

# Benchmarking Deep Reinforcement Learning Algorithms

**Benchmarking Deep Reinforcement Learning Algorithms**

Bachelor thesis by Felix Helfenstein

Date of submission: May 18, 2021

1. Review: Davide Tateo, Ph.D.
2. Review: Prof. Jan Peters, Ph.D.  
Darmstadt



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



---

---

## **Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt**

---

Hiermit versichere ich, Felix Helfenstein, die vorliegende Bachelorarbeit 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 gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 18. Mai 2021

---

F. Helfenstein

---

---

# Abstract

---

Deep Reinforcement Learning (RL) achieved several successes ranging from super human performances in multiple video games to difficult manipulation skills for specific robot learning tasks. In recent years, several different algorithms have been developed which now serve as baselines to compare the performance of novel work with. However, the benchmarks of these algorithms are often not carried out in a scientifically sound way, which causes the results to not be reproducible at best and highly misleading at worst. Missing hyperparameter configurations, too few independent experiment trials and the use of biased evaluation metrics all lead to inconsistent and inaccurate results which therefore are hard to use as point of comparison. This problem is enhanced by the high variance inherent to some algorithms and the stochasticity of several benchmark environments. In this thesis, we provide benchmarks for the currently most well-known model-free RL algorithms, namely A2C, TRPO, PPO, DDPG, TD3 and SAC. We evaluate these algorithms on four of the most common benchmark tasks for continuous control (Hopper, Walker, Cheetah and Ant) using the implementations of three different benchmark libraries (OpenAI Gym, PyBullet and the DeepMind Control Suite). We provide meaningful and reproducible results by averaging our outcomes over a large amount of independent trials, displaying confidence bounds, explaining our experiment setup in detail and using meaningful evaluation metrics. Specifically, we examine how the algorithm performances vary between the different tasks and libraries and also show the impact of input normalization and different network sizes.

---

---

# Zusammenfassung

---

In den letzten Jahren konnten viele Erfolge im Bereich des bestärkenden Lernens erzielt werden, die von übermenschlichen Leistungen in mehreren Videospielen bis hin zu schwierigen Manipulationsfähigkeiten für spezielle Roboterlernaufgaben reichen. Im Laufe der Zeit haben sich verschiedene Algorithmen herauskristallisiert, die nun als Vergleichsbasis für neue Arbeiten dienen. Allerdings werden die Benchmarks dieser Algorithmen oft nicht wissenschaftlich fundiert durchgeführt, was dazu führt, dass die Ergebnisse im besten Fall nicht reproduzierbar und im schlimmsten Fall sehr irreführend sind. Fehlende Hyperparameterkonfigurationen, zu wenige unabhängige Experimentierversuche und die Verwendung voreingenommener Bewertungsmetriken führen zu inkonsistenten und ungenauen Ergebnissen, die deshalb nur schwer als Vergleichspunkt verwendet werden können. Dieses Problem wird durch die hohe Varianz, die einigen Algorithmen innewohnt, und die Stochastizität einiger Benchmark-Umgebungen verstärkt.

In dieser Arbeit stellen wir Benchmarks für die derzeit bekanntesten modellfreien Algorithmen des bestärkenden Lernens, nämlich A2C, TRPO, PPO, DDPG, TD3 und SAC, bereit. Wir evaluieren diese Algorithmen auf vier der gängigsten Benchmark-Aufgaben für kontinuierliche Steuerung (Hopper, Walker, Cheetah und Ant) unter Verwendung der Implementierungen von drei verschiedenen Benchmark-Bibliotheken (OpenAI Gym, PyBullet und die DeepMind Control Suite). Wir liefern aussagekräftige und reproduzierbare Ergebnisse, indem wir unsere Resultate über eine große Anzahl von unabhängigen Versuchen mitteln, Konfidenzintervalle angeben, unseren Versuchsaufbau detailliert erklären und aussagekräftige Bewertungsmetriken verwenden. Insbesondere untersuchen wir, wie die Algorithmusleistungen zwischen den verschiedenen Aufgaben und Bibliotheken variieren und zeigen auch die Auswirkungen von Eingabenormalisierung und verschiedenen Netzwerkgrößen.

---

---

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.0.1	The importance of Benchmarking . . . . .	4
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Reinforcement Learning . . . . .	6
2.1.1	Markov Decision Process . . . . .	6
2.1.2	Model-Free vs Model-Based Reinforcement Learning . . . . .	8
2.1.3	Q-Learning and Policy Search . . . . .	9
2.1.4	On-Policy vs Off-Policy . . . . .	11
2.2	Algorithms . . . . .	12
2.2.1	A2C . . . . .	13
2.2.2	TRPO . . . . .	13
2.2.3	PPO . . . . .	14
2.2.4	DDPG . . . . .	15
2.2.5	TD3 . . . . .	16
2.2.6	SAC . . . . .	17
2.3	Environments . . . . .	19
2.3.1	Hopper . . . . .	21
2.3.