

2.1 Reinforcement Learning

RL is the problem faced by a learner, called agent, that must learn behavior through trial-and-error interactions with a dynamic environment [16]. The agent tries to maximize a numerical reward signal, also known as reinforce, by discovering which action yields the most reward. Another important distinguishing feature of RL is delayed reward which means that in some cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards [1]. RL should be differentiated to other techniques used in current research areas like supervised learning which means learning from examples provided by a knowledgable external supervisor. However alone this kind of learning alone is not adequate for learning from interactions [1]. The most important difference to supervised learning is that in RL is no presentation of input/output pairs [16].

The general RL problem is meant to be a straightforward framing of the problem of learning from action-environment interaction to achieve a goal. A complete specification of an environment defines a task which could be described more specifically [1]:
The agent interacts with the environment at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, ...$ At each time step t, the agent receives some representation of the environment's state $\mathbf{s}_t \in \mathcal{S}$ and selects an action $\mathbf{a}_t \in \mathcal{A}$ based on the received state $\mathbf{s}_t$. $\mathcal{S}$ is the set of all possible states and $\mathcal{A}$ is the set of actions. In the next time step, after an action $\mathbf{a}_t$ was taken, the agent receives a numerical reward signal r_{t+1} and a new state $\mathbf{s}_{t+1}$. Based on that reward r_{t+1} and state $\mathbf{s}_{t+1}$ the agent has to act again and so on. The described agent-environment interaction is shown in figure 2.1.

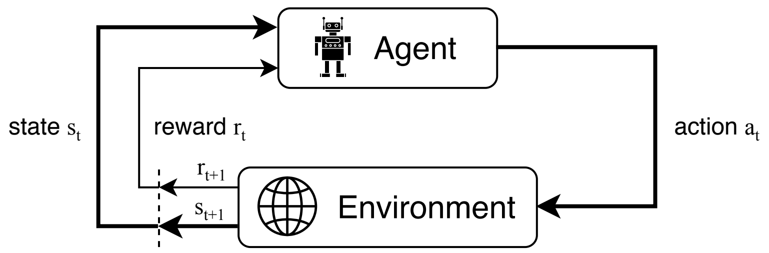

Figure 2.1.: The agent-environment interaction in a typical RL problem (from [1]). Each action $\mathbf{a}_t$ executed by the agent in state $\mathbf{s}_t$ at time step t is resulting in a new state $\mathbf{s}_{t+1}$ and new reward r_{t+1}, both returned by the environment.

As mentioned before, the agent's action affects not only its immediate reward, but might affect the next state(s) and reward(s). The agent must therefore be able to learn from delayed reward to possibly reach states with high reward after taking a sequence of actions. How problems with delayed reward can be modelled will be described in the next section.

2.1.1 Markov Decision Process (MDP)

A RL task that satisfies the *Markov* property, which means that a state signal succeeds in retaining all relevant information, is called a Markov decision process (MDP) [1]. A MDP is a tuple $<\mathcal{S}, \mathcal{A}, T, r>$ that consists of [2]:

- a set of states $\mathbf{s} \in \mathcal{S}$,
- a set of actions $\mathbf{a} \in \mathcal{A}$,
- a transition function defined as $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$, where the probability of ending up in the next state **s**' after doing action **a** in state **s** is denoted $T(\mathbf{s}, \mathbf{a}, \mathbf{s}')$,
- and r a reward function defined as $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$.

Specifically, the transition function must satisfy the *Markov* property $P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t-1}, \mathbf{a}_{t-1}, ...) = P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$, i.e. the result of an action does not depend on previous actions and visited states, but depends only on the currents state [2]. Different types of reward functions are commonly used, including rewards depending only on the current state $r = r(\mathbf{s})$, rewards depending on the current state and action $r = r(\mathbf{s}, \mathbf{a})$, and rewards including the transitions $r = r(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ [17]. We will use a reward function depending on the current state and action as denoted in the MDP definition above.

There exist several distinct types of systems that can be modelled by the MDP definition listed above. One of these types are episodic tasks where the goal is to take the agent from a starting state, given by the initial sate distribution $I : \mathcal{S} \rightarrow [0,1]$ which gives the probability of the system being started in that state, to a goal state in an episode of some length [2]. Further, it can be distinguished between finite, fixed horizon tasks in which each episode consists of a fixed number of time steps, indefinite horizon tasks in which each episode can end but episodes can have arbitrary length, and infinite horizon tasks, also called continuing tasks, where the system does not end at all [2]. We will consider episodic tasks with either finite or indefinite horizon in this work. In some tasks e.g. with infinite or indefinite horizon, the long-run reward is taken into account, which means that the agent tries to select action so that the sum of discounted rewards received over the future is maximized [1]. There γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate or discount factor.

2.1.2 Partially Observable Markov Decision Process (POMDP)

In many real-world environments, it is not possible for the agent to have perfect and complete perception of the state of the environment, which is necessary for learning methods based on MDPs [16]. When we consider scenarios where observations are incomplete or noisy e.g. in embedded system like drones which can only sense parts of the environment and do not have a complete view of the world, then for these tasks the Markov property for MDPs does not hold. This is because results of actions do not longer depend only on the current state signal. To face this limiting factor, the MDP framework could be extended into POMDPs which allows us to solve decision making processes under uncertainty i.e. with incomplete state information. Formally a POMDP can be described as a tuple $<\mathcal{S}, \mathcal{A}, \mathcal{O}, T, \mathcal{O}, r>$ consisting of (based on [18] and [2]):

- a set of states $\mathbf{s} \in \mathcal{S}$,
- a set of actions $\mathbf{a} \in \mathcal{A}$,
- a set of observations $\mathbf{o} \in \mathcal{O}$ the agent can experience of its world,
- a transition function defined as $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$, where the probability of ending up in the next state **s**' after doing action **a** in state **s** is denoted $T(\mathbf{s}, \mathbf{a}, \mathbf{s}')$,
- an observation function defined as $\mathcal{O} : \mathcal{S} \times \mathcal{A} \times \mathcal{O} \rightarrow [0,1]$, where $\mathcal{O}(\mathbf{s}', \mathbf{a}, \mathbf{o})$ is the probability of making observation **o** given that the agent took action **a** and landed in state **s**',
- and r a reward function defined as $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$.

What now distinguishes a POMDP from a fully observable MDP is that the agent perceives an observation $\mathbf{o} \in \mathcal{O}$, instead of observing the environment's state **s** directly [2]. Therefore, the definition of POMDPs contains two additional elements compared to MDPs: The set of observations $\mathcal{O}$ represents all possible sensor reading the agent can receive. The observation function $\mathcal{O}$ tells which particular observation the agent receives depending on the next state **s**' and maybe also on its action **a** [2]. The agent-environment interaction known from section 2.1 can be extended to fit the definiton of POMDPs which is illustrated in figure 2.2. How POMDP problems could be solved will be shown in section 2.1.7.

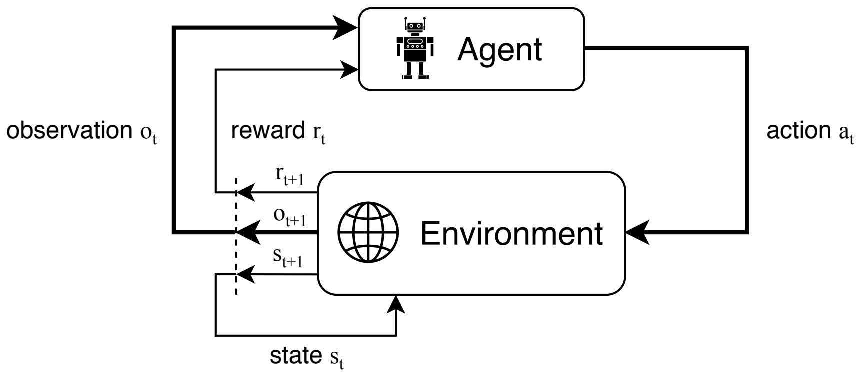

Figure 2.2.: Example for the agent-environment interaction in a POMDP (inspired by [1] and [2]). Each action $\mathbf{a}_t$ executed by the agent in state $\mathbf{s}_t$ at time step t is resulting in a new state $\mathbf{s}_{t+1}$, observation $\mathbf{o}_{t+1}$ and reward r_{t+1}. The agent receives only the observation and reward, while the environment's true state is hidden from the agent.

2.1.3 Policies and Value Functions

At each step, the agent implements a mapping, called policy π, from states $\mathbf{s}$ to probabilities of selecting each possible action $\mathbf{a}$ [1]. There are two different types of policies that are differentiated [2]: A deterministic policy $\pi(\mathbf{a})$ which is a function defined as $\pi : \mathcal{S} \to \mathcal{A}$ and a stochastic policy $\pi(\mathbf{s},\mathbf{a})$ which is a function defined as $\pi : \mathcal{S} \times \mathcal{A} \to [0,1]$ so that for each state $\mathbf{s} \in \mathcal{S}$, it holds $\pi(\mathbf{s},\mathbf{a}) \geq 0$ and $\sum_{\mathbf{a}\in\mathcal{A}} \pi(\mathbf{s},\mathbf{a}) = 1$. The deterministic policy $\pi(\mathbf{a})$ returns an action $\mathbf{a}$ for a given state $\mathbf{s}$ while the stochastic policy $\pi(\mathbf{s},\mathbf{a})$ returns a probability of taking an action $\mathbf{a}$ given state $\mathbf{s}$. We expect as well as [16] that the environment will be non-deterministic; that is, that taking the same action in the same state on two different observations may result in different next states and/or different rewards. For that reason we will use stochastic policies $\pi(\mathbf{s},\mathbf{a})$ in this work. Further, we will use parameterized policies $\pi_{\boldsymbol{\theta}}$ that are specified by a parameter vector $\boldsymbol{\theta}$. For an agent in discrete action spaces we will focus on softmax policies [19]

$$\pi_{\boldsymbol{\theta}}(\mathbf{s},\mathbf{a}) = \frac{\exp\big(f_{\boldsymbol{\theta}}(\mathbf{s},\mathbf{a})\big)}{\sum_{\mathbf{a}'\in\mathcal{A}} \exp\big(f_{\boldsymbol{\theta}}(\mathbf{s},\mathbf{a}')\big)}\,, \tag{2.1}$$

where softmax is a gerneralization of the logistic function of multiple variables and $f_{\boldsymbol{\theta}}$ is a neural network with trainable weighs represented by the parameter vector $\boldsymbol{\theta}$. In contrast, for an agent in continuous action spaces we focus on Gaussian policies

$$\pi_{\boldsymbol{\theta}}(\mathbf{s},\mathbf{a}) \sim \mathcal{N}\big(\mathbf{a}\big|\mu(\mathbf{s}),\boldsymbol{\sigma}^2\big)\,, \tag{2.2}$$

where the mean $\mu(\mathbf{s}) = f_{\mathbf{w}}(\mathbf{s})$ is represented by the output of a neural network with trainable weights $\mathbf{w}$. Moreover, the vector of trainable parameters $\boldsymbol{\sigma}^2$ specifies the variance of the Gaussian distribution. Both parameters together $\boldsymbol{\theta} = [\mathbf{w},\boldsymbol{\sigma}^2]$ build the parameter vector $\boldsymbol{\theta}$ of the policy.

Most of the existing RL algorithms for solving MDPs compute optimal policies based on estimating value functions. A value function is a function of states (or of state-action pairs) that estimates "how good" it is for the agent to be in a given state (or "how good" it is to perform a given action in a given state) [1]. Value functions are defined for particular policies and the notation of "how good" is expressed in terms of an optimality criterion, i.e. in terms of the expected return [2]. According to [1], the value of a state $\mathbf{s}$ under a policy π, denoted $V^\pi(\mathbf{s})$, is the expected return when starting in $\mathbf{s}$ and following π thereafter. Formally, $V^\pi(\mathbf{s})$ can be expressed as

$$V^\pi(\mathbf{s}) = E_\pi\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| \mathbf{s}_t = \mathbf{s} \right\}, \tag{2.3}$$

where $E_\pi\{\ \}$ denotes the expected value given that the agent follows policy π. Similarly, the value of taking action $\mathbf{a}$ in state $\mathbf{s}$ under a policy π, denoted $Q^\pi(\mathbf{s},\mathbf{a})$, as the expected return starting from $\mathbf{s}$, taking action $\mathbf{a}$, and thereafter following policy π:

$$Q^\pi(\mathbf{s},\mathbf{a}) = E_\pi\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a} \right\}. \tag{2.4}$$

Note that this definition of the state-value function $V^{\pi}(\mathbf{s})$ and the state-action-value function $Q^{\pi}(\mathbf{s},\mathbf{a})$ is for an infinite horizon.

A fundamental property of value functions is that they satisfy particular recursive relationships. For any policy π and any state $\mathbf{s}$ the following consistency condition, the so-called Bellman Equation, holds between the value of $\mathbf{s}$ and the value of its possible successor states:

$$V^{\pi}(\mathbf{s}) = E_{\pi}\left\{ r_{t+1} + \gamma V^{\pi}(\mathbf{s}_{t+1}) \middle| \mathbf{s}_t = \mathbf{s} \right\} \tag{2.5}$$

2.1.4 Goals of Reinforcement Learning

The aim of RL is to discover an optimal policy π^* that maps states $\mathbf{s}$ (or observations $\mathbf{o}$) to actions $\mathbf{a}$ so as to maximize the expected return J, which corresponds to the cumulative expected reward [17]. Informally, the agent tries to maximize the total amount of reward it receives which means maximizing not immediate reward, but cumulative reward in the long run [1]. There exist different models of optimal behavior [16] which result in different definitions of the expected return corresponding to the types of tasks that can be modelled with the MDP framework as defined in section 2.1.1. A finite-horizon model only attempts to maximize the expected reward for the horizon H, i.e. the next H time steps k, denoted by [17] as

$$J = E\left\{ \sum_{k=0}^{H} r_k \right\} . \tag{2.6}$$

In many cases the agent-environment interaction does not break naturally into identifiable episodes with horizon H, but goes on continually without limit [1], defined as infinite-horizon model where the long-run reward is taken into account, but the rewards that are received in the future are discounted to how far away in time they will be received [2]. In this discounted case, rewards obtained later are discounted more than rewards obtained earlier. Following [17], the expected discounted return can be defined as

$$J = E\left\{ \sum_{k=0}^{\infty} \gamma^k r_k \right\} , \tag{2.7}$$

where γ is the discount factor as already introduced in section 2.1.1. The discount factor trades off immediate and future rewards, and ensures that - even with infinite horizon $H = \infty$ - the sum of rewards obtained is finite [2].

2.1.5 Generalization and Function Approximation

We have so far assumed that our estimates of value functions are represented as a table with one entry for each state (or for each state-action pair) which is limited to tasks with small numbers of states and actions [1]. The problem for tasks with big state spaces is that a huge amount of date is needed to fill the tables accurately which means that the agent must visit all states many times in order to produce enough data. Especially for continuous tasks, the state and action spaces become too large to visit and to store all states. To face this problem, we need to generalize from previously experienced states and actions to those that have never been experienced before. The kind of generalization we require is often called function approximation as it takes examples from a desired function (e.g. a value function) and attempts to generalize from those to construct an approximation of the entire function [1]. Function approximation is an instance of supervised learning which has been studied extensively in related topics and can be applied to RL. One way of determining the values of the approximate function is using a set of tunable parameters as it is done in linear and non-linear function approximation.

Linear Function Approximation

We assume some given feature-extraction function $\phi : \mathcal{S} \rightarrow \Phi$that maps states into features in the features space Φ where $\Phi \subseteq \mathbb{R}^{D_\Phi}$ and D_Φ is the dimension of the feature space [2]. There exist several ways for constructing feature vectors and choosing good features which we will not cover here. The approximate state value-function is given by

$$V_t(\mathbf{s}) = \boldsymbol{\theta}_t^T \boldsymbol{\phi}(\mathbf{s}) , \tag{2.8}$$

where $\boldsymbol{\theta}_t \in \Theta$ denotes the adaptable parameter vector at time t and $\boldsymbol{\phi}(\mathbf{s}) \in \Phi$ is the feature vector of state $\mathbf{s}$ [2]. In this case, the approximate value function is said to be a linear function approximator. Note that for this definition the dimension D_Θ of the parameter space is equal to the dimension D_Φ of the feature space. Linear basis function approximators form one of the most widely used approximate value-function techniques in continuous and discrete state spaces due to the simplicity of their representation. Another reason for the popularity is because of the convergence theory for the approximation of value function based on samples [17].

The main drawback of linear function approximation is the need for good informative features while features are often assumed to be hand-picked beforehand which may require domain knowledge. Even if convergence in the limit on an optimal solution is guaranteed, this mentioned solution is only optimal in the sense that it is the best possible linear function of the given features [2].

Non-linear Function Approximation

If useful features are unknown beforehand for the given problem, it can be beneficial to use non-linear function approximation. Empirical results have been obtained by combining RL algorithms with non-linear function approximators, such as neural networks [2]. In a parametric non-linear function approximator, the function that should be optimized is represented by some predetermined parametric function e.g. for value-based algorithms we may have (by [2])

$$V_t(\mathbf{s}) = V(\boldsymbol{\phi}(\mathbf{s}), \boldsymbol{\theta}_t)\,, \tag{2.9}$$

where the size of $\boldsymbol{\theta}_t \in \Theta$ is not necessarily equal to the size of $\boldsymbol{\phi}(\mathbf{s}) \in \Phi$. For instance, V may be a neural network where $\boldsymbol{\theta}_t$ is a vector with all its trainable weights at time t. In general, a non-linear function approximator may approximate an unknown function with better accuracy than a linear function approximator that uses the same features as input [2].

2.1.6 Fundamental Reinforcement Learning Algorithms

In this section, we will discuss some fundamental algorithms for RL that are relevant for this work. First of all, the most important distinction of existing algorithms is the one between model-based and model-free algorithms. The basic assumption in model-based algorithms is that a model of the MDP is known beforehand. Thus, it can be used to compute value functions and policies while most methods are aimed at computing state value functions which in the presence of the model are used for optimal action selection [2]. The most prominent set of model-based algorithms is also known under the term dynamic programming (DP). In contrast, model-free approaches learn a policy or value function directly from interactions with the environment and not rely on a perfect model of the environment. This interaction of the agent with the environment generates samples $\tau = (\mathbf{s}_0, \mathbf{a}_0, r_0, \mathbf{s}_1, \mathbf{a}_1, r_1, ...)$, also called rollouts, trajectories or paths which contain the taken actions, observed states and received rewards of one episode. Model-free approaches can be further divided into value-based and policy-based methods. Value-based methods are trying to estimate a value function directly and implicitly find an optimal policy while policy-based methods are trying to optimize the policy directly. Actor-critic methods are a combination of both that learn an optimal value function as well as an optimal policy.

Value-based Methods

In value-based methods, samples τ are used to update a state-value function $V^\pi(\mathbf{s})$ or state-action-value (also called action-value) function $Q^\pi(\mathbf{s}, \mathbf{a})$ that gives an approximation of the current optimal policy π^*. A very important underlaying mechanism is the so-called generalized policy iteration (GPI) principle that refers to the general idea of two interacting processes (policy evaluation and policy improvement), independent of the granularity and other details of the two processes [1]. The policy evaluation step estimates the utility of the current policy π by computing $V^\pi(\mathbf{s})$ to gather information about the policy for computing the second step, the policy improvement step [2] which makes the policy greedy with respect to the current value function (policy improvement) [1]. A policy that is greedy with respect to the optimal value function V^* is an optimal policy [1]. The GPI principle is initially from DP which assumes that the model is fully known, but there exist also model-free methods like Monte Carlo (MC) or Temporal-Difference (TD) learning methods that use some variation of this principle.

TD learning algorithms learn estimates of values based on other estimates [2] and are a combination of MC ideas and DP ideas. Like MC methods, TD methods can learn directly from raw experience without a model of the environment's dynamics [1]. Given some experience following a policy π, both methods update their estimate V of V^π. Whereas MC methods must wait until the end of the episode to determine the increment to V, TD methods need to wait only until the next time step. The simplest TD method, known as TD(0) algorithm [20], is

$$V(\mathbf{s}_t) \leftarrow V(\mathbf{s}_t) + \alpha\big[r_{t+1} + \gamma V(\mathbf{s}_{t+1}) - V(\mathbf{s}_t)\big]\,, \tag{2.10}$$

where $\alpha \in [0,1]$ is the learning rate that determines by which amount the values get updated. The target for the TD update is $r_{t+1} + \gamma V(\mathbf{s}_{t+1})$. As TD methods base their update partially on an existing estimate, they are said to be a bootstrapping method [1].

Since the value function update in equation 2.10 is only based on the approximation of a state value function, using an action-value function $Q(\mathbf{s},\mathbf{a})$ could lead to a better action selection. One of the most popular methods to estimate Q-value functions in a model-free style is an off-policy TD control algorithm called Q-learning [21]. Its simplest form, one-step Q-learning, is defined by

$$Q(\mathbf{s}_t,\mathbf{a}_t) \leftarrow Q(\mathbf{s}_t,\mathbf{a}_t) + \alpha\left[r_{t+1} + \gamma \max_{\mathbf{a}} Q(\mathbf{s}_{t+1},\mathbf{a}) - Q(\mathbf{s}_t,\mathbf{a}_t)\right]. \tag{2.11}$$

In this case, the learned action-value function Q directly approximates Q^*, the optimal action-value function, independent of the followed policy which dramatically simplifies the analysis of the algorithm and enables early convergence proofs [1]. Q-learning is exploration-intensive, meaning that it will converge to the optimal policy regardless of the exploration policy being followed, under the assumption that each state-action pair is visited an infinite number of times, and that the learning rate α is decreased appropriately [2].

Unlike the off-policy Q-learning algorithm, there exists also an on-policy TD learning variant, called SARSA [22]. Off-policy algorithms can learn about the value of a different policy than the one that is being followed, whereas on-policy algorithms approximate the state-value function $V^{\pi}(\mathbf{s})$ or the action-value function $Q^{\pi}(\mathbf{s},\mathbf{a})$, which represent the value of the policy π that they are currently following [2]. The SARSA algorithm aims at estimating the optimal policy π^* by estimating $Q^{\pi}(\mathbf{s},\mathbf{a})$ for the current behavior policy π as well as for all states $\mathbf{s}$ and actions $\mathbf{a}$. After every transition from a non-terminal state $\mathbf{s}_t$, the following update rule is executed:

$$Q(\mathbf{s}_t,\mathbf{a}_t) \leftarrow Q(\mathbf{s}_t,\mathbf{a}_t) + \alpha\left[r_{t+1} + \gamma Q(\mathbf{s}_{t+1},\mathbf{a}_{t+1}) - Q(\mathbf{s}_t,\mathbf{a}_t)\right], \tag{2.12}$$

where the action $\mathbf{a}_{t+1}$ is the action that is being taken by the current policy for state $\mathbf{s}_{t+1}$ and the max-operator in Q-learning is replaced by the estimate of the value of the next action according to the policy [2]. This update rule uses every element of the quintuple of events $(\mathbf{s}_t,\mathbf{a}_t,r_{t+1},\mathbf{s}_{t+1},\mathbf{a}_{t+1})$ that make up a transition from one state-action pair to the next and is additionally the reason for the name SARSA [1].

Policy-based Methods

While value-based algorithms parametrize the policy indirectly by estimating state or action values from which a policy can be inferred, policy-based (also called direct policy-search or actor-only) algorithms store a policy directly and try to update this policy to approximate the optimal policy π^* [2]. Model-free and policy-based methods try to update the parameters $\boldsymbol{\theta}$ such that trajectories τ with higher rewards become more likely. When following the new policy, the average return

$$J_{\boldsymbol{\theta}} = E\left\{r(\tau)\middle|\boldsymbol{\theta}\right\} = \int \pi_{\boldsymbol{\theta}}(\tau) r(\tau)\, d\tau \tag{2.13}$$

increases [23]. The total accumulated reward $r(\tau)$ for one sampled trajectory τ is defined as $r(\tau) = \sum_{t=0}^{T} r(\mathbf{s}_t,\mathbf{a}_t)$ and $\pi_{\boldsymbol{\theta}}$ is usually a parameterized stochastic policy e.g. Gaussian policy (see section 2.1.3). There exist several strategies for updating the policy, e.g. policy gradient (PG) methods or expectation-maximization-based methods. We will concentrate on PG methods which use gradient ascent for maximizing the objective in equation 2.13. In gradient ascent, the parameter update direction is given by the gradient $\nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}}$ as it points in the direction of steepest ascent of the expected return. The policy gradient update is given by

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}}\ , \tag{2.14}$$

where α is the learning rate [23]. The policy gradient can be estimated using the likelihood ratio trick $\nabla_x f(x) = f(x)\nabla_x \log f(x)$, sometimes called REINFORCE [24] trick, by rewriting the gradient by (following [25])

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}} &= \int \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\tau) r(\tau)\, d\tau \\ &= \int \pi_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau) r(\tau)\, d\tau \\ &= E\left\{\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau) r(\tau)\right\}. \end{aligned} \tag{2.15}$$

Exploiting the structure of the trajectory distribution $\pi_\theta(\tau)$ by decomposing $\nabla_\theta \log \pi_\theta(\tau)$ into the single time steps and substracting a baseline b leads us to

$$\nabla_\theta J_\theta = E\left\{\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\big(r(\tau)-b\big)\right\}, \tag{2.16}$$

which is also known as the REINFORCE policy gradient [23]. The baseline $b \in \mathbb{R}$ can be chosen arbitrarily [24] but usually with the goal to minimize the variance of the gradient estimator [25].The REINFORCE algorithm also uses MC sampling to estimate the gradient from sampled trajectories τ:

$$\nabla_\theta J_\theta \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(\mathbf{a}_t^{(i)}|\mathbf{s}_t^{(i)})\big(r^{(i)}(\tau)-b\big). \tag{2.17}$$

Actor-Critic Methods

Actor-critic methods are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function. Thereby, the policy structure is known as the actor, because it is used to select actions. The estimated value function is known as the critic, because it judges the actions made by the actor [1]. After action selection, the critic evaluates the action using the TD-error

$$\delta_t = r_{t+1} + \gamma V(\mathbf{s}_{t+1}) - V(\mathbf{s}_t) \tag{2.18}$$

and drives all learning in both actor and critic, as shown in figure 2.3. The purpose of this error is to strengthen or weaken the selection of this action in this state [2].

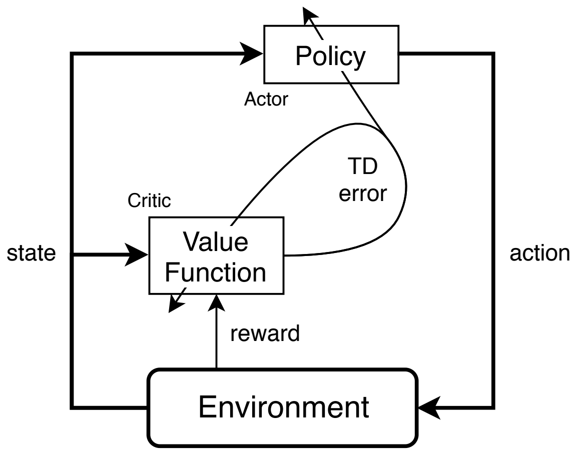

Figure 2.3.: The actor-critic architecture (from [1]) where the TD-error of the value function (critic) is used to evaluate the actions made by the policy (actor).

It can be shown that if $\mathbf{a}_t$ is selected according to π, under some assumptions the TD-error δ_t is an unbiased estimator of $Q^\pi(\mathbf{s}_t,\mathbf{a}_t) - V^\pi(\mathbf{s}_t)$ which is also called an advantage function $A^\pi(\mathbf{s}_t,\mathbf{a}_t)$ [2]. Instead of using the returns $r(\tau)$ for the gradient of the policy in equation 2.1.6, we can also use the expected reward at time step t, i.e. $Q^\pi(\mathbf{s}_t,\mathbf{a}_t)$, to evaluate an action $\mathbf{a}_t$ as it is used by the Policy Gradient Theorem algorithm [26]. The Policy Gradient Theorem states that

$$\begin{aligned}\nabla_\theta J_\theta &= E\left\{\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\left(\sum_{j=t}^{T} r(\mathbf{s}_j,\mathbf{a}_j)-b\right)\right\}\\ &= E\left\{\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\big(Q^\pi(\mathbf{s}_t,\mathbf{a}_t)-b\big)\right\},\end{aligned} \tag{2.19}$$

where the rewards are not correlated with future actions [23]. Assuming $b = V^\pi(\mathbf{s}_t)$ for the baseline leads to the above mentioned advantage function and therefore to an unbiased estimate for the gradient of the policy:

$$\nabla_\theta J_\theta = E\left\{\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) A^\pi(\mathbf{s}_t,\mathbf{a}_t)\right\} \tag{2.20}$$

This procedure shows how a policy-based method can be turned into an actor-critic method by using a state-value function (critic) as baseline. The update rule of the policy parameters remains the same as in equation 2.14.

2.1.7 Solution Techniques for POMDPs

In section 2.1.2 we introduced the general POMDP framework and now we will give a brief overview of techniques for solving POMDPs. In general, existing techniques could be divided into model-based methods and model-free methods. A model-based approach for solving POMDPs is the use of an internal belief state that summarizes previous experience. We will show the reformulation of POMDPs into a belief MDP in the next paragraph. Model-free methods do not need to reformulate POMDP problems or to learn a model. Instead, model-free methods learn an optimal policy directly from agent-environment interactions. We will mainly focus on model-free methods in this thesis. A more detailed insight into related approaches for solving POMDPs in literature will be given in section 2.2.

Model-based Methods

The agent's observations in a POMDP do not uniquely identify the state of the environment. This is why it is difficult to find an optimal policy. Maintaining an internal belief state and choosing actions based ons this state could improve the performance. The belief state b can be represented as a probability distribution over states of the world $\mathcal{S}$. This distribution encodes the agent's subjective probability about the state of the world an provide a basis for action under uncertainty [18]. Computing the update from an existing belief state b in addition with an action $\mathbf{a}$ and an observation $\mathbf{o}$ to a new belief state b' can be described as follows:

Let $b(\mathbf{s})$ denote the probability assigned to the world state $\mathbf{s}$ by belief state b so that $0 \leq b(\mathbf{s}) \leq 1$ for all $\mathbf{s} \in \mathcal{S}$ and that $\sum_{\mathbf{s}\in\mathcal{S}} b(\mathbf{s}) = 1$. Then, the new degree of belief in some state $\mathbf{s}'$, $b'(\mathbf{s}')$, can be obtained from basic probability theory:

$$\begin{aligned} b'(\mathbf{s}') &= P(\mathbf{s}'|\mathbf{o},\mathbf{a},b) \\ &= \frac{P(\mathbf{o}|\mathbf{s}',\mathbf{a},b)P(\mathbf{s}'|\mathbf{a},b)}{P(\mathbf{o}|\mathbf{a},b)} \\ &= \frac{P(\mathbf{o}|\mathbf{s}',\mathbf{a})\sum_{\mathbf{s}\in\mathcal{S}} P(\mathbf{s}'|\mathbf{a},b,\mathbf{s})P(\mathbf{s}|\mathbf{a},b)}{P(\mathbf{o}|\mathbf{a},b)} \\ &= \frac{\mathcal{O}(\mathbf{s}',\mathbf{a},\mathbf{o})\sum_{\mathbf{s}\in\mathcal{S}} T(\mathbf{s},\mathbf{a},\mathbf{s}')b(\mathbf{s})}{P(\mathbf{o}|\mathbf{a},b)} \ . \end{aligned} \tag{2.21}$$

The denominator $P(\mathbf{o}|\mathbf{a},b)$ is independent of $\mathbf{s}'$ and can be treated as a normalizing factor that causes b' to sum up to 1 [18]. The observation function $\mathcal{O}$ and the transition function T are from the original POMDP definition (see section 2.1.2). A continuous space belief MDP is an extension of a POMDP and contains the following elements (according to [18]):

- a set of belief states $b \in \mathcal{B}$ that covers the state space $\mathcal{S}$,
- a set of actions $\mathbf{a} \in \mathcal{A}$ which remains the same as in POMDPs,
- a state-transition function $\mathcal{T}(b,\mathbf{a},b')$, which is defined as

$$\mathcal{T}(b,\mathbf{a},b') = P(b'|\mathbf{a},b) = \sum_{\mathbf{o}\in O} P(b',|\mathbf{a},b,\mathbf{o})P(\mathbf{o}|\mathbf{a},b) \ , \tag{2.22}$$

  where

$$P(b',|\mathbf{a},b,\mathbf{o}) = \begin{cases} 1 & \text{if } b = b' \\ 0 & \text{otherwise,} \end{cases} \tag{2.23}$$

- and a reward function $r_b(b,\mathbf{a})$ on belief states which is constructed form the original reward function:

$$r_b(b,\mathbf{a}) = \sum_{\mathbf{s}\in\mathcal{S}} b(\mathbf{s})r(\mathbf{s},\mathbf{a}) \ . \tag{2.24}$$

In these belief MDPs an optimal policy π^* maps beliefs to actions and provides therefore optimal behavior for the original POMDP. A policy π can be characterized by a value function V^π (see section 2.1.3) which is defined for belief states b as:

$$V^\pi(b) = E_\pi\left\{ \sum_{k=0}^{\infty} \gamma^k r_b(b_{t+k+1},\mathbf{a}_{t+k+1}) \middle| b_t = b \right\} \ . \tag{2.25}$$

The optimal value function V^* satisfies the Bellman optimality equation

$$V^* = H_{POMDP} V^* \,, \tag{2.26}$$

where H_{POMDP} is the Bellman backup operator for POMDPs, defined as

$$V^*(b) = \max_{\mathbf{a}\in\mathcal{A}} \left[r_b(b, \mathbf{a}) + \gamma \sum_{\mathbf{o}\in O} \mathcal{T}(\mathbf{o}, \mathbf{a}, b) V^*(b') \right] , \tag{2.27}$$

with b' given by equation 2.21 [2]. Computing value functions over a continuous belief space are difficult to solve, but the value function has a particular structure that can be exploited [27]. It can be shown that V^* is piecewise linear a convex, and can thus be written as

$$V^*(b) = \max_{\alpha\in\mathcal{V}} b \cdot \alpha \,, \tag{2.28}$$

for some set of vectors $\mathcal{V} = [\alpha_1, ..., \alpha_n]$ [28]. This set of vectors is defined for each action $\mathbf{a}$: $\alpha^{\mathbf{a}}(\mathbf{s}) = r(\mathbf{s}, \mathbf{a})$ and $(\cdot)$ denotes the inner product [2].

By representing a POMDP as belief MDP, many model-based methods could be applied e.g the value iteration algorithm or point-based methods.

Model-free Methods

Besides model-based methods, there exists also the possibility to apply model-free algorithms on POMDPs. The most native strategy for dealing with partial observability is to ignore it [16]. Therefore the observations received by the agent are treated as if they were states of the environment and MDP methods are applied to learn the behavior. The problem of this approach is, that it is NP-hard [29] to find the optimal mapping from observations to actions, and even the best mapping can have very poor performance. This means, finding a locally-optimal policy in this setting is possible, but finding a globally-optimal policy is hard i.e. NP-hard. The described approach can also be categorized as memory-less method.

The only way to behave truly effective in a wide-range of environments is to use memory, also called internal state, of previous actions and observations to disambiguate the current state [16]. There exists a variety of memory-based approaches for applying this principle. Storing the complete history of the process is not a practical option for several reasons. Firstly, as in the model-free case, the agent is not able to compute a belief state as this representation grows without bounds. Secondly, such a representation does not allow an easy generalization [2]. In the following, we will provide a brief overview of different representations of the internal state.
One of the possible approaches is to use a RNN to learn value functions or policies. The network can be trained using backpropagation through time (BPTT) (or some other suitable technique) and thus learns to retain history features to predict the value function or policy [16].
Another common approach is to maintain a history window containing the last H observations (and actions), where the horizon H is typically an a-priori defined parameter [2]. This finite-memory approach restores the Markov property by allowing decisions based on the history [16]. A finite history with length H at time step t is usually represented as

$$\mathbf{h}_t = \left(\mathbf{o}_t, \mathbf{a}_{t-1}, ..., \mathbf{o}_{t-H}, \mathbf{a}_{t-H-1}\right) . \tag{2.29}$$

Besides fixed history windows, several algorithms for variable history windows have been proposed, e.g. from McCallum [30]. These techniques allow the history window to have a different depth in different parts of the state space [2]. We will also follow a memory-based representation with a fixed-length history for our approach (see section 3.2). For more model-free approaches see section 2.2 where we cover some recent algorithms for POMDPs in literature.

2.1.8 Deep Reinforcement Learning

Most of the success in deep RL is based on scaling up prior work in RL to high-dimensional problems due to the learning of low-dimensional feature representations and the powerful function approximation properties of neural networks [31]. In general, deep RL is based on training deep neural networks to approximate an optimal policy and/or optimal value functions. In the following we will briefly describe relevant neural networks that are commonly used for deep RL algorithms (see section 2.3) and solving POMDPs (see section 2.2). Thereafter, we will mention two basic methods that will be extended in section 2.2, section 2.3 and section 3.1.

Deep feedforward networks, also called feedforward neural networks, or multilayer perceptrons (MLPs), are essential deep learning methods with the goal of approximating some function, for example a state-value function V or a Q-function. A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$ that maps an input $\mathbf{x}$ to an output $\mathbf{y}$ and learns the value of the parameters $\boldsymbol{\theta}$ that results in the best function approximation. The term feedforward comes form the information flow through the function being evaluated from $\mathbf{x}$, through the intermediate computations used to define f, and finally to the output $\mathbf{y}$ while there are no feedback connections. Feedforward networks are associated with the directed acyclic graph describing how the functions are composed together. The most commonly used structure for example, is assuming that we have three functions $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$ connected in a chain to form $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$. The final layer of a feedforward network, $f^{(3)}$ in this example, is called output layer and all previous layers, $f^{(1)}$ and $f^{(2)}$ in this example, are called hidden layers [32]. At each layer, a set of units, also called neurons, compute a weighted sum of their inputs from the previous layer and pass the result through a non-linear function [6].

RNNs are a family of neural networks for processing sequential data [32]. Only one element is processed at a time, while maintaining a "state vector" in the hidden units that implicitly contains information about the history of all the past elements of a sequence. When we consider the outputs of the hidden units at different discrete time steps as if they were the outputs of different neurons in a deep multilayer network, RNNs can be seen as a very deep feedforward network in which all the layers share the same weights. Although their main purpose is to learn long-term dependencies, it is difficult to learn how to store information for longer time. To resolve that problem, one idea is to augment the network with an explicit memory [6]. The first technique that puts this idea into practice is the Long Short-Term Memory (LSTM) [33] network which uses special hidden units with the neural behavior to remember inputs for a long time. A special unit called the memory cell acts like a gated leaky neuron: it has a connection to itself at the next time step that has the weight of one. Therefore it copies its own real-valued state and accumulates the external signal. However, this self-connection is multiplicatively gated by another unit that learns to decide when to clear the content of the memory. LSTMs have subsequently proved to be more effective than conventional RNNs e.g. on speech recognition tasks, especially when they have several layers for each time step [6]. Many methods for solving partially observable RL problems also use LSTMs to remember previous observations and actions, see section 2.2 for more details.

The last kind of neural networks that we will briefly describe here are convolutional neural networks (CNNs) which are often used in the context of RL to learn from raw inputs like pixels. CNNs are a specialized kind of neural networks for processing data that has a known grid-like topology. The name of these networks indicates that the network employs a mathematical operation called convolution which is a specialized kind of linear operations [32]. The architecture of a typical CNN is structured as a series of stages. The first few stages consist of two types of layers: convolutional layers and pooling layers. The units in a convolutional layer are organized in feature maps, where each unit is connected to local patches in the feature maps of the previous layer through a set of weights called a filter bank. The result of this local weighted sum is then passed through a non-linear function. The role of the convolutional layer is to detect local conjunctions of features from the previous layer, while the role of the pooling layer is to merge semantically similar features into one. A typical pooling unit computes the maximum of a local patch of units in one feature map [6]. In later stages, CNNs often use fully-connected layers.

A very prominent example for deep RL methods is the Deep Q-Network (DQN) [34] [35] which could achieve comparable performance as a professional human games tester across a set of 49 Atari 2600 games [36], using the same algorithm, network architecture and hyperparameters. The inputs to the DQN are four greyscale frames of the game, concatenated over time, which are initially processed by several convolutional layers in order to extract spatiotemporal features, such as the movement of a ball. The final feature map from the convolutional layers is processed by several fully connected layers, which more implicitly encode the effects of the actions. At the final fully connected layer, the network uses a Q-function to select and output a discrete action, which corresponds to one of the possible control inputs for the game (e.g. directions of the joystick or the fire button). In general, the strength of the DQN lies in its ability to compactly represent both high-dimensional observations and the Q-function using deep neural networks [31].

Another popular example is the deep deterministic policy gradient (DDPG) [37], an off-policy actor-critic method, that is a deep variant of deterministic policy gradient (DPG) [38] using a Q-function to enable off-policy learning and a deterministic actor that maximizes the Q-function. DPG extends the standard policy gradient theorems for stochastic policies (see section 2.1.6) to deterministic policies. An advantage of DPGs is, that it only integrates over the state space instead of integrating over state and action spaces as in stochastic policy gradient, which requires fewer samples in problems with large action spaces [31].

2.2 Related POMDP Approaches

We have already seen a general overview of methods that can be applied to solve POMDPs in section 2.1.7 as well as RL algorithms targeted on solving MDPs. In this section, we will have a look on some specific approaches for partially observable environments. Most of the following approaches are model-free algorithms because the focus of this work is also on model-free approaches. All these model-free approaches are memory-based approaches that use finite memory representation to learn based on previous interactions with the environment. Nevertheless, we will present one model-based approach at the end of this section to provide an insight into alternatives for model-free approaches.

In model-free approaches the application of RNNs, especially LSTM, has been studied extensively. This is the reason why most of the following approaches will use these techniques as memory for keeping track of previous observations (and actions). One of the first approaches that applies LSTMs to POMDPs in the context of RL in general is from Bakker [7]. A RNN was used to directly approximate the value function of a RL algorithm whereby the state of the environment is approximated by the current observations, which is the input for the network, together with the recurrent activations in the network, which represent the agent's history. The neural network approximates the value function of Advantage learning which is an improvement of Q-learning (see section 2.1.6). The LSTM network's output units directly code for the Advantage values of different actions. The TD error δ_t, derived from the equivalent of the Bellman equation for Advantage learning, is taken as the function approximatior's prediction error at time step t:

$$\delta_t = V(\mathbf{s}_t) + \frac{r_t + \gamma V(\mathbf{s}_{t+1}) - V(\mathbf{s}_t)}{\kappa} - A(\mathbf{s}_t, \mathbf{a}_t) \ , \tag{2.30}$$

where $V(\mathbf{s}_t) = \max_{\mathbf{a}} A(\mathbf{s}_t, \mathbf{a}_t)$ is the value of state $\mathbf{s}$ and κ is a constant scaling of the difference between values of optimal and suboptimal actions. Further, Advantage learning was extended with eligibility traces, which have often been found to improve learning in RL, especially in non-Markovian domains. This yields Advantage(λ) learning that requires the storage of one eligibility trace e_{im} per weight w_{im} of the neural network (of the connection from unit m to unit i). A weight update then corresponds to

$$w_{im,t+1} = w_{im,t} + \alpha \delta_t e_{im,t} \ , \tag{2.31}$$

where

$$e_{im,t} = \gamma \lambda e_{im,t-1} + \frac{\partial y_t^K}{\partial w_{im}} \ . \tag{2.32}$$

y^K is a unit activation where K indicates the output unit associated with the executed action and λ is a parameter determining how fast the eligibility trace decays. Bakker [7] employs a directed exploration technique because non-Markovian RL requires extra attention to the issue of exploration. In order to do this, a separate multilayer feedforward neural network, with the same input as the LSTM network (representing the current observation) an one output unit y^v, is trained concurrently with the LSTM network. This network predicts the absolute value of the current TD error δ plus its own discounted prediction at the next time step:

$$y_{d,t}^v = |\delta_t| + \beta y_{t+1}^v \ , \tag{2.33}$$

where $y_{d,t}^v$ is the desired value for output y_t^v, and $\beta \in [0,1]$ is a discount parameter. This is used to identify which observations are problematic, in the meaning that they are associated with large errors in the current value estimation, or precede situations with large errors.

Wierstra et al. [8] applied the policy gradient, as we have seen in policy-based approaches in section 2.1.6, on RNNs in combination with backpropagation i.e. BPTT. Specifically, they use eligibility-BPTT to update all parameters conjunctively, yielding solutions that better generalize over complex histories. In order to estimate the gradients for a history-based approach, they map histories to action probabilities by using LSTM's internal state representation. Only the output part of the neural network is interpreted stochastically which allows to only estimate the eligibilities of the output units at every time step during the backward pass. The gradient on the other parameters can be derived efficiently via eligibility-BPTT, treating output eligibilities like normal errors that would be treated in an RNN trained with gradient descent. For the output structure of the network they used a softmax layer for discrete tasks and a Gaussian output structure for continuous tasks.

Wierstra et al. presented also another similar approach [39] based on an actor-critic architecture. Their policy gradient actor-critic (PGAC) approach involves estimating a PG for an actor through a Policy Gradient Critic which evaluates probability distributions on actions. Instead of using an actor-only method which leads to high variance estimates, they use a dual actor-critic architecture. The actors parameters are updated using a model-based estimated gradient on action probabilities where the model is the Policy Gradient Critic. LSTM is used as memory-capable differentiable recurrent function approximator for both actor and Policy Gradient Critic. Like many conventional TD learning algorithms for POMDPs, the PGAC algorithm uses tow differentiable recurrent function approximators: actor $\pi_{\boldsymbol{\theta}}$ parameterized by $\boldsymbol{\theta}$, and critic $Q_{\mathbf{w}}$ parameterized by $\mathbf{w}$. The difference between PGAC and other methods is that its Policy Gradient Critic's Q-function evaluates probability distributions over actions rather than single actions. The actor $\pi_{\boldsymbol{\theta}}$ outputs at every time step t, deterministically given history h_t probability distribution parameters $\mathbf{p}_t = \pi_{\boldsymbol{\theta}}(h_t)$ from which the agents actions are drawn. The actor is updated by updating $\boldsymbol{\theta}$ in the direction of higher expected future discounted reward as predicted by the Policy Gradient Critic:

$$\Delta\boldsymbol{\theta} = \alpha \sum_i \frac{\partial Q(h_t, \mathbf{p}_t)}{\partial \mathbf{p}_t^{(i)}} \frac{\partial \mathbf{p}_t^{(i)}}{\partial \boldsymbol{\theta}} , \tag{2.34}$$

where α denotes the learning rate. The actor can only be updated if the Policy Gradient Critic provides sufficiently accurate estimations on future discounted rewards. To train the Policy Gradient Critic, on-policy TD learning is used. The TD error δ_t that can be extracted from experience are pairs $\langle h_{t-1}, \mathbf{p}_{t-1}\rangle$ and $\langle h_{t-1}, \mathbf{a}_{t-1}\rangle$:

$$\begin{aligned} \delta_t\langle h_{t-1}, \mathbf{p}_{t-1}\rangle &= r_t + \gamma Q(h_t, \mathbf{p}_t) - Q(h_{t-1}, \mathbf{p}_{t-1}) \\ \delta_t\langle h_{t-1}, \mathbf{a}_{t-1}\rangle &= r_t + \gamma Q(h_t, \mathbf{p}_t) - Q(h_{t-1}, \mathbf{a}_{t-1}) \end{aligned} \tag{2.35}$$

The two TD errors might not provide enough data points to reliably estimate the Policy Gradient Critic's Jacobian. This is because the region around $\mathbf{p}_t$ is not sampled by the actor, although that is the region where the most useful information is localized. Therefore, Wierstra et al. added perturbed samples around $\mathbf{p}_t$ in order to be able to estimate how the Q-values change with respect to $\mathbf{p}_t$. Such samples are provided through a perturbation operation $\mathscr{P}$ that perturbs probability distribution parameters $\mathbf{p}$ onto a new parameter vector $\mathbf{p}^* \sim \mathscr{P}(\mathbf{p})$, such that the expected distribution of actions $\mathbf{a}$ drawn from $\mathbf{a} \sim \mathscr{D}_{\mathbf{p}^*}$ follow the same distribution as actions drawn from the original $\mathbf{a} \sim \mathscr{D}_{\mathbf{p}}$. Thus constructing $\mathbf{p}^*$ values around $\mathbf{p}$ provides informative extra samples which yields the following additional TD error for $\mathbf{p}^*$:

$$\delta_t\langle h_{t-1}, \mathbf{p}^*_{t-1}\rangle = r_t + \gamma Q(h_t, \mathbf{p}_t) - Q(h_{t-1}, \mathbf{p}^*_{t-1}) \tag{2.36}$$

The PGAC approach could be applied on continuous and discrete tasks like the other approach from Wierstra et al. [8].

Hausknecht et al. [9] replaced the first post-convolutional fully-connected layer of a DQN with a recurrent LSTM to make these networks applicable for POMDPs. The resulting Deep Recurrent Q-Network (DRQN) is able to successfully integrate information through time even if only a single frame, e.g. from Atari games or partially observable equivalents, can be seen at each time step. Additionally, the recurrency confers benefits when the quality of observations change during evaluation time. In contrast to e.g. the approach from Wierstra et al. [8], they are able to learn directly from pixels and do not require hand-engineered features because of jointly training convolutional and LSTM layers. They showed that DRQNs can perform the required information integration to resolve short-term partial observability (e.g. to estimate velocities) that is achieved via stacks of frames in the original DQN architecture. Further, they demonstrated that when training with full observations and evaluating with partial observations, DRQN's performance degrades less than DQN's.

Heess et al. [10] extended two related, model-free algorithms for continuous control - DDPG and stochastic value gradients (SVG(0)) [40] - to solve partially observable domains using RNNs trained with BPTT. By using LSTM they are able to solve physical control problems with different memory requirements. These include the short-term integration of information from noisy sensors and the identification of system parameters as well as long-term memory problems that require preserving information over many time steps. DDPG is an off-policy actor-critic algorithm that uses learned approximation of the Q-function to obtain approximate action-value gradients which are used to update a deterministic policy. The counterpart, SVG(0) similarly updates the policy via backpropagation of action-value gradients from an action-value critic but learns a stochastic policy. The core idea of the DDPG algorithm is that for a deterministic policy $\mu_{\boldsymbol{\theta}}$ with parameters $\boldsymbol{\theta}$, and given access to the true action-value function associated with the current policy Q^{μ}, the policy can be updated by backpropagation. Under partial observability the optimal policy and the associated action-value function are both functions of the entire preceding observation-action history h_t. Therefore, the following update rule can be obtained:

$$\nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}} = E_{\tau}\left\{ \sum_t \gamma^{t-1} \nabla_{\mathbf{a}} Q^{\mu}(h_t, \mathbf{a})\Big|_{\mathbf{a}=\mu_{\boldsymbol{\theta}}(h_t)} \nabla_{\boldsymbol{\theta}} \mu_{\boldsymbol{\theta}}(h_t) \right\} , \tag{2.37}$$

where the expectation is written over entire trajectories τ which are drawn from the trajectory distribution inducted by the current policy. Similar to DDPG, SVG(0) updates the policy through backpropagation from the action-value function, but does so for stochastic policies. The stochastic policy is represented by e.g. a Gaussian policy that can be re-parameterized as follows: $\mathbf{a} = \pi_{\boldsymbol{\theta}}(h, \nu) = \mu_{\boldsymbol{\theta}}(h) + \sigma \nu$ where $\nu \sim N(\cdot|0,1)$. The stochastic policy is updated as follows:

$$\nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}} = E_{\tau,\nu}\left\{ \sum_t \gamma^{t-1} \nabla_{\mathbf{a}} Q^{\pi_\theta}(h_t, \mathbf{a})\big|_{\mathbf{a}=\pi_{\boldsymbol{\theta}}(h_t,\nu_t)} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(h_t, \nu_t) \right\}, \tag{2.38}$$

with τ drawn from the trajectory which is conditioned on independently and identically distributed (i.i.d.) draws of ν_t at each time step t. Learning stochastic policies SVG(0) admits on-policy learning whereas DDPG is typically used in an off-policy setting due to the fact that the policy is deterministic but exploration is needed in order to learn the gradient of Q with respect to the actions.

Meuleau et al. [41] followed a completely different approach. Instead of training a neural network they learned finite-state controllers for partially observable environments. Every policiy can be represented in the form of a (possibly infinite) state-automata, also called policy graph. A priori, the optimal solution of a POMDP will be an infinite policy graph. However, because of evident computational limits, the search of polices representable as finite policy graphs may be reduced. Meuleau et al. proposed a model-free algorithm for learning general finite policy graphs of a given size. The algorithm performs stochastic gradient decent in the parameters of the policy graph and therefore ensures to converge to local optima. A policy graph for a given POMDP is a graph where the nodes are labeled with actions $\mathbf{a} \in \mathcal{A}$ and the arcs are labeled with observations $\mathbf{o} \in \mathcal{O}$. Furthermore, there is only one outgoing arc at each node for each possible observation. When the system is in a certain node, it executes the action associated with this node which implies a state transition in the POMDP and eventually a new observation. This observation itself conditions a transition in the policy graph to the destination node of the arc associated with the new observation. They extended the VAPS [42] algorithm which learns a reactive policy through trial-based interactions with the process to be optimized. Because VAPS is limited to learn only memoryless policies, Meuleau et al. extended it so that the structure of the policy graphs does not have to be completely fixed in advance. As a drawback, this approach has to compute complicated gradient on stochastic internal states. In contrast, the approach from Wierstra et al. [8] has only stochastic outputs and eligibility-BPTT disambiguates relevant hidden state automatically, if possible.

Each of the previously described approaches were a model-free approach. Now we will also have a look on a model-based approach for solving POMDPs. Futoma et al. [43] proposed a prediction constrained training for POMDPs which allows effective model learning even in the settings with misspecified models, as it can ignore observation patterns that would distract classic two-stage training. They consider POMDPs with K discrete states, A discrete actions and continuous D-dimensional observations. For each dimension $d \in \{1, 2, ..., D\}$, they independently sample observations $\mathbf{o}_d \sim N(\mu_{kad}, \sigma^2_{kad})$, where k identifies the state just entered ($\mathbf{s}'$) and $\mathbf{a}$ is the action just taken. The parameters $\boldsymbol{\theta}$ define an input-output hidden Markov model (IO-HMM) [44], where the likelihood $p(\mathbf{o}|\mathbf{a}, \boldsymbol{\theta})$ of observations for given actions can be evaluated via DP. Additionally, point based value iteration (PBVI) [45], an efficient solver for small and medium-sized POMDPs, was used. A standard approach for identifying the optimal policy involves two stages: first, transition and observation models are fitted given the data. Then, the learnt POMDP will be solved to obtain a policy. The proposed training objective balances two goals: providing accurate explanations of the data through a generative model (the POMDP), and learning a high-value policy. The new training objective learns $\boldsymbol{\theta}$ by maximizing both the likelihood and an estimated value for the policy $\pi_{\boldsymbol{\theta}}$ given by PBVI:

$$\max_{\boldsymbol{\theta}} \frac{1}{D\left(\sum_n T_n\right)} \sum_{n \in \mathcal{D}^{expl}} \log p(\mathbf{o}_{n,1:T_n} | \mathbf{a}_{n,1:T_n - 1}, \boldsymbol{\theta}) + \lambda \cdot \text{value}^{CWPDIS}(\pi_{\boldsymbol{\theta}}, \pi_{beh}, \mathcal{D}_{beh}, r, \gamma), \tag{2.39}$$

where the first term is the IO-HMM data likelihood and the second term is an off-policy estimate of the value of the optimal policy under the model parameters $\boldsymbol{\theta}$. The objective trades off generative and reward-seeking properties of the model parameters by the tradeoff scalar $\lambda > 0$ which controls how important the reward-seeking term is. The set $\mathcal{D}^{expl}$ contains sequences collected under an exploration policy and thus allows better estimation of the transition and emission parameters $\boldsymbol{\theta}$. Futoma et al. collected rollouts via MC under a reference behavior policy π_{beh} which is known in advance, and reweighed the observed rewards from data collected under π_{beh} using consistent weighted per-decision importance sampling (CWPDIS) [46].

2.3 Deep Reinforcement Learning Methods

This section will introduce several relevant deep RL algorithms which we will use later for our GRL approach in section 3.2 and for the experiments in chapter 4. First, we will present the natural policy gradient which forms a fundamental principle in policy search that will be taken up by some methods in this section. Later, we will describe three on-policy policy search methods and off-policy actor-critic method in detail. All of these algorithms are model-free and use neural networks for approximating optimal policies and/or optimal value functions.

2.3.1 Natural Policy Gradient

The natural policy gradient was introduced in RL by Kakade [47] and follows the well-known natural gradient [48] for optimizing parameterized probability distributions. The natural policy gradient method represents the steepest decent direction based on the underlying structure of the parameter space. Traditional gradient methods typically use an Euclidean metric to determine the update direction of the parameters $\delta\boldsymbol{\theta} = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$, i.e. they assume that all parameter dimensions have similarly strong effects on the resulting distribution. This means that already small changes in $\boldsymbol{\theta}$ might result in large changes of the resulting distribution. The intuition of the natural gradients to limit the distance between two subsequent distributions is also useful in the context of RL, specifically in policy search. To measure the distance between two distributions, the natural gradient uses a second order Taylor approximation of the Kullback-Leibler (KL) divergence

$$KL(\pi_{\boldsymbol{\theta}}(\tau)||\pi_{\boldsymbol{\theta}+\delta\boldsymbol{\theta}}(\tau)) \approx \delta\boldsymbol{\theta}^T \mathbf{F}_{\boldsymbol{\theta}} \delta\boldsymbol{\theta} \leq \epsilon \; , \tag{2.40}$$

where the KL is a similarity measure of two distributions, here two policies. $\mathbf{F}$ is known as the Fisher-information matrix and can be computed for a trajectory distribution $\pi_{\boldsymbol{\theta}}(\tau)$ as follows

$$\mathbf{F}_{\boldsymbol{\theta}} = E_{\pi_{\boldsymbol{\theta}}(\tau)}\left\{\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\tau)^T\right\} . \tag{2.41}$$

The natural gradient $\nabla_{\boldsymbol{\theta}}^{NG} J_{\boldsymbol{\theta}}$ is now defined as

$$\nabla_{\boldsymbol{\theta}}^{NG} J_{\boldsymbol{\theta}} = \mathbf{F}^{-1} \nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}} \; , \tag{2.42}$$

where $\nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}}$ can be estimated by any traditional policy gradient method e.g. with equation 2.1.6 [23].

2.3.2 Trust Region Policy Optimization (TRPO)

Trust Region Policy Optimization (TRPO) [49] is a practical algorithm that is similar to natural policy gradient methods (see section 2.3.1) and is effective for optimizing large non-linear policies such as neural networks. We will consider the single-path method here which can be applied in the model-free setting. The "surrogate" objective $L_{\boldsymbol{\theta}_{old}}(\boldsymbol{\theta})$ uses a constraint on the KL divergence between the new policy $\pi_{\boldsymbol{\theta}}$ and the old policy $\pi_{\boldsymbol{\theta}_{old}}$, i.e. a trust region constraint. Therefore, the TRPO optimization problem for generating a policy update is formulated as

$$\begin{aligned} &\max_{\boldsymbol{\theta}} L_{\boldsymbol{\theta}_{old}}(\boldsymbol{\theta}) = E_{\mathbf{s}\sim\pi_{\boldsymbol{\theta}_{old}}, \mathbf{a}\sim\pi_{\boldsymbol{\theta}}}\left\{\frac{\pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s})}{\pi_{\boldsymbol{\theta}_{old}}(\mathbf{a}|\mathbf{s})} A^{\pi_{\boldsymbol{\theta}_{old}}}(\mathbf{s},\mathbf{a})\right\} \\ &\text{s.t. } E_{\mathbf{s}\sim\pi_{\boldsymbol{\theta}_{old}}}\left\{KL(\pi_{\boldsymbol{\theta}_{old}}(\cdot|\mathbf{s})||\pi_{\boldsymbol{\theta}}(\cdot|\mathbf{s}))\right\} \leq \epsilon \; , \end{aligned} \tag{2.43}$$

where $A^{\pi_{\boldsymbol{\theta}_{old}}}$ is an estimated advantage value that is computed at each state-action pair $(\mathbf{s}_t, \mathbf{a}_t)$ by taking the discounted sum of future rewards along the trajectory. The parameter ϵ specifies the maximum KL divergence in a single update step. In the practical algorithm, this constrained optimization problem is approximately solved by using the conjugate gradient algorithm followed by a line search. This involves two steps:

1. Compute a search direction $s \approx \mathbf{A}^{-1} g$, using a linear approximation to the objective and a quadratic approximation to the constraint, and

2. perform a line search in direction s and ensure that the non-linear objective will improve while satisfying the non-linear constraint.

The parameter g in step 1 is the derivative of the "surrogate" loss function $L_{\boldsymbol{\theta}_{old}}(\boldsymbol{\theta})$ and $\mathbf{A}$ is the Fisher information matrix, i.e., the quadratic approximation to the KL divergence constraint:

$$\overline{KL}(\pi_{\boldsymbol{\theta}_{old}}(\cdot|\mathbf{s})||\pi_{\boldsymbol{\theta}}(\cdot|\mathbf{s})) \approx \frac{1}{2}\delta\boldsymbol{\theta}^T \mathbf{A} \delta\boldsymbol{\theta} \; , \tag{2.44}$$

where $\delta\boldsymbol{\theta} = \boldsymbol{\theta} - \boldsymbol{\theta}_{old}$ and the approximation for the Fisher information matrix is

$$A_{ij} = \frac{\partial}{\partial \theta_i}\frac{\partial}{\partial \theta_j} \overline{KL}(\pi_{\boldsymbol{\theta}_{old}}(\cdot|\mathbf{s})||\pi_{\boldsymbol{\theta}}(\cdot|\mathbf{s})) \, . \tag{2.45}$$

In large-scale problems, the computation of the full matrix $\mathbf{A}$ or its inverse $\mathbf{A}^{-1}$ is with respect to the computational overhead and memory very costly. Therefore, TRPO uses the conjugate gradient algorithm to approximately solve equations of the form $\mathbf{A}x = b$ with matrix-vector products between the averaged Fischer information matrix and arbitrary vectors.

2.3.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [50] optimizes a "surrogate" objective function which forms a pessimistic estimate (i.e. lower bound) of the performance of the policy using stochastic gradient ascent. This method uses some benefits from TRPO, but uses only first-order optimization. Without the KL constraint, the TRPO objective (see equation 2.43) would lead to a very large policy update. Therefore, PPO modifies the objective to penalize changes to the policy that move the probability ratio

$$r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(\mathbf{a}_t|\mathbf{s}_t)}{\pi_{\boldsymbol{\theta}_{old}}(\mathbf{a}_t|\mathbf{s}_t)} \tag{2.46}$$

with $r(\boldsymbol{\theta}_{old}) = 1$ away from 1. The clipped "surrogate" objective is defined as

$$\max_{\boldsymbol{\theta}} L^{CLIP}(\boldsymbol{\theta}) = E_t\left\{\min\left(r_t(\boldsymbol{\theta})A(\mathbf{s}_t,\mathbf{a}_t), \operatorname{clip}(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon)A(\mathbf{s}_t,\mathbf{a}_t)\right)\right\}, \tag{2.47}$$

where ϵ is a hyperparameter e.g. $\epsilon = 0.2$. The first term inside the min() is well known, e.g. from TRPO, while the second term modifies the "surrogate" objective by clipping the probability ratio, which removes the incentive for moving $r_t(\boldsymbol{\theta})$ outside of the interval $[1-\epsilon, 1+\epsilon]$. By taking the minimum of both terms, the final objective is a lower bound on the unclipped objective. This formulation only ignores changes in the probability ratio when it would make the objective improve and includes it if it worsens the objective.

As an alternative to the clipped "surrogate" objective, Schulman et al. [50] proposed also an approach that uses a penalty on the KL divergence and adapts the penalty coefficient so that some target value of the KL divergence d_{targ} is achieved each policy update. The KL-penalized objective is the following:

$$\max_{\boldsymbol{\theta}} L^{KLPEN}(\boldsymbol{\theta}) = E_t\left\{r_t(\boldsymbol{\theta})A(\mathbf{s}_t,\mathbf{a}_t) - \beta KL(\pi_{\boldsymbol{\theta}_{old}}(\cdot|\mathbf{s}_t)||\pi_{\boldsymbol{\theta}}(\cdot|\mathbf{s}_t))\right\}. \tag{2.48}$$

The initial value for β is an additional hyperparameter but is not important in practice because the algorithm quickly adjusts it. The updated β is used for the next policy update by computing $d = E_t\left\{KL(\pi_{\boldsymbol{\theta}_{old}}(\cdot|\mathbf{s}_t)||\pi_{\boldsymbol{\theta}}(\cdot|\mathbf{s}_t))\right\}$ after each optimization:

- If $d < d_{targ}/1.5$, then $\beta \leftarrow \beta/2$
- If $d > 1.5 d_{targ}$, then $\beta \leftarrow 2\beta$

2.3.4 Compatible Policy Search (COPOS)

COPOS [51] is a policy search method that uses an entropy bound in addition to the common KL-bound, known from trust region optimization e.g. TRPO. Adding an entropy bound results in a new update rule which ensures that the policy looses entropy at the correct pace, leading to convergence to a good policy. This means that the entropy bound allows to limit the change in exploration potentially preventing greedy policy convergence. The trust region problem for this case can be formulated as

$$\begin{aligned} &\operatorname{argmax}_{\pi}\; E_{\mathbf{s}\sim\pi_{\boldsymbol{\theta}_{old}}}\left\{\int \pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s}) Q^{\pi_{\boldsymbol{\theta}_{old}}}(\mathbf{s},\mathbf{a}) d\mathbf{a}\right\} \\ \text{s.t.}\;\; &E_{\mathbf{s}\sim\pi_{\boldsymbol{\theta}_{old}}}\left\{KL(\pi_{\boldsymbol{\theta}}(\cdot|\mathbf{s})||\pi_{\boldsymbol{\theta}_{old}}(\cdot|\mathbf{s}))\right\} \leq \epsilon \\ &E_{\mathbf{s}\sim\pi_{\boldsymbol{\theta}_{old}}}\left\{H(\pi_{\boldsymbol{\theta}}(\cdot|\mathbf{s})) - H(\pi_{\boldsymbol{\theta}_{old}}(\cdot|\mathbf{s}))\right\} \leq \beta\,, \end{aligned} \tag{2.49}$$

where the second constraint limits the expected loss in entropy for the new distribution. $H()$ denotes the Shannon entropy in the discrete case and differential entropy in the continuous case. The parameter β specifies the maximum difference in entropy in a single update step.

Compatible function approximation can be used to obtain an unbiased gradient with typically smaller variance. An approximation of MC estimates $\tilde{G}_{\mathbf{w}}^{\pi_{\boldsymbol{\theta}_{old}}}(\mathbf{s},\mathbf{a}) = \boldsymbol{\phi}(\mathbf{s},\mathbf{a})^T\mathbf{w}$ is compatible to the policy $\pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s})$, if the features $\boldsymbol{\phi}(\mathbf{s},\mathbf{a})$ are given by the log gradients of the policy, that are

$$\boldsymbol{\phi}(\mathbf{s},\mathbf{a}) = \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s}) = \boldsymbol{\psi}(\mathbf{s},\mathbf{a}) - E_{\pi_{\boldsymbol{\theta}}(\cdot|\mathbf{s})}[\boldsymbol{\psi}(\mathbf{s},\cdot)]\,, \tag{2.50}$$

where $\boldsymbol{\psi}(\mathbf{s},\mathbf{a})$ are the features e.g. the output of the last fully-connected neural network layer. We call $\boldsymbol{\phi}(\mathbf{s},\mathbf{a})$ the compatible features. The weights $\mathbf{w}$ can be computed by solving

$$\mathbf{w} = \mathbf{F}^{-1}\nabla_{\boldsymbol{\theta}}^{PG} J_{\boldsymbol{\theta}} \ , \tag{2.51}$$

where $\nabla_{\boldsymbol{\theta}}^{PG} J_{\boldsymbol{\theta}}$ is the policy gradient that we already know from section 2.1.6:

$$\nabla_{\boldsymbol{\theta}}^{PG} J_{\boldsymbol{\theta}} = \sum_i \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_i|\mathbf{s}_i) A^{\pi_{\boldsymbol{\theta}_{old}}}(\mathbf{s}_i,\mathbf{a}_i) \ . \tag{2.52}$$

Now the compatible approximation function $\tilde{G}_{\mathbf{w}}^{\pi_{\boldsymbol{\theta}_{old}}}(\mathbf{s},\mathbf{a})$ can be computed using $\mathbf{w}$. The policy update rule can be formed for the constrained optimization problem in equation 2.49 by using the method of Lagrange multipliers and the the compatible approximation function $\tilde{G}_{\mathbf{w}}^{\pi_{\boldsymbol{\theta}_{old}}}(\mathbf{s},\mathbf{a})$:

$$\pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s}) \propto \pi_{\boldsymbol{\theta}_{old}}(\mathbf{a}|\mathbf{s})^{\frac{\eta}{\eta+\omega}} \exp\left(\frac{\tilde{G}_{\mathbf{w}}^{\pi_{\boldsymbol{\theta}_{old}}}(\mathbf{s},\mathbf{a})}{\eta+\omega}\right) \propto \exp\left(\boldsymbol{\psi}(\mathbf{s},\mathbf{a})^T\left(\frac{\eta\boldsymbol{\theta}_{old}+\mathbf{w}}{\eta+\omega}\right)\right) , \tag{2.53}$$

with η associated with the KL divergence bound ϵ and ω associated with the entropy bound β. Note that the value function part of $\tilde{G}_{\mathbf{w}}^{\pi_{\boldsymbol{\theta}_{old}}}(\mathbf{s},\mathbf{a})$ does not influence the updated policy. Hence, if the natural parameterization of the distributions s used in combination with compatible function approximation, then the parametric update is

$$\boldsymbol{\theta} = \frac{\eta\boldsymbol{\theta}_{old}+\mathbf{w}}{\eta+\omega} \ . \tag{2.54}$$

So far, only models with log-linear parameterization have been considered. For more complex models, such as deep neural networks, non-linear parameters $\boldsymbol{\beta}$ for the feature vector $\boldsymbol{\psi}(\mathbf{s},\mathbf{a}) = \boldsymbol{\psi}_{\boldsymbol{\beta}}(\mathbf{s},\mathbf{a})$ need to be introduced. The weights that we have seen before, can be separated into the weights for the log-linear parameters $\boldsymbol{\theta}$ and for the non-linear parameters $\boldsymbol{\beta}$ as $\mathbf{w} = (\mathbf{w}_{\boldsymbol{\theta}}, \mathbf{w}_{\boldsymbol{\beta}})$. Then the update in equation 2.54 is based on $\mathbf{w}_{\boldsymbol{\theta}}$ and the approximate update rule for $\boldsymbol{\beta}$ is

$$\boldsymbol{\beta} = \boldsymbol{\beta}_{old} + \frac{\mathbf{w}_{\boldsymbol{\beta}}}{\eta} \ . \tag{2.55}$$

A softmax distribution is used in the discrete case and the continuous case focuses on a Gaussian distribution with a constant variance where the mean is a product of neural network features and a mixing matrix that could be part of the neural network output layer.

2.3.5 Soft Actor-Critic (SAC)

SAC [52] is one of the first off-policy actor-critic deep RL algorithms that is based on the maximum entropy RL framework, where the actor aims to maximize the expected reward while also maximizing entropy. In contrast to the on-policy algorithms seen before in this section, SAC does not require new samples to be collected for each gradient step and instead aims to reuse past experience in form of a replay buffer. The DDPG algorithm that we have seen in section 2.1.8 can be seen as a deterministic actor-critic algorithm as well as an approximate Q-learning algorithm. Since the combination of both techniques typically makes DDPG difficult to stabilize and very hyperparmeter sensitive, SAC uses a stochastic actor with an entropy maximization objective which results in a more stable and scalable algorithm. In maximum entropy RL a more general objective (than the expected sum of rewards used in conventional RL, see equation 2.6) will be considered:

$$J = \sum_{t=0}^{T} E_{(\mathbf{s}_t,\mathbf{a}_t)\sim\pi}\{r(\mathbf{s}_t,\mathbf{a}_t) + \alpha H(\pi(\cdot|\mathbf{s}_t))\} \ , \tag{2.56}$$

where the temperature parameter α determines the relative importance of the entropy term against the reward and thus controls the stochasticity of the optimal policy. This objective has the advantage that the policy is encouraged to explore more widely. Further, the policy will commit equal probability to multiple actions that seem equally attractive.

SAC learns a parameterized soft state-value function $V_{\psi}(\mathbf{s}_t)$, two soft Q-functions $Q_{\boldsymbol{\theta}_1}(\mathbf{s}_t,\mathbf{a}_t)$ and $Q_{\boldsymbol{\theta}_2}(\mathbf{s}_t,\mathbf{a}_t)$, and a traceable policy $\pi_{\boldsymbol{\phi}}(\mathbf{a}_t|\mathbf{s}_t)$. In the following, we will write the Q-functions as $Q_{\boldsymbol{\theta}}(\mathbf{s}_t,\mathbf{a}_t)$ for simplicification because both Q-functions have the same form. Therefore, the parameters of these networks are $\boldsymbol{\psi}$, $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$. Function approximatiors will be used for the Q-function and the policy, and the algorithm alternates between optimizing both networks with stochastic gradient descent. Since the soft state-value function is related to the Q-function and policy, there is no need to include a separate function approximator:

$$V_\psi(\mathbf{s}_t) = E_{\mathbf{a}_t \sim \pi_\phi} \left\{ Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) \right\} . \tag{2.57}$$

The soft state-value function is trained to minimize the squared residual error

$$J_V(\psi) = E_{\mathbf{s}_t \sim \mathcal{D}} \left\{ \frac{1}{2} \Big(V_\psi(\mathbf{s}_t) - E_{\mathbf{a}_t \sim \pi_\phi} \left\{ Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) \right\} \Big)^2 \right\} , \tag{2.58}$$

where $\mathcal{D}$ is the distribution of previously sampled states and actions i.e. a replay buffer. The gradient of the soft state-value function can be derived with an unbiased estimator

$$\hat{\nabla}_\psi J_V(\psi) = \nabla_\psi V_\psi(\mathbf{s}_t) \big(V_\psi(\mathbf{s}_t) - Q_\theta(\mathbf{s}_t, \mathbf{a}_t) + \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) \big) , \tag{2.59}$$

where the actions are sampled according to the current policy, instead of the replay buffer.
The soft Q-function can be trained to minimize the soft Bellman residual

$$J_Q(\theta) = E_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left\{ \frac{1}{2} \Big(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - r(\mathbf{s}_t, \mathbf{a}_t) - \gamma E_{\mathbf{s}_{t+1} \sim \pi_\phi} \left\{ V_{\psi_{targ}}(\mathbf{s}_{t+1}) \right\} \Big)^2 \right\} , \tag{2.60}$$

which uses a target value network $V_{\psi_{targ}}$, where ψ_{targ} can be an exponentially moving average of the value network weights. $J_Q(\theta)$ can be optimized with stochastic gradients

$$\hat{\nabla}_\theta J_Q(\theta) = \nabla_\theta Q_\theta(\mathbf{s}_t, \mathbf{a}_t) \big(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - r(\mathbf{s}_t, \mathbf{a}_t) - \gamma V_{\psi_{targ}}(\mathbf{s}_{t+1}) \big) . \tag{2.61}$$

Finally, the policy can be learned by using the likelihood ratio gradient estimator which does not require backpropagating the gradient through the policy and the target network. With the reparameterization trick we can derive a lower variance estimator for the policy by using a neural network transformation $\mathbf{a}_t = f_\phi(\epsilon_t; \mathbf{s}_t)$, where ϵ_t is an input noise vector, sampled from a fixed distribution e.g. a spherical Gaussian. The objective for optimizing the policy can be written as

$$J_\pi(\phi) = E_{\mathbf{s}_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}} \left\{ \log \pi_\phi(f_\phi(\epsilon_t; \mathbf{s}_t)|\mathbf{s}_t) - Q_\theta(\mathbf{s}_t, f_\phi(\epsilon_t; \mathbf{s}_t)) \right\} , \tag{2.62}$$

where the policy π_ϕ is defined implicitly in terms of f_ϕ. Now, the gradient can be approximated with the following unbiased gradient estimator, which extends DDPG style PGs to any traceable stochastic policy:

$$\hat{\nabla}_\phi J_\pi(\phi) = \nabla_\phi \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) + \big(\nabla_{\mathbf{a}_t} \log \pi_\phi(\mathbf{a}_t|\mathbf{s}_t) - \nabla_{\mathbf{a}_t} Q_\theta(\mathbf{s}_t, \mathbf{a}_t) \big) \nabla_\phi f_\phi(\epsilon_t; \mathbf{s}_t) , \tag{2.63}$$

where $\mathbf{a}_t$ is evaluated at $f_\phi(\epsilon_t; \mathbf{s}_t)$. The practical SAC algorithm uses two Q-functions to mitigate positive bias in the policy improvement step that is known to degrade the performance of value based methods. The two Q-functions are parameterized with the parameters θ_i where $i \in \{1, 2\}$, and trained independently to optimize the objective $J_Q(\theta_i)$ in equation 2.60. Then, the minimum of the Q-functions is used for the value gradient (see equation 2.59) and policy gradient (see equation 2.63).

3 Guided Reinforcement Learning

In this chapter we present a novel guided RL approach, called guided reinforcement learning (GRL) for solving POMDP problems. With our model-free approach we guide RL algorithms like COPOS or SAC with additional full state information during the learning process to increase their performance solving POMDP in the test phase. The guidance is mainly based on mixing samples containing full or partial state information. In order to gain more flexibility, we choose a model-free approach because they do not need to learn an accurate model of the environment first. Before we present of our GRL approach in detail we will refer to some related approaches which have been published recently. Especially guided policy search algorithms inspired us to propose an easier and more effective approach that has not been covered in prior work. Results for our GRL approach on several POMDP problems can be seen in chapter 4.

3.1 Related Guided Policy Search Approaches

This section introduces guided policy search by presenting some related approaches from literature. Guided policy search algorithms assist the policy learning with additional information or mechanisms. The type of assistance can vary depending on the domain of the problem or on the used policy search method.
Zhang et al. [11] presented an algorithm which combines model predictive control (MPC) with reinforcement learning in the framework of guided policy search. Their algorithm can be considered as one of the closest work to our proposed approach described in section 3.2. Zhang et al. [11] use MPC which receives the full state of the system only during the training phase in order to supervise the learning of the neural network policy. MPC is an effective and reliable method for controlling robotic systems [11]. In their specific domain of autonomous arial vehicles, training deep neural network policies with guided policy search yields the following two main advantages. First, the final neural network policy does not need to use the same input as MPC and can therefore restrict the inputs to only those observations that are directly available during test time. Second, the neural network policy is computationally much less expensive than MCP and can be performed onboard on specialized hardware while the MPC solution can be computed offboard during training. Instead of using an offline trajectory phase to generate the controller that is then executed on the real system as proposed in prior applications [53], [54], [55], Zhang et al. [11] replaced the offline trajectory optimization in guided policy search with online MPC. Offline optimization might require a learned model of the true system dynamics [12] which makes these methods liable to fail catastrophically when the model is inaccurate [11]. The approach by Zhang et al. [11] is still a model-based approach because they combine MPC with offline trajectory optimization to generate guiding samples for guided policy search and accordingly assume access to an approximate model of the system dynamics in the training phase. Additionally, it should be mentioned that their approach is targeted on continuous action and state spaces.

Levine et al. proposed a guided policy search algorithm [12] which uses trajectory optimization. Later they published several modifications of the guided policy search algorithm for continuous domains that has been applied to locomotion [53], robotic manipulation [54], and vision-based robotic control [55]. The initial algorithm [12] uses an importance sampled variant of the likelihood ratio estimator to incorporate the off-policy guiding samples generated with differential dynamic programming (DDP) into the policy search. Importance sampling is a technique for estimating an expectation $E_p[f(x)]$ with respect to a distribution $p(x)$ using samples drawn from a different distribution $q(x)$ [12]. To include samples from multiple distributions Levine et al. [12] use a fused distribution where each sample is either a previous policy or a guiding distribution constructed by DDP. The DDP solutions can be initialized with human demonstrations or with an offline planning algorithm. The policy search alternates between optimizing the objective $\Phi(\theta)$ and gathering new samples from the current policy. Optionally, adaptive guiding samples could be used by adding new guiding samples to the sample set at each iteration. The policy search component is model-free, while DDP requires a model of the system dynamics [12].
In their following publication [53] Levine et al. introduce a constrained variant of their guided policy search method that gradually brings the trajectories into agreement with the policy, ensuring that the trajectories and policy match at convergence. They accomplish the matching by gradually enforcing a constraint between the trajectories and the policy using dual gradient decent, resulting in an algorithm that iterates between optimizing the policy to agree with the trajectories, optimizing the trajectories to minimize expected cost and agree with the policy, and updating the dual variables to improve constraint satisfaction.
Both mentioned guided policy search methods [12] [53] use model-based trajectory optimization algorithms which require known, differentiable system dynamics. With a new hybrid approach [56] Levine et al. can perform guided policy search under unknown dynamics. The hybrid method uses iteratively refitted local linear models to optimize trajectory

distributions for large, continuous problems. By fitting time-varying linear dynamics models and not relying on learning a global model they could speed up learning. They use local linear models to efficiently optimize a time-varying linear-Gaussian controller, which induces an approximate Gaussian distribution over trajectories. Since the trajectory distribution is approximately Gaussian, this can be done efficiently, in terms of both sample count and computation time [56]. While the approaches from Levine et al. ([53] - [56]) assume trajectory optimization with uni-modal trajectories, our simple GRL approach allows to combine this approach with a variety of existing model-free RL algorithms as well as a variety of setting where these algorithms can be applied to.

Zhang et al. [57] adapted the trajectory-centric guided policy approach under unknown dynamics from Levine et al. [56] to the task of training policies with internal memory. They added continuous-valued memory states to both the trajectory-centric teacher and the final neural network policy. The trajectory optimization phase chooses the values of the memory states that will make it easier for the policy to produce the right actions in future states, while the supervised learning phase encourages the policy to use memorization actions to produce those memory states [57]. Instead of directly optimizing RNNs with backpropagation through time, Zhang et al. consider a different method for integrating memory into the policy. In their approach [57], the memory states are directly concatenated to the physical state of the system and the observations to produce an augmented state and augmented observations. They also concatenate memory writing actions to the actions of the system to produce an augmented action. Since the memory states and memory writing actions are simply appended to the observation and action vectors, the supervised learning procedure for the policy remains identical, and the policy is automatically trained to use the memory actions to mimic the pattern of memory activations optimized by the trajectory-centric "teacher" algorithms [57].

Montgomery et al. [58] showed that guided policy search algorithms can be interpreted as an approximate variant of mirror decent, where the projection onto the constraint manifold is not exact. Their new algorithm is partially inspired by previous work from Levine et al. [56], [55]. Instead of constraining each local policy against the previous local policy, Montgomery et al. constraint it directly against the global policy, and set the surrogate cost to be the true cost. They perform the optimization on the space of trajectory distributions, with a constraint that the policy must lie on the manifold of policies with the chosen parameterization. The optimization is solved in two alternating steps by mirror decent. The first step finds a new distribution that minimizes the cost and is close to the previous policy, while the second step projects this distribution onto the constraint set [58].

Carr et al. [59] combine techniques from machine learning and formal verification in their model-based approach. First, they learn a policy via RNNs and data stemming from knowledge of the underlying structure of POMDPs. Secondly, they extract a finite-memory candidate strategy from the RNN and use it directly on a given POMDP which is resulting in the Markov chain inducted by the POMDP and the strategy. By applying formal verification they can disclose whether temporal logic specifications are satisfied or not. If specifications are not satisfied they generate a counterexample [60], that points to parts of the Markov chain, which are critical for the specifications. With the diagnostic information from counterexamples they could improve the strategy by iteratively training the RNN. Their proposed method focuses on discrete state spaces as well as on discrete action spaces.

For the specific domain of swarm systems with distributed partial observability of the state, Hüttenrauch et al. [61] presented an guided actor-critic approach for learning policies for a set of homogeneous agents who try to fulfil a cooperative task. While the actor learns a decentralized control policy operation on locally sensed information, they provide the critic the full system state to guide the swarm through the learning process. Hüttenrauch et al. [61] follow a similar scheme as the DDPG by learning a Q-Function based on the global state information to evaluate the whole swarm behavior, while the actions chosen by the agents are determined based on observation histories only. The difference to the algorithm for learning the Q-Function used in DDPG is that a single policy function is used to compute the actions of all agents, whereas in DDPG the policy would be different for each dimension of the action vector when viewing the actions of all agents as a large combined action vector [61].

3.2 Guided Reinforcement Learning (GRL) Approach

In this section our own guided RL approach, called GRL will be presented which is mainly based on mixing samples containing full or partial state information during the learning process. While most of the existing guided RL methods are model-based approaches as described in section 3.1, we focus on a model-free approach. Model-based approaches have to learn a global model of the environment first which can be challenging for complex tasks and can require extensive computation. In contrast, model-free approaches learn a policy directly from interactions with the environment and offer much more flexibility because they do not require an accurate representation of the environment. Depending on the complexity of the problem, a sufficient amount of interactions with the environment has to be provided to receive good results with model-free approaches. Our simple GRL approach could be combined with a variety of existing model-free RL algorithms. We demonstrate the example usage with the on-policy algorithm COPOS and the off-policy algorithm SAC. Both algorithms combined with our approach are also used later for the experiments in chapter 4.

3.2.1 General Functionality

We assume that we have access to the full observations during the learning process and only to partial observations at test time. The partial observability at test time could either be that there are parts of the full state information which are missing or noisy, or that the complete observations are noisy. Alternatively interpreted, the problem to be solved could be a POMDP and during the learning process there are additional information of the full state provided. We assume also that we know the dimensions of the state observation which will be not available or noisy during the test phase. Our GRL approach may help the underlaying algorithm with the additional full state information to prevent being stuck in poor local optima. We will describe our approach based on samples τ which are tuples containing information like an observation $\mathbf{o}$, an action $\mathbf{a}$ or the reward r. Samples are gathered form interactions with the environment, called sampling and the contained information depends on the used RL algorithm. Policy search algorithms usually use a certain number of samples, also called sample set $\mathcal{T}$, for the optimization and the policy update. The following procedure will be performed at the learning phase:
At the beginning of training, all samples τ contain information of the fully observable state which means that the ratio p_{full} of full state observations is 100% ($p_{full} = 1$). Then the ratio of full state observations p_{full} will be decreased in each learning iteration by a fixed percentage (e.g. $d = 0.005$). Now the sample set $\mathcal{T}$ for one single optimization iteration contains p_{full} samples with full state information and $1-p_{full}$ samples with only partial information. At some point p_{full} will reach the value 0 which implies that the current sample set $\mathcal{T}$ contains only samples with partial information. From that point ($p_{full} = 0$) the training will be continued with partial information in order to optimize the learned policy for partial observability. Figure 3.1 visualizes this general procedure.
For the parameter d we found out that mixing samples until the half of the iterations in the learning phase is always a good starting point. If the RL algorithm should be executed for e.g. 1000 iterations, then parameter d will be set to $1/500$, $d = 0.002$. Depending on the problem, parameter d can be optionally fine tuned by slightly varying it to a bigger or smaller value.

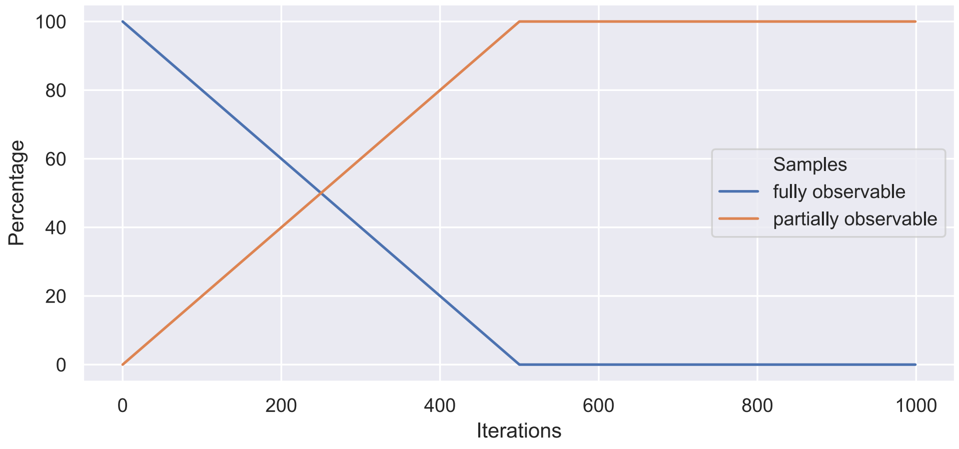

Figure 3.1.: Example distribution of generated samples during the learning process of the GRL approach. The learning phase starts with only fully observable samples, then fully and partially observable samples are mixed until a previous defined point (500 iterations in this example). From that point the training will be continued with partial information to optimize the policy under partial observability.

3.2.2 Implementation Details

The general functionality of our GRL approach has been introduced in the previous section. Now, further details with respect to the used policy search algorithm COPOS and actor-critic algorithm SAC will be provided. Both algorithms receive a modified input from interactions with the environment. Therefore we build a wrapper for all used environments, which will be discussed later in section 4.1. The wrapper returns a history of the last fixed number of observations instead of a single observation. With this limited memory we turn the used RL algorithms into memory-based approaches that are more suitable for solving POMDPs, see section 2.2.

COPOS uses a set of samples $\mathcal{T}$ which are received from interactions with the environment during the sampling in each iteration. The length of $\mathcal{T}$ is defined as batchsize ($|\mathcal{T}| =$ batchsize). The environment returns always a fully observable sample after taking an action in the training phase. The received fully observable sample will be directly modified into a partially observable sample until the maximum number of partially observable samples for the current iteration is reached, which is $|\mathcal{T}| \cdot (1 - p_{full}) = \text{batchsize} \cdot (1 - p_{full})$. When the sampling is done, the resulting sample set $\mathcal{T}$ could be described as a concatenation of two subsets, the set of fully observable samples $\mathcal{T}_{full}$ and the set of partially observable samples $\mathcal{T}_{partial}$:

$$\mathcal{T} = \mathcal{T}_{full} \cup \mathcal{T}_{partial}$$

The modification of fully observable samples into partially observable samples includes overwriting the known dimensions of the state observation which will not be available in the POMDP problem with zeros. These dimensions will be overwritten and in addition, a flag that tells the algorithm wether full or partial information is available will be flipped in each step of the history contained in a given sample. At test time this modification does not have to be performed because then the environments or our wrappers directly return partially observable observations. The COPOS algorithm for discrete and continuous action space as well as the application of our GRL approach as a guided version of COPOS was implemented using the OpenAI baselines framework [62].

For SAC our GRL approach has to be implemented differently because there is no explicit sampling phase to collect a fixed and individual sample set for the optimization. In SAC the optimization will be performed after each (or each few) interaction(s) with the environment based on a replay buffer. The replay buffer can be seen as a rolling buffer with a fixed length where a sample will be added after each agent-environment interaction. Therefore, the decision whether a sample is fully or partially observable has to be made after each interaction. We model this decision by sampling random choices from a discrete probability distribution. The probability that a sample is fully observable is the ratio for fully observable samples, defined in the previous section, that means $P(\text{"fully observable"}) = p_{full}$. Consequently, the probability that a sample is partially observable is denoted as $P(\text{"partially observable"}) = 1 - p_{full}$. The probability distribution and p_{full} will be adjusted after each interaction with the environment instead of after each iteration as mentioned in the previous section. Thus, the parameter d for decreasing p_{full} is directly defined based on the number of total time steps (instead of iterations) for which the learning should be performed. For example, if new samples should be mixed until the half of the total learning time steps, e.g. 1 million in this example, then d will be set to $d = 2/10^6 = 0.000002$. Note, that in the special case of using the SAC algorithm, the replay buffer should not be too long, otherwise the optimization might still take fully observable samples into account long after the last fully observable sample was added to the replay buffer. In contrast, setting the replay buffer length too short, the performance of SAC will decrease significantly. We modified the implementation of the SAC algorithm from the Stable Baselines framework [63] to be able to apply our GRL approach.

4 Experimental Results

This chapter covers the introduction in environments and tasks with continuous and discrete action space that we used to design experiments for the evaluation of our method. Additionally, we will present empirical results on the designed experiments to answer the following research questions:

1. How does our GRL approach perform on POMDP problems compared to other model-free RL algorithms? Could our approach lead the policy search algorithm COPOS and actor-critic algorithm SAC to produce better results in contrast to learning only on partial state information?

2. Is our approach useful when the quality of observations is bad? How does the application of our approach affect the results in comparison with training directly on noisy observations?

To clarify the first question we analyzed the performance of our approach and other RL algorithms on different POMDP tasks. Especially, we compared the results of our approach in combination with the COPOS algorithm and the SAC algorithm against training these algorithms either with partial or full state information. For answering the second question we designed a continuous control task where all observations returned by the environment are noisy. Also on this task we compare our approach against training on either noisy or the original state information. For some of the tasks we provide results for additional baseline algorithms. In section 4.2 we mention further detail on the setup of our experiments and the parametrization.

4.1 Environments and Tasks

We differentiate between environments with continuous and discrete action space because some of the used algorithms are not implemented or different for both classes of tasks. For example in the COPOS algorithm the policy is modelled as Gaussian policy for continuous action tasks and as softmax policy for discrete action tasks. In the following we will describe the original problems and our modifications as well as details on the observation history that we use as input for the RL algorithms. As discrete action task the RockSample [13] problem will be introduced. The continuous action space tasks that we evaluate are LunarLander-POMDP [64], Noisy-LunarLander and six continuous control tasks in the MuJoCo [15] physics simulator.

4.1.1 Continuous Action Space

In this section we will introduce the LunarLander-POMDP and Noisy-LunarLander environment which are both based on the OpenAI Gym [14] environment LunarLanderContinuous-v2. Further, we will introduce six different MuJoCo tasks which are also implemented in the OpenAI Gym framework. The Noisy-LunarLander is designed to answer the second research question while all the other environments focus on answering the first research question.

LunarLander-POMDP

The LunarLander-POMDP environment is a modification that turns the LunarLanderContinuous-v2 environment into a partially observable problem and was initially developed by Rong Zhi [64]. First, we describe the original LunarLanderContinuous-v2 task and discuss the modification into LunarLander-POMDP later. LunarLanderContinuous-v2 is a 2D continuous control task in the Box2D simulator where a lander agent attempts to land on landing pad without collision. The landing pad is always at coordinates (0,0) and the lander starts from the top of the screen. Figure 4.1a illustrates the environment including the lander, landing pad and landscape that is initialized randomly at start. An episode finishes if the lander crashes, leaves the screen or comes to rest so landing outside of the landing pad is also possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt but spending less fuel is beneficial in terms of the reward. The reward function for time step t is defined as:

$$\begin{aligned} r_t = &-100\left(\sqrt{p_{x,t}^2 + p_{y,t}^2} - \sqrt{p_{x,t-1}^2 + p_{y,t-1}^2}\right) - 100\left(\sqrt{v_{x,t}^2 + v_{y,t}^2} - \sqrt{v_{x,t-1}^2 + v_{y,t-1}^2}\right) - 100\left(|\theta_t| + |\theta_{t-1}|\right) \\ &+ 10\left(g_{l,t} - g_{l,t-1}\right) + 10\left(g_{r,t} - g_{r,t-1}\right) - 0.3\left(e_{m,t} - e_{m,t-1}\right) - 0.3\left(e_{s,t} - e_{s,t-1}\right) + 100\left(1 - a_t\right) - 100 o_t, \end{aligned} \tag{4.1}$$

where the x and y position of the lander is represented by p_x and p_y, the velocity of the lander is represented by v_x and v_y , the angle of the lander is represented by θ, the ground contact of the left and right leg of the lander is represented by g_l and g_r, the power of the main and side engines is represented by e_m and e_s, the status if the lander is awake is represented by a and the game over status if the lander crashes or leaves the screen is represented by o. The position p_x and p_y, velocity v_x and v_y, angle θ, engine power p_m and p_s are continuous values. In contrast, the ground contact g_l and g_r, awake flag a and the game over flag o are discrete values (0 or 1). The action that the agent can take is a vector of two continuous values from -1 to 1. The first value e_m controls the main engine where -1 to 0 means off and 0 to 1 means 50% to 100% power. The second value e_s controls the orientation engines where -1.0 to -0.5 means firing the left engine, 0.5 to 1 means firing the right engine and -0.5 to 0.5 means both engines off. The composition of the mentioned values into an 8-dimensional observation and 2-dimensional action space can be seen in table 4.1.

Observation	Symbol	Variable Type
position x	p_x	continuous
position y	p_y	continuous
velocity x	v_x	continuous
position y	v_y	continuous
angle	θ	continuous
angular velocity	ω	continuous
ground contact left leg	g_l	discrete (0 or 1)
ground contact right leg	g_r	discrete (0 or 1)
alarm signal [1]	b	discrete (-1 or 1)

(a)

Action	Symbol	Variable Type
main engine	e_m	continuous
side engine	e_s	continuous

(b)

Table 4.1.: Observation space (a) and action space (b) of the LunarLanderContinuous-2 environment and the LunarLander-POMDP environment. The alarm signal b is only available for LunarLander-POMDP and tells whether the lander is in the blind area where it does not perceive any information of the environment or not.

In the LunarLander-POMDP environment the action space and reward function remain the same as in LunarLander-Continuous-v2. The modification that has been made to turn LunarLander-POMDP into a partially observable problem, is adding a "blind" area between the starting point on the top and the landing pad. When the lander agent enters the blind area, it cannot perceive any information of the environment until it leaves the area. Instead of information of the environment, the lander receives an alarm signal b that has the value -1 when the lander is in this "dangerous" blind area and 1 when the lander is outside. The height of the blind area can be specified with a parameter. Figure 4.1b shows the LunarLander-POMDP environment with the blind area which is represented by a grey row.

Instead of using the original observation output from the environment, we used a history of fixed length as input for the RL algorithms. With this memory of a finite horizon H we turn the used RL algorithms into memory-based approaches that are more suitable for solving POMDPs, see section 2.2. The history includes the last H observations $\mathbf{o}$, actions $\mathbf{a}$ and a flag f that tells the algorithm whether full or partial state information is available. For this task we define the used history for time step t as $\boldsymbol{h}_t = \left(f_t, \mathbf{o}_t, \mathbf{a}_{t-1}, ..., f_{t-H}, \mathbf{o}_{t-H}, \mathbf{a}_{t-H-1}\right)$. If the flag f has the value 1, the given sample is for training on full observability, while value 0 indicates a sample is for training on partial observability. In the test phase f always has the value 0 which tells the algorithm that all samples might be partial observable for the case that the lander is in the blind area.

Noisy-LunarLander

Based on the LunarLanderContinuous-v2 environment we built a modification, called Noisy-LunarLander, to examine how the results behave when the quality of the observations gets worse and to investigate if our GRL approach could help to learn a policy that is more robust to uncertainty in observations. The observation and action space as well as the reward function remain the same as in the LunarLanderContinuous-v2 environment, see table 4.1 and equation

[1] only in LunarLander-POMDP

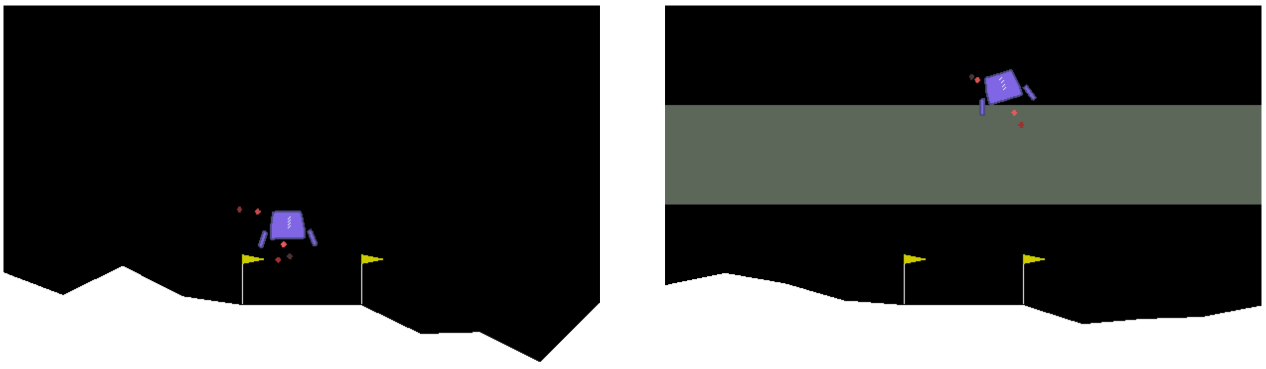
(a) LunarLanderContinuous-v2 **(b)** LunarLander-POMDP

Figure 4.1.: Renderings of the LunarLanderContinuous-v2 (a) and LunarLander-POMDP (b) environment where a lander agent attempts to land on landing pad without collision. In LunarLander-POMDP the "blind" area with height of 1/4 of the environment height is represented as grey row.

4.1. Our modification into Noisy-LunarLander is that we added Gaussian noise by randomly drawing samples from a normal (Gaussian) distribution, represented by the mean μ and the standard deviation σ. Unlike the other experiments, we do not use a history and a flag that tells the algorithm whether there are full/partial state information available or original/noisy observations respectively. Applying a history including observations, previous actions and a flag should also be possible and should improve the results even further but we refrained that for now because we want to investigate this problem in a more principal way first. Even with this slightly different setup, the application of our GRL approach is possible with the difference that the algorithm does not have to flip a flag as described in 3.2.2. Instead of overwriting the additional full state information with zeros, the algorithm adds Gaussian noise to the original observation returned from the environment.

MuJoCo Tasks

The MuJoCo stands for Multi-Joint dynamics with Contact and is a physics engine. There exist several well-known continuous control benchmark tasks which are also included in the OpenAI Gym framework. Specifically, we focus on the two-dimensional tasks HalfCheetah-v2, Hopper-v2, InvertedDoublePendulum-v2, Reacher-v2, Swimmer-v2 and Walker2d-v2. Figure 4.2 shows a captured frame for each of the mentioned tasks. Typical observations for these tasks include positions, angles, velocities or forces. The environment can be controlled by e.g. specifying the torques for the motors in the hinge joints.

In the following we will give a short description and the reward function for each of the tasks we used for our experiments:

- **HalfCheetah-v2:** A robot represents one half of a cheetah by a torso, a front leg and a back leg which both contain hinge joints between foot, shin, thigh and torso. The goal is to make this robot run as fast as possible. The reward function takes the x position before (x_t) and after (x_{t+1}) taking action $\mathbf{a}_t$ which contains the torques for the motors in the hinge joints as well as the difference in time dt between x_{t+1} and x_t:

$$r_t = \frac{x_{t+1} - x_t}{dt} - 0.1 \sum_{a_i \in \mathbf{a}_t} a_i^2 \,. \tag{4.2}$$

- **Hopper-v2:** The goal of this task is to make a one-legged robot hop forward as fast as possible. The robot consists of one torso, one thigh, one leg and one foot. The reward function takes again the positions x_t and x_{t+1}, the action $\mathbf{a}_t$ and the difference in time dt between x_{t+1} and x_t:

$$r_t = \frac{x_{t+1} - x_t}{dt} + 1 - 0.001 \sum_{a_i \in \mathbf{a}_t} a_i^2 \,. \tag{4.3}$$

- **InvertedDoublePendulum-v2:** In this task the agent has to balance a double inverted pendulum on a cart. The pendulum itself is unstable, meaning that it will fall down unless it is controlled by the motor in the cart. The reward function takes the x and y position of the top end of the second pole as well as the velocities of both joints $v_{t,1}$ and $v_{t,2}$ after taking action $\mathbf{a}_t$:

$$r_t = 10 - 0.01 x_{t+1}^2 - (y_{t+1} - 2)^2 - 0.001 v_{t+1,1}^2 - 0.005 v_{t+1,2}^2 \,. \tag{4.4}$$

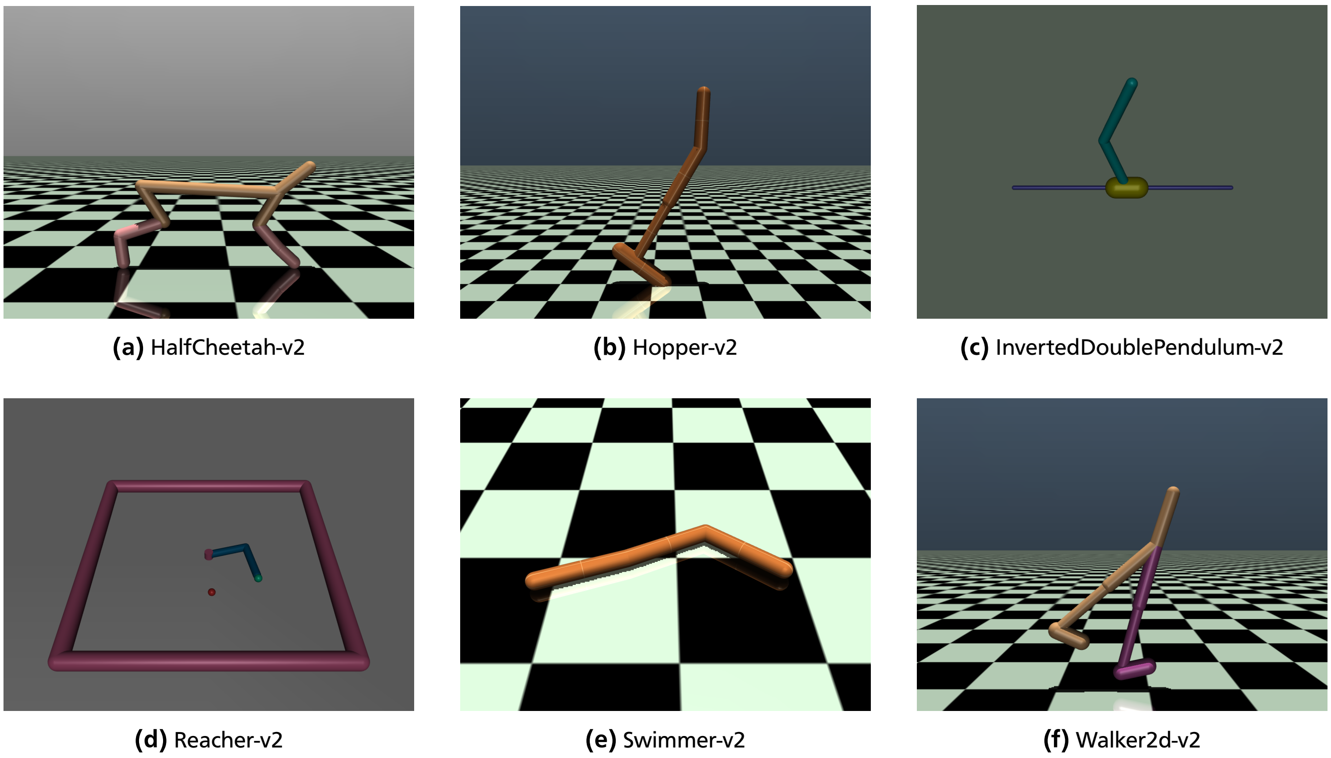
(a) HalfCheetah-v2 (b) Hopper-v2 (c) InvertedDoublePendulum-v2 (d) Reacher-v2 (e) Swimmer-v2 (f) Walker2d-v2

Figure 4.2.: Renderings for six continuous control tasks simulated by the MuJoCo physics engine. The agent can control the motors in the hinge joints or sliders.

- **Reacher-v2:** Here a robot arm attempts to reach a randomly located target. The agent can control the torques of the two hinge joints of the robot arm. The reward function takes the x and y position of the robots fingertip and target as well as the action $\mathbf{a}_t$:

$$r_t = -\sqrt{(x_t - x_{target})^2 + (y_t - y_{target})^2} - \sum_{a_i \in \mathbf{a}_t} a_i^2 \ . \tag{4.5}$$

- **Swimmer-v2:** This task involves a swimming robot in a viscous fluid, where the goal is to make it swim forward as fast as possible, by actuating the motors for the two hinge joints. The reward function takes the positions x_t and x_{t+1}, the action $\mathbf{a}_t$ and the difference in time dt between x_{t+1} and x_t:

$$r_t = \frac{x_{t+1} - x_t}{dt} - 0.0001 \sum_{a_i \in \mathbf{a}_t} a_i^2 \ . \tag{4.6}$$

- **Walker2d-v2:** The goal of this task is to make a bipedal robot walk forward as fast as possible. The robot consists of one torso, two thighs, two shins and two feet. The reward function takes again the positions x_t and x_{t+1}, the action $\mathbf{a}_t$ and the time difference dt between x_{t+1} and x_t:

$$r_t = \frac{x_{t+1} - x_t}{dt} + 1 - 0.001 \sum_{a_i \in \mathbf{a}_t} a_i^2 \ . \tag{4.7}$$

In order to create partially observable environments, we modified the MuJoCo tasks listed above by deactivating parts of the observation. Specifically, we select a number of dimensions of the observation vector and overwrite them with zeros. For the selection of these dimensions for the individual MuJoCo tasks, we run some preliminary tests and chose dimensions whose deactivation has a noticeable impact on the overall performance. A detailed overview is given in table 4.2.

Similar to LunarLander-POMDP and RockSample, we used a history of fixed length as input for the RL algorithms in both cases: fully observable and partially observable. Further, we defined the history and flag which tells the algorithm whether full or partial state information is available in the same way, as we have already already defined for the LunarLander-POMDP environment.

Task	Ob. Space Dimensionality	Inactive Ob. Dimensions	Description
HalfCheetah-v2	17	3-10	angle of front/back thigh, shin and foot joint
			x and y velocity
Hopper-v2	11	3-7	angle of thigh, leg and foot joint
			x and y velocity
InvertedDoublePendulum-v2	11	1-4	angle of both pole joints
			y position of both pole joints
Reacher-v2	11	5-11	target x and y position
			angular velocity of both joints
			distance between fingertip and target
Swimmer-v2	8	2-6	angle of both joints, x and y velocity
			angular velocity of torso
Walker2d-v2	17	9-14	x and y velocity, angular velocity of torso
			angular velocity of right thigh, shin and foot joint

Table 4.2.: Observation (Ob.) space dimensionality and the dimensions we deactivated for the partially observable versions of the MuJoCo tasks. The column "Description" is for the inactive observation dimensions in the POMDP versions.

4.1.2 Discrete Action Space

The experiments on discrete action space will be performed on the partially observable environment RockSample. We used two different sizes of the problem in order to answer the first research question and reported the results in section 4.3.

RockSample

RockSample is a well known benchmark problem for POMDP algorithms and was first mentioned in [13]. In this scalable problem, a rover can achieve reward by sampling rocks in an area modelled as small grid and by continuing its traverse (reaching the exit by leaving the grid on the right side). Some of the rocks have scientific values denoted as "good". The rover always knows its own position and the position of the rocks but it does not know which rocks are valuable. Additionally, the rover is equipped with a noisy long-range sensor to gather information on the rocks before choosing whether to sample a rock or not. The accuracy of the rover's long-range sensor, called efficiency η, depends on the distance between the position of rover and the rocks. The efficiency η decreases exponentially as a function of Euclidean distance which means that the sensor always returns a correct value at $\eta = 1$ and returns the value for "good" or "bad" with 50% probability each at $\eta = 0$. Values between 0 and 1 for η are combined linearly. An instance of the partially observable RockSample problem is defined by the map size $n \times n$ as well as by k rocks and is expressed as RockSample(n, k). The value of a rock i is denoted as $RockType_i = \{Good, Bad\}$. An example for a RockSample instance is shown in figure 4.3. The rover can select between $k + 5$ possible actions: $\mathcal{A} = \{North, South, East, West, Sample, Check_1, ..., Check_k\}$. The first four actions are for moving the rover by a single step over the grid. Taking the $Sample$ action samples the rock at the rover's current position and the rover receives a positive reward if the rock has the value $Good$ and a negative reward if the rock has the value Bad or if there is no rock. Further, agent receives a positive reward when it leaves the grid on the right side into the exit area which also ends an episode. When the rover leaves the grid on one of the other three sides, it receives a negative reward and the episode also ends. Taking a $Check_i$ action applies a long-range sensor reading to rock i with efficiency η as mentioned before and returns a noisy observation from $\{Good, Bad\}$.

We used the RockSample implementation from gym-pomdp [65] which are extensions of OpenAI Gym for POMDPs and built a wrapper on top of it because the original implementation only returns the agent's observation for sampling or checking rocks after each environment step. Within our wrapper we retrieve additional state information like the true

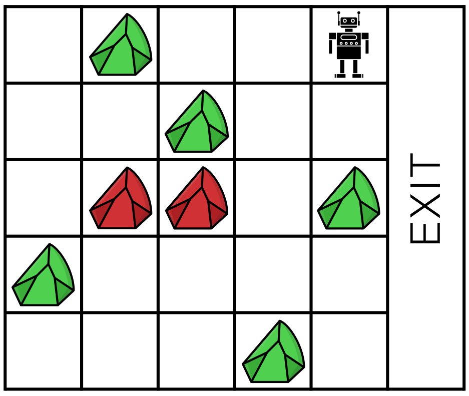

Figure 4.3.: Visualization for the partially observable RockSample problem where an agent (robot) explores an area (grid) and looks for rocks with scientific value. The agent does not know the true value of the rocks and can take noisy long-range sensor reading to check whether a rock is "good" (green) or "bad" (red). This example is for RockSample(5,7) with a grid size of 5 × 5 and 7 rocks.

rock values that will be used for fully observable samples in the GRL approach, and the position of the agent in the grid. We put all these information together along with the previous taken action and a flag that tells the algorithm whether full or partial state information are available and take it as input to build a fixed-length history in a similar way as described in section 4.1.1. We applied a one-hot encoding on the agent's position and the previous taken action because the used algorithms in our experiments could deal better with that input in the discrete action case. Further we do not use the rock positions as input for the reason that the dimensionality would be very high e.g. in RockSample(5,7) we would need $7 \cdot 5 \cdot 2 = 70$ additional dimension each history step for the one-hot encoded rock positions. Table 4.3 shows the inputs for the history in detail.

Input	Dimensionality	Comment
flag	1	indicating fully/partially observability
true rock values	k	only in fully observable case
agent (x,y) position	$2n$	one-hot encoded
previous action	$\lvert\mathcal{A}\rvert$	one-hot encoded
agent's observation	1	for sampling/checking rocks

Table 4.3.: Inputs for building a finite horizon history $\boldsymbol{h}_t$ of an instance of the RockSample(n, k) environment. The true rock values are only present in the fully observable case which is indicated by the flag, otherwise these dimensions are filled with zeros.

In general, RockSample is a relatively challenging task for model-free approaches because there is no learned model of the environment including the position of the rocks available. Therefore the agent needs much more exploration to learn advanced behavior like finding the position of the rocks, interpreting the observation from the noisy sensor readings or matching the sensor reading with the corresponding rock. For our experiments we use two different sizes of the problem: RockSample(4,4) and RockSample(5,7) where the first parameter denotes the grid size (e.g. 4x4 or 5x5) and the second one denotes the number of rocks. The easiest policy that a model-free approach can learn is leaving the grid at the exit area on the right without sampling or checking any rocks. We will show in section 4.3 that our GRL approach can help to learn a more advanced policy than this simple policy.

4.2 Experimental Setup

For answering the two research questions formulated at the beginning of this chapter, we run a series of experiments on the environments which have been introduced in the previous section. Now, we will name some of the overall conditions for our experiments. Most of the calculations for this research were conducted on the Lichtenberg high performance computer of the TU Darmstadt. Altogether, our experiments consumed about 58,000 hours of computation time. We did not use multiprocessing or GPUs for the computation, instead we used one core on Intel® Xeon® E5-2670 processors and a maximum of 6 GB RAM for each job. By applying our GRL approach, we implemented a guided version of the COPOS algorithm and SAC algorithm. From now on, we call these algorithms "COPOS-guided" and "SAC-guided". In addition to these two algorithms, we also used also the original implementation as well as the OpenAI baselines [62] implementation of the TRPO algorithm and the PPO algorithm as baseline.

We run all experiments for 50 random seeds each, except of a few computational very expensive experiments which will be mentioned at the respective position in the results section. Further, we trained the algorithms for a total number of 1 million time steps on the MuJoCo tasks, 5 million time steps on the LunarLander-POMDP and Noisy-LunarLander environments, 3 million time steps on the RockSample(4,4) environment and 10 million time steps on the RockSample(5,7) environment. The results of our experiments are plotted as learning curve for the average cumulative reward over the last 40 episodes, except of the RockSample environments where we plotted the average discounted reward with a discount factor of 0.95. In addition to these curves, we also plotted the entropy curves for all experiments.

All of the mentioned algorithms for the evaluation make use of feedforward neural networks to represent policies and/or value functions. For COPOS and TRPO the network consists of 2 hidden layers with 32 units each and tanh non-linearities. PPO uses neural networks with 2 hidden layers, 64 units for each layer and also tanh non-linearities. The SAC algorithm uses two hidden neural network layers and Rectified Linear Units (ReLU) as activation function. We had to use 256 units at each hidden layer (as in the original paper), because the performance decreased significantly in preliminary tests for a lower amount of units e.g. 32 as for COPOS. This leads to a very long computation time which is for example about 16 hours for 1 million time steps on most of the MuJoCo tasks while in contrast COPOS only needs about 20 minutes for the same task. The observation history has a fixed length of four, which means that the input for the RL algorithm includes the current observation as well as the four previous observations. For a more detailed overview on the individual algorithms, see appendix A.

4.3 Results

In this section, we will present the results of our experiments evaluated with the experimental setup, described in section 4.2. For better clarity we created separate plots for the fully observable and partially observable case of each environment. For each task, we plotted the learning curve and entropy curve for different algorithms.

4.3.1 LunarLander-POMDP

On the LunarLander-POMDP environment we compared COPOS-guided with the algorithms COPOS, PPO, SAC and TRPO for partial observablility and the comparison algorithm among themselves for fully observability as baseline. Figure 4.4 and Table 4.4 show that our GRL approach in combination with COPOS outperformed all the other algorithms in the partially observable case. It can be seen that the performance for COPOS-guided increases quickly as long as many fully observable samples are available in the first iterations. Then the performance decreases when the amount of fully observable samples also decreases. Finally the performance becomes stable and continues to increase while the amount of fully observable sample decreases. Then the algorithm is then trained only on partially observable samples. The SAC algorithm performs best for full observability which can also be seen in figure 4.4. PPO could not perform well on both the fully observable and partially observable case. Figure 4.4 also illustrates the entropy curve for all algorithms. It can be seen that TRPO suffers from premature convergence at the beginning of the training because it does not use entropy regularization. It should also be mentioned that one single run of the SAC algorithm on this problem took about 80 hours of computation time while all the other algorithms needed only between two and four hours. In addition to the results presented here, we rendered some runs of the LunarLander-POMDP (partially observable) environment with the final learned policy for COPOS-guided and COPOS. We could recognize a noticeable difference in learned behavior: With the policy learned by COPOS, the lander moves slowly into the blind area, then it is falling faster through the blind area and after that slowing down before landing, while with the policy learned by COPOS-guided, the speed of the lander is more constant also inside the blind area and the lander uses the orientation engine for moving left and right instead of using only the main engine in the blind area.

4.3.2 MuJoCo Tasks

We evaluated six algorithms, including the application of GRL as COPOS-guided and SAC-guided compared to COPOS, PPO, SAC and TRPO, on six different continuous control tasks in the MuJoCo physics simulator. We run all tasks for full observability as well as the partially observable versions we designed for the tasks. Figure 4.5 and 4.6 along with table 4.5 demonstrate that our GRL approach, meaning either COPOS-guided or SAC-guided, could outperform all the

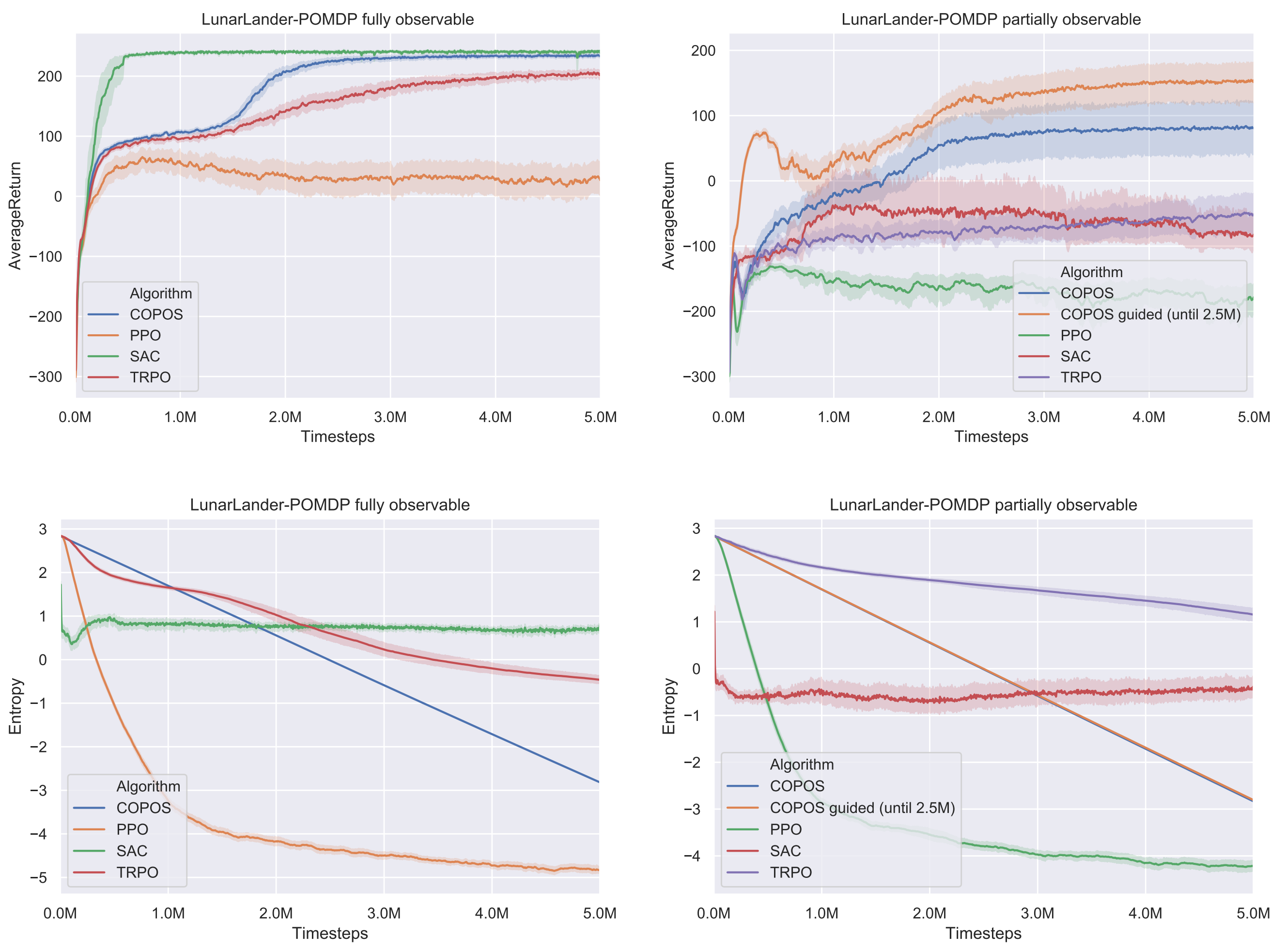

Figure 4.4.: Average return (top) and entropy (bottom) for both LunarLander-POMDP with full observations (left) and partial observations (right) over 50 random seeds except of SAC which was evaluated over 20 random seeds. Algorithms were executed for 5 million time steps. Shaded area denotes the bootstrapped 95% confidence interval.

other algorithms on all partially observable MuJoCo tasks. On the original and fully observable versions of the MuJoCo tasks, SAC could perform best on three tasks (HalfCheetah-v2, Hopper-v2 and Walker2d-v2), COPOS could perform best on two tasks (InvertedDoublePendulum-v2 and Reacher-v2) and TRPO could perform best on one task (Swimmer-v2). In the POMDP versions for the tasks, COPOS-guided could outperform all the other algorithms on four tasks (Hopper-v2, InvertedDoublePendulum-v2, Reacher-v2 and Swimmer-v2) while SAC-guided could outperform the other algorithms on two tasks (HalfCheetah-v2 and Walker2d-v2). In general, it can be seen that SAC and SAC-guided perform better on difficult tasks like HalfCheetah-v2 or Walker2d-v2 whereas COPOS and COPOS-guided can perform better on easier tasks like InvertedDoublePendulum-v2, Reacher-v2 or Swimmer-v2. A good example for this observation is the InvertedDoublePendulum-v2 task where SAC could learn a good policy at the beginning of training which would outperform the other algorithms, but the performance then decreases for more exploration. The figures 4.7 and 4.8 illustrate the entropy for all algorithms while training. COPOS drives down the entropy constantly during the training process while for SAC the entropy drops at the beginning of training and then remains relatively constant.

4.3.3 RockSample

For evaluating our GRL approach on environments with discrete action space, we compared COPOS-guided against COPOS and TRPO. We were not able to test against SAC because this algorithm is only implemented for continuous action spaces. We used two different sizes of the RockSample problem i.e. RockSample(4,4) and RockSample(5,7) for full and partial observability each. Table 4.6 and figure 4.9 demonstrate that COPOS-guided outperformed the other

Task	COPOS	COPOS-guided	PPO	SAC	TRPO
LunarLander-POMDP full	233.07±1.29	-	26.92±12.80	**238.97±1.30**	204.45±3.52
LunarLander-POMDP partial	81.03±20.29	**154.16±15.03**	-178.38±11.44	-82.33±15.10	-52.43±15.86

Table 4.4.: Mean of the average return on LunarLander-POMDP (fully and partially observable) over the last 40 episodes ± standard error over 50 random seeds except of SAC where we have 20 random seeds.

Task	COPOS	COPOS-guided	PPO
HalfCheetah-v2 full	1629.53±69.91	-	1303.73±78.72
HalfCheetah-v2 partial	1490.79±48.58	1491.02±59.99	1314.10±59.97
Hopper-v2 full	1899.34±81.80	-	1161.65±75.75
Hopper-v2 partial	1411.96±77.40	**2219.61±83.34**	994.93 ±48.56
InvertedDoublePendulum-v2 full	**8112.95±181.20**	-	3367.50±208.42
InvertedDoublePendulum-v2 partial	5528.74±297.16	**7535.25±241.52**	2629.53±133.94
Reacher-v2 full	**-4.61±0.04**	-	-6.37±0.13
Reacher-v2 partial	-9.35±0.05	**-4.69±0.05**	-9.66±0.05
Swimmer-v2 full	101.03±2.42	-	84.79±3.26
Swimmer-v2 partial	60.95±2.09	**92.63±2.96**	38.04±1.30
Walker2d-v2 full	1147.71±89.68	-	871.69±44.40
Walker2d-v2 partial	780.44±46.69	993.29±78.10	598.23±18.06

Task	SAC	SAC-guided	TRPO
HalfCheetah-v2 full	**3830.97±357.30**	-	1009.25±27.33
HalfCheetah-v2 partial	1536.54±119.12	**1835.62±154.52**	1049.87±31.06
Hopper-v2 full	**2130.49±61.89**	-	1647.12±38.42
Hopper-v2 partial	726.70±37.76	858.00±55.88	1038.55±12.95
InvertedDoublePendulum-v2 full	5164.28±266.02	-	6589.08±277.21
InvertedDoublePendulum-v2 partial	6661.90±161.32	5922.65±252.54	5089.96±190.03
Reacher-v2 full	-6.86±0.09	-	-4.74±0.05
Reacher-v2 partial	-11.06±0.20	-11.58±0.24	-9.34±0.06
Swimmer-v2 full	51.83±2.42	-	**114.95±0.73**
Swimmer-v2 partial	41.47±0.77	42.25±0.68	67.67±1.19
Walker2d-v2 full	**3740.61±78.33**	-	1331.77±49.70
Walker2d-v2 partial	1892.36±88.80	**1911.40±99.52**	862.46±27.01

Table 4.5.: Mean of the average return on six different MuJoCo tasks (fully and partially observable) over the last 40 episodes ± standard error over 50 random.

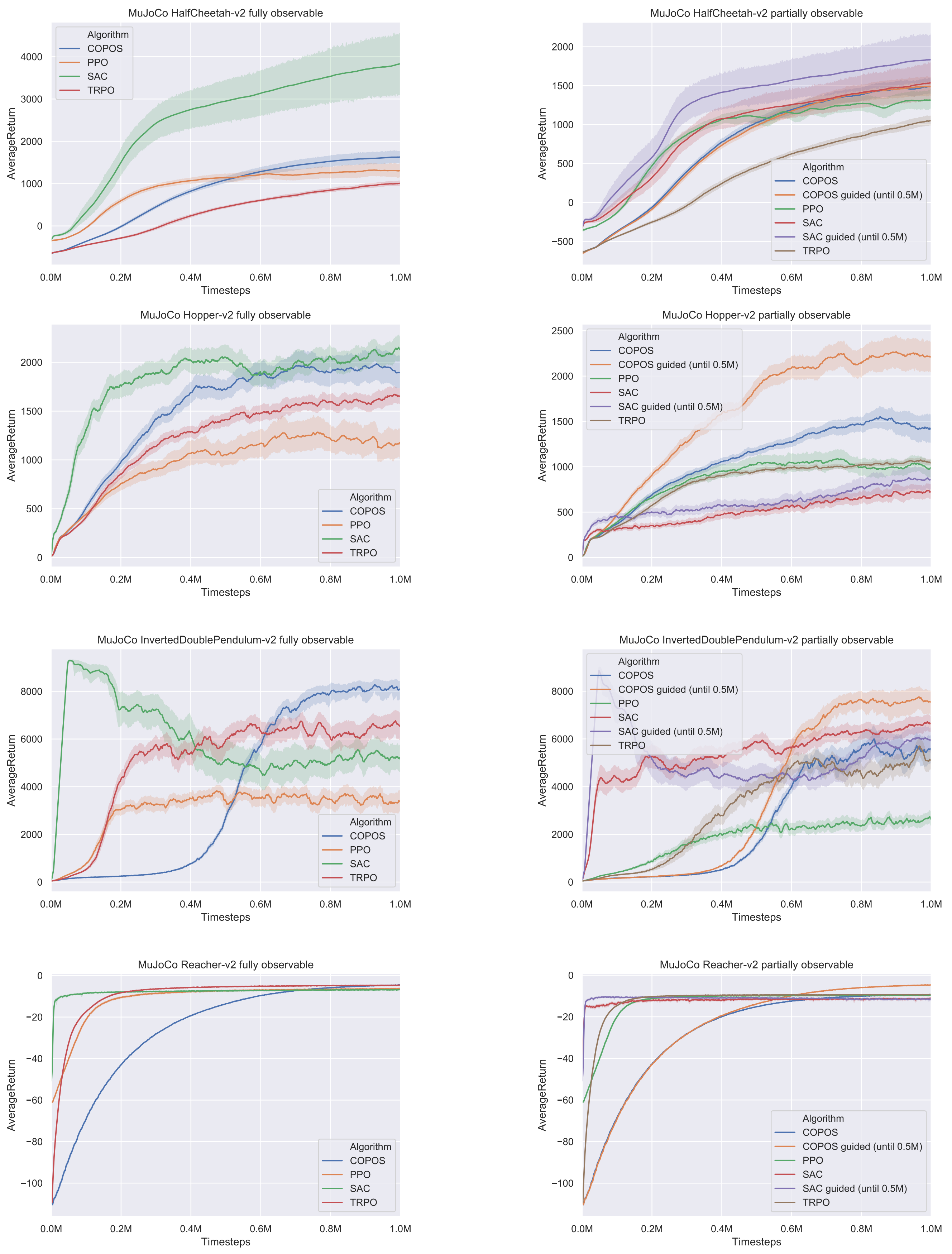

Figure 4.5.: Average return for four MuJoCo tasks (see figure 4.6 for further MuJoCo tasks), all tasks with full observations (left) and partial observations (right) over 50 random seeds. Algorithms were executed for 1 million time steps. Shaded area denotes the bootstrapped 95% confidence interval.

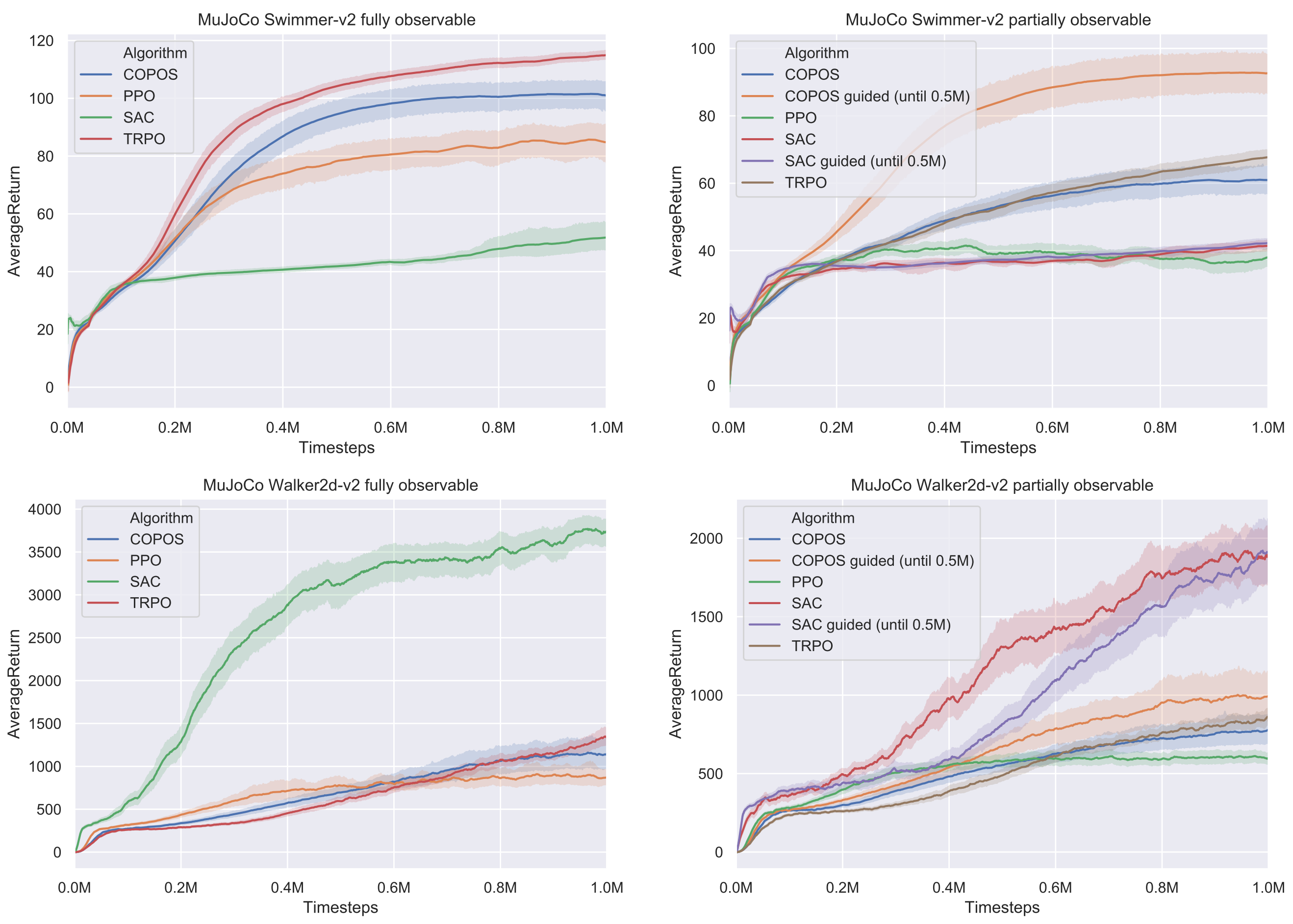

Figure 4.6.: Average return for two MuJoCo tasks (see figure 4.5 for further MuJoCo tasks), all tasks with full observations (left) and partial observations (right) over 50 random seeds. Algorithms were executed for 1 million time steps. Shaded area denotes the bootstrapped 95% confidence interval.

algorithms in both problem sizes. Our approach supports the algorithm to not being stuck in poor local optima and therefore allows to learn a more advanced policy. The COPOS brought the best performance in the fully observable case. Also on this tasks, TRPO suffers from premature convergence which can be seen in the entropy plots in figure 4.10. The entropy is falling too fast for TRPO so that the policy converges to a simple one which lets the robot leave the grid at the exit area on the right without sampling or checking any rocks. We had to run the RockSample(5,7) for more time steps because this bigger problem needed more exploration. As already mentioned before, RockSample is a relatively challenging task for model-free approaches because there is no learned model of the environment including the position of the rocks available. Therefore, model-based approaches could already achieve better results on this problem.

4.3.4 Noisy-LunarLander

On our Noisy-LunarLander environment we compare COPOS-guided against COPOS to investigate how the application of our GRL approach affects the results in comparison with training directly on noisy observations. For the Gaussian noise we use the parameters $\mu = 0$ and $\sigma = 0.05$. Figure 4.11 shows the average return as well as the entropy for both algorithms over 50 random seeds. By taking the mean of the average return ($\pm$ standard error) over the last 40 episodes we got 130.57 ($\pm$4.46) for COPOS-guided and 111.73 ($\pm$5.25) for COPOS. These results show us, that the GRL approach leads to a higher performance than using the COPOS algorithm directly on noisy data. Thodoroff et al. [66] introduced Recurrent Value Functions (RVFs) as an alternative to estimate value functions of a state by estimating the value function of the current state using the value function of past states visited along the trajectory. Their RFV approach represents also an alternative to our GRL approach for domains with noisy observations. For this reason a comparison of both approaches would be interesting for future work.

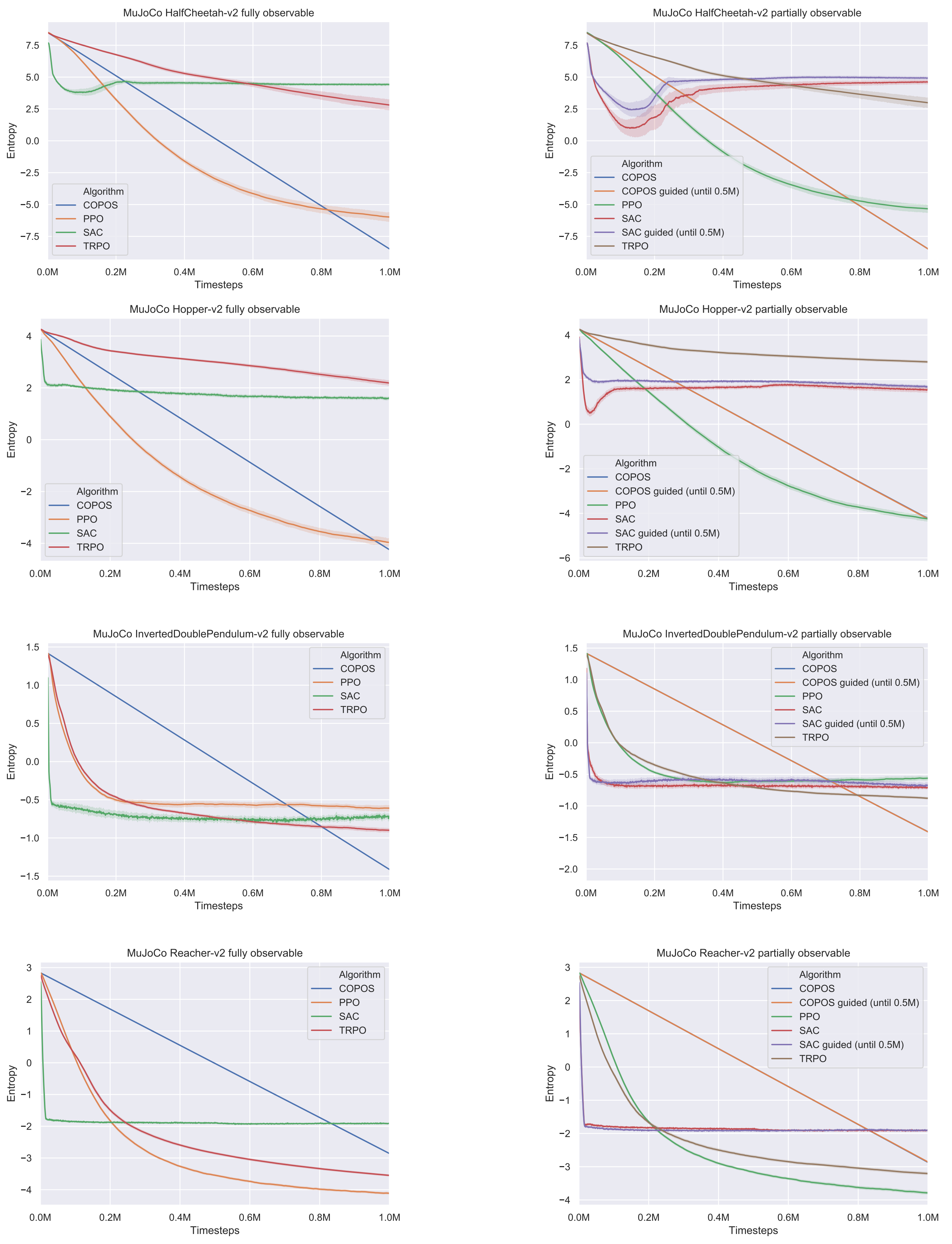

Figure 4.7.: Entropy for four MuJoCo tasks (see figure 4.8 for further MuJoCo tasks), all tasks with full observations (left) and partial observations (right) over 50 random seeds. Algorithms were executed for 1 million time steps. Shaded area denotes the bootstrapped 95% confidence interval.

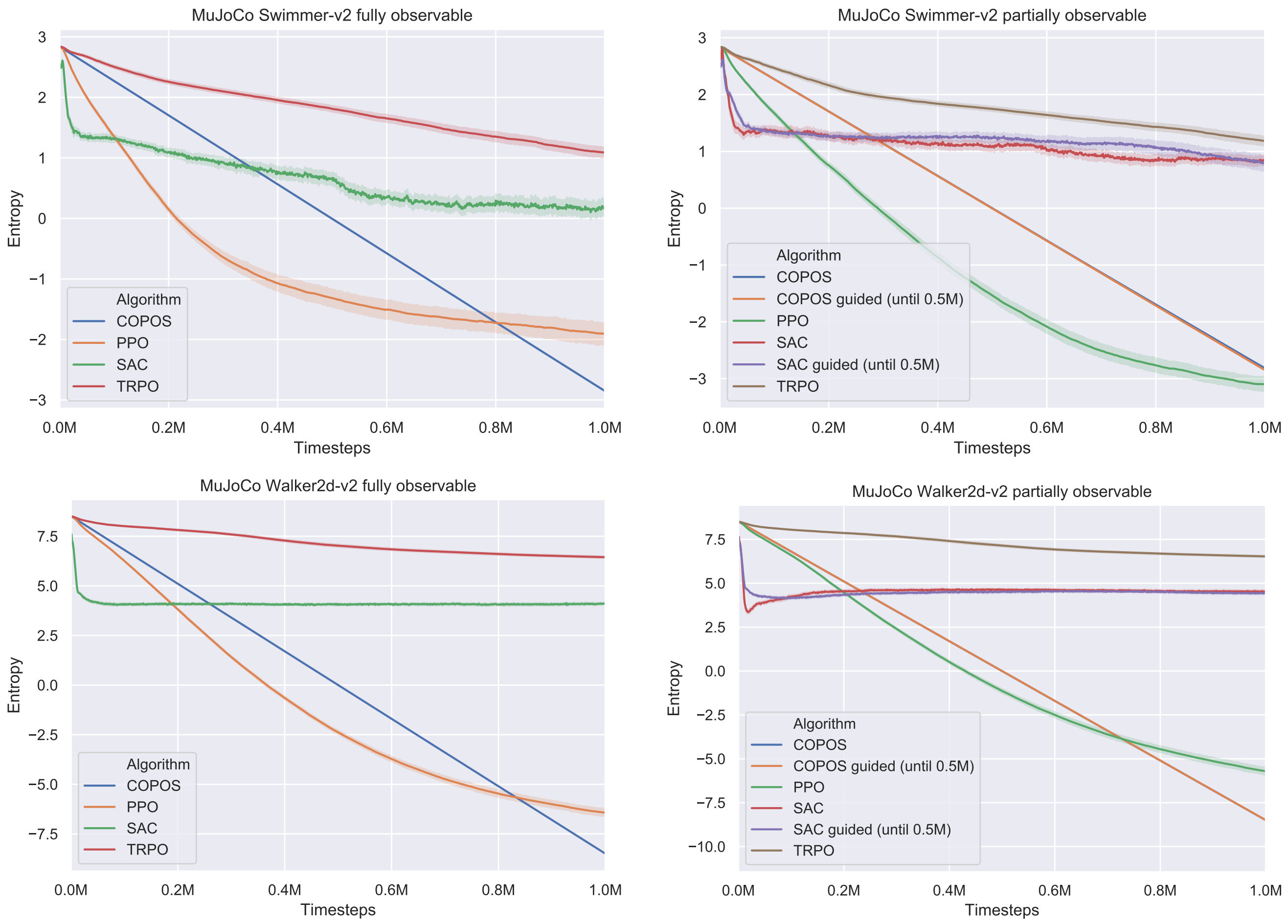

Figure 4.8.: Entropy for two MuJoCo tasks (see figure 4.7 for further MuJoCo tasks), all tasks with full observations (left) and partial observations (right) over 50 random seeds. Algorithms were executed for 1 million time steps. Shaded area denotes the bootstrapped 95% confidence interval.

Task	**COPOS**	**COPOS-guided**	**TRPO**
RockSample(4,4) full	**12.82±0.28**	-	9.03±0.20
RockSample(4,4) partial	9.95±0.28	**11.83±0.30**	8.57±0.25
RockSample(5,7) full	**13.13±0.29**	-	8.37±0.16
RockSample(5,7) partial	8.07±0.02	**9.66±0.21**	8.15±0.00

Table 4.6.: Mean of the average discounted return on two instances of RockSample (fully and partially observable) over the last 40 episodes ± standard error over 50 random seeds.

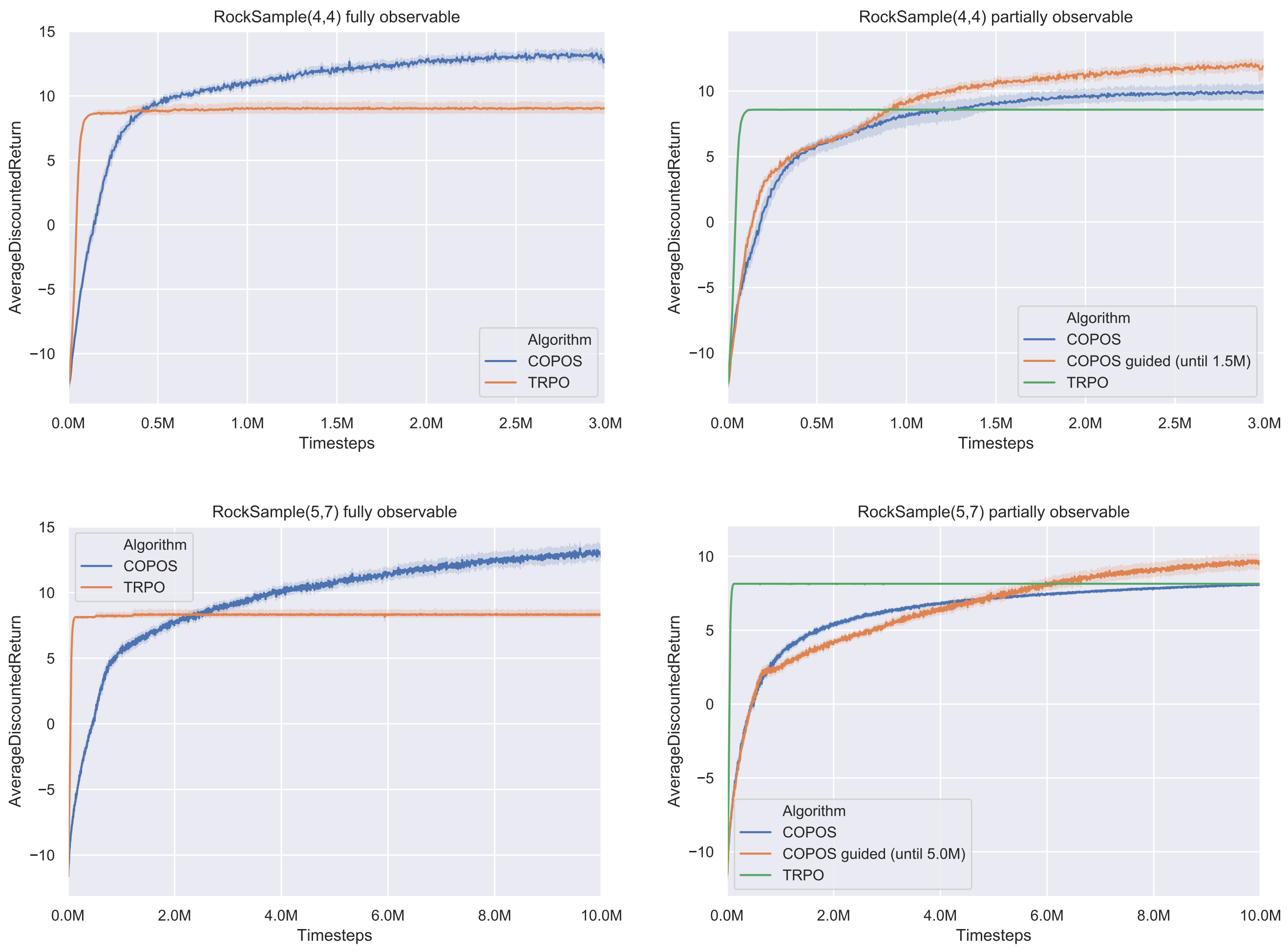

Figure 4.9.: Average discounted return for two instances of RockSample, both with full observations (left) and partial observations (right) over 50 random seeds. Algorithms were executed for 5 million time steps on RockSample(4,4) (top) and 10 million time steps on RockSample(5,7) (bottom). Shaded area denotes the bootstrapped 95% confidence interval.

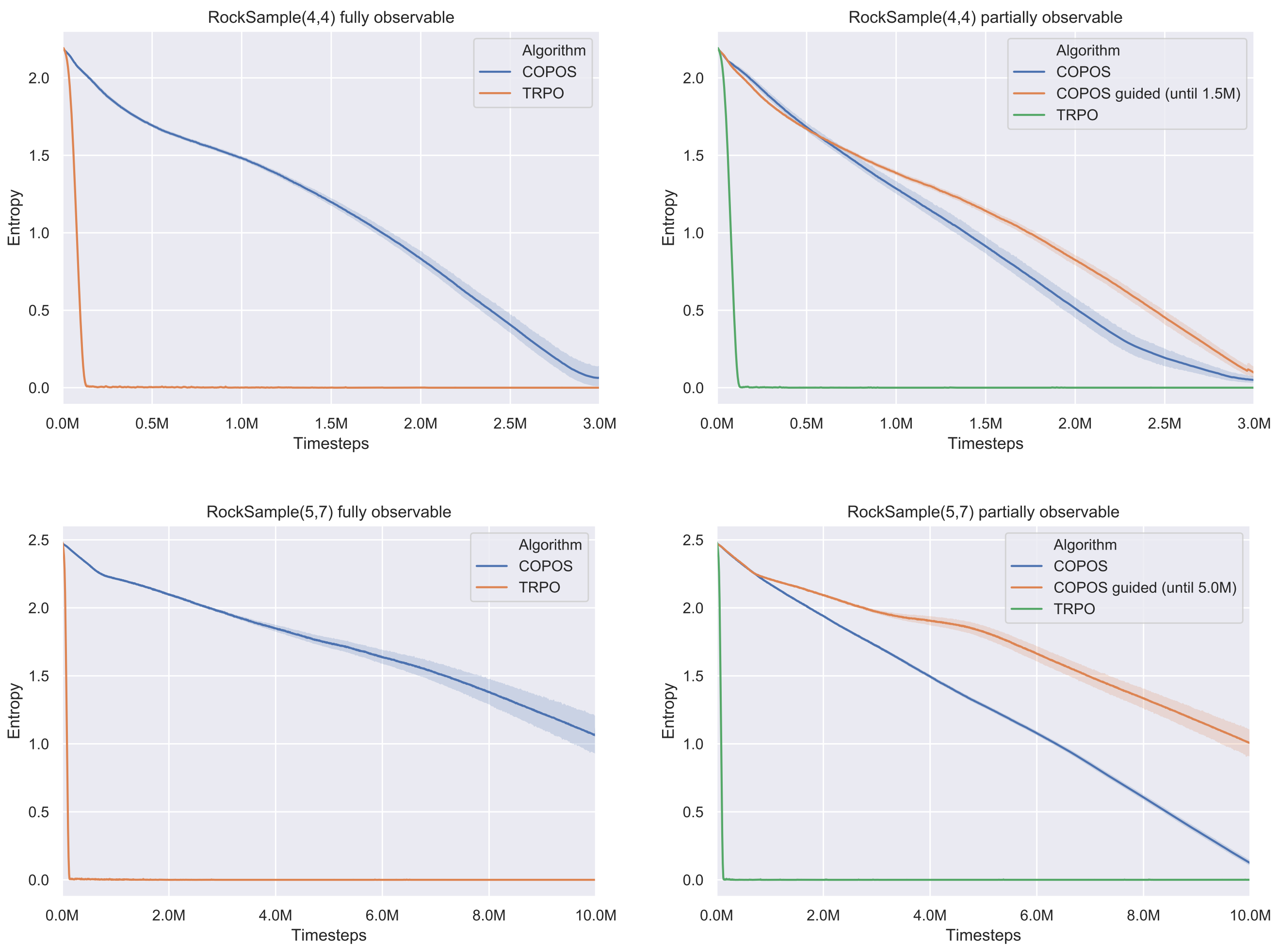

Figure 4.10.: Entropy for two instances of RockSample, both with full observations (left) and partial observations (right) over 50 random seeds. Algorithms were executed for 5 million time steps on RockSample(4,4) (top) and 10 million time steps on RockSample(5,7) (bottom). Shaded area denotes the bootstrapped 95% confidence interval.

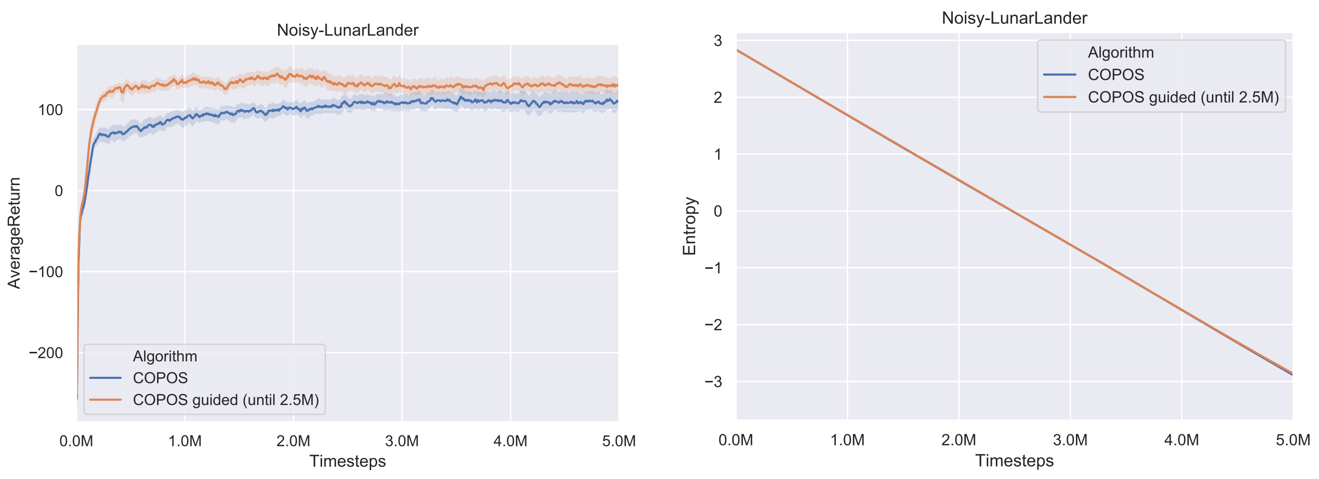

Figure 4.11.: Average return (left) and entropy (right) for Noisy-LunarLander over 50 random seeds. Algorithms were executed for 5 million time steps. Shaded area denotes the bootstrapped 95% confidence interval.

5 Conclusion and Outlook

In this work, we first provided an introduction in RL including basic concepts related with this field. We defined the MDP and POMDP framework which we need for fully and partially observable RL problems and showed up suitable methods and algorithms for solving MDPs and POMDPs. Further we summarized related approaches in literature for POMDPs and introduced the state-of-the-art deep RL algorithms, namely the policy search algorithms COPOS, PPO and TRPO as well as the actor-critic algorithm SAC.

Our main contribution is proposing the novel guided RL approach, which guides RL algorithms with additional full state information during the learning process to increase their performance solving POMDPs in the test phase. The guidance is mainly based on mixing samples containing full or partial state information while we gradually decrease the amount of full state information during training which ends up with a policy compatible with partial observations. The general formulation of our simple GRL approach allows to combine this approach with a variety of existing model-free RL algorithms as well as a variety of settings where RL algorithms can be applied to. We demonstrate the example usage with the COPOS algorithm and the SAC algorithm. Our GRL approach has not been covered yet in related work what we demonstrate by showing up related guiding approaches that have been published recently. With sufficient exploration we should be able to converge to a good policy. Further, we are able to efficiently learn behavior for parts of the problem that are actually fully observable while making learning easier for the algorithms due to using full observability at the beginning of learning. We demonstrated that our GRL approach can outperform other baseline algorithms that are trained directly on partial observations on nine different tasks. These tasks include two instances of the well-known discrete action space problem RockSample and the continuous action space problem LunarLander-POMDP which is a partially observable modification of the LunarLanderContinuous-v2 environment. Further, the benchmark tasks also include six partially observable tasks that we have constructed based on continuous control problems, simulated in the MuJoCo physics simulator. Our results show that we can converge to an even better policy for partial observability by applying our GRL when we already have a good policy for the underlaying RL algorithm. Moreover, we are able to correct low performance behavior through training with full observability at the beginning of the learning process. Also based on the LunarLanderContinuous-v2 environment we built another modification, called Noisy-LunarLander, to address the general case of bad observation quality. We compare our approach against training directly on noisy state information. On this task, we showed that our GRL approach could help to learn a policy that is more robust to uncertainty in observations.

For turning the used deep RL algorithms into memory-based approaches that are suitable for solving POMDPs, we have only used a simple windowing approach in form of a fixed-length history. In future work it would be interesting to replace this representation with a LSTM because it has been demonstrates in related work that RNNs are more powerful in partially observable domains. In addition, it would also be interesting to implement our approach to further algorithms and evaluate the performance on the used benchmark tasks, especially on the Noisy-LunarLander problem. Our approach mixes fully and partially observable samples linearly. At this point there is also potential for future work by applying more complex functions for the ratio between fully and partially observable samples at the beginning of training.

Bibliography

[1] R. S. Sutton, A. G. Barto, *et al.*, *Introduction to reinforcement learning*, vol. 2. MIT press Cambridge, 1998.

[2] M. Wiering and M. Van Otterlo, “Reinforcement learning,” *Adaptation, learning, and optimization*, vol. 12, p. 3, 2012.

[3] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, “Deepface: Closing the gap to human-level performance in face verification,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1701–1708, 2014.

[4] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, “Mastering the game of go without human knowledge,” *Nature*, vol. 550, no. 7676, p. 354, 2017.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[6] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, p. 436, 2015.

[7] B. Bakker, “Reinforcement learning with long short-term memory,” in *Advances in neural information processing systems*, pp. 1475–1482, 2002.

[8] D. Wierstra, A. Foerster, J. Peters, and J. Schmidhuber, “Solving deep memory pomdps with recurrent policy gradients,” in *International Conference on Artificial Neural Networks*, pp. 697–706, Springer, 2007.

[9] M. Hausknecht and P. Stone, “Deep recurrent q-learning for partially observable mdps,” in *2015 AAAI Fall Symposium Series*, 2015.

[10] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver, “Memory-based control with recurrent neural networks,” *arXiv preprint arXiv:1512.04455*, 2015.

[11] T. Zhang, G. Kahn, S. Levine, and P. Abbeel, “Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search,” in *2016 IEEE international conference on robotics and automation (ICRA)*, pp. 528–535, IEEE, 2016.

[12] S. Levine and V. Koltun, “Guided policy search,” in *International Conference on Machine Learning*, pp. 1–9, 2013.

[13] T. Smith and R. Simmons, “Heuristic search value iteration for pomdps,” in *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pp. 520–527, AUAI Press, 2004.

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

[15] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, IEEE, 2012.

[16] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[17] J. Kober, J. A. Bagnell, and J. Peters, “Reinforcement learning in robotics: A survey,” *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[18] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, “Planning and acting in partially observable stochastic domains,” *Artificial intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[19] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[20] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[21] C. J. C. H. Watkins, “Learning from delayed rewards,” 1989.

[22] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*, vol. 37. University of Cambridge, Department of Engineering Cambridge, England, 1994.

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

[24] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[25] J. Peters and S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural networks*, vol. 21, no. 4, pp. 682–697, 2008.

[26] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, pp. 1057–1063, 2000.

[27] E. J. Sondik, "The optimal control of partially observable markov processes.," tech. rep., Stanford Univ Calif Stanford Electronics Labs, 1971.

[28] R. D. Smallwood and E. J. Sondik, "The optimal control of partially observable markov processes over a finite horizon," *Operations research*, vol. 21, no. 5, pp. 1071–1088, 1973.

[29] M. L. Littman, "Memoryless policies: Theoretical limitations and practical results," in *From Animals to Animats 3: Proceedings of the third international conference on simulation of adaptive behavior*, vol. 3, p. 238, MIT Press, 1994.

[30] R. A. McCallum, "Instance-based utile distinctions for reinforcement learning with hidden state," in *Machine Learning Proceedings 1995*, pp. 387–395, Elsevier, 1995.

[31] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *arXiv preprint arXiv:1708.05866*, 2017.

[32] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[33] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[34] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[35] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[36] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[37] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[38] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," 2014.

[39] D. Wierstra and J. Schmidhuber, "Policy gradient critics," in *European Conference on Machine Learning*, pp. 466–477, Springer, 2007.

[40] N. Heess, G. Wayne, D. Silver, T. Lillicrap, T. Erez, and Y. Tassa, "Learning continuous control policies by stochastic value gradients," in *Advances in Neural Information Processing Systems*, pp. 2944–2952, 2015.

[41] N. Meuleau, L. Peshkin, K.-E. Kim, and L. P. Kaelbling, "Learning finite-state controllers for partially observable environments," in *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pp. 427–436, Morgan Kaufmann Publishers Inc., 1999.

[42] L. C. Baird III and A. W. Moore, "Gradient descent for general reinforcement learning," in *Advances in neural information processing systems*, pp. 968–974, 1999.

[43] J. Futoma, S. Harvard, M. C. Hughes, and F. Doshi-Velez, "Prediction-constrained pomdps," *32nd Conference on Neural Information Processing Systems (NIPS)*, 2018.

[44] Y. Bengio and P. Frasconi, "An input output hmm architecture," in *Advances in neural information processing systems*, pp. 427–434, 1995.

[45] J. Pineau, G. Gordon, S. Thrun, *et al.*, "Point-based value iteration: An anytime algorithm for pomdps," in *IJCAI*, vol. 3, pp. 1025–1032, 2003.

[46] P. S. Thomas, *Safe reinforcement learning*. PhD thesis, University of Massachusetts Libraries, 2015.

[47] S. M. Kakade, "A natural policy gradient," in *Advances in neural information processing systems*, pp. 1531–1538, 2002.

[48] S.-I. Amari, "Natural gradient works efficiently in learning," *Neural computation*, vol. 10, no. 2, pp. 251–276, 1998.

[49] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, pp. 1889–1897, 2015.

[50] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[51] J. Pajarinen, H. L. Thai, R. Akrour, J. Peters, and G. Neumann, "Compatible natural gradient policy search," *Machine Learning*, pp. 1–24, 2019.

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

[53] S. Levine and V. Koltun, "Learning complex neural network policies with trajectory optimization," in *International Conference on Machine Learning*, pp. 829–837, 2014.

[54] S. Levine, N. Wagener, and P. Abbeel, "Learning contact-rich manipulation skills with guided policy search (2015)," *arXiv preprint arXiv:1501.05611*, 2015.

[55] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.

[56] S. Levine and P. Abbeel, "Learning neural network policies with guided policy search under unknown dynamics," in *Advances in Neural Information Processing Systems*, pp. 1071–1079, 2014.

[57] M. Zhang, Z. McCarthy, C. Finn, S. Levine, and P. Abbeel, "Learning deep neural network policies with continuous memory states," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 520–527, IEEE, 2016.

[58] W. H. Montgomery and S. Levine, "Guided policy search via approximate mirror descent," in *Advances in Neural Information Processing Systems*, pp. 4008–4016, 2016.

[59] S. Carr, N. Jansen, R. Wimmer, A. C. Serban, B. Becker, and U. Topcu, "Counterexample-guided strategy improvement for pomdps using recurrent neural networks," *arXiv preprint arXiv:1903.08428*, 2019.

[60] N. Jansen, F. Corzilius, M. Volk, R. Wimmer, E. Ábrahám, J.-P. Katoen, and B. Becker, "Accelerating parametric probabilistic verification," in *International Conference on Quantitative Evaluation of Systems*, pp. 404–420, Springer, 2014.

[61] M. Hüttenrauch, A. Šošić, and G. Neumann, "Guided deep reinforcement learning for swarm systems," *arXiv preprint arXiv:1709.06011*, 2017.

[62] 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.

[63] A. Hill, A. Raffin, M. Ernestus, A. Gleave, 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.

[64] R. Zhi, “Deep reinforcement learning under uncertainty for autonomous driving,” Master’s thesis, Technische Universität Darmstadt, 2018.

[65] S. Totaro, “gym-pomdp.” `https://github.com/d3sm0/gym_pomdp`, 2018.

[66] P. Thodoroff, N. Anand, L. Caccia, D. Precup, and J. Pineau, “Recurrent value functions,” *arXiv preprint arXiv:1905.09562*, 2019.

A Experiment Parameters

Additionally to the parameters already described in section 4.2, see table A.1 for COPOS, A.2 for PPO, A.3 for SAC and A.4 for TRPO.

Parameter	**Value**
Time steps per batch	5000 (LunarLander and RockSample) or 2000 (MuJoCo)
Maximum KL-divergence (β)	0.01
Maximum difference in entropy (ϵ)	auto (continuous tasks) or 0.01 (discrete tasks)
Conjugate gradient iterations	10
Conjugate gradient damping	0.1
GAE parameters (γ and λ)	0.99 and 0.98
Value function iterations	5
Value function step size	0.001

Table A.1.: Hyperparameters for running the COPOS algorithm.

Parameter	**Value**
Time steps per batch	2048
Number of mini-batches	32
Clip range	0.2
GAE parameters (γ and λ)	0.99 and 0.95
Number of epochs	10
Learning rate	$3 \cdot 10^{-4}$

Table A.2.: Hyperparameters for running the PPO algorithm.

Parameter	**Value**
Batchsize	256
Target update interval	1
Gradient steps	1
Buffer size	$5 \cdot 10^{5}$
Learning rate	$3 \cdot 10^{-4}$

Table A.3.: Hyperparameters for running the SAC algorithm.

Parameter	Value
Time steps per batch	5000 (LunarLander and RockSample) or 2000 (MuJoCo)
Maximum KL-divergence	0.01
Conjugate gradient iterations	10
Conjugate gradient damping	0.1
GAE parameters (γ and λ)	0.99 and 0.98
Value function iterations	5
Value function step size	0.001

Table A.4.: Hyperparameters for running the TRPO algorithm.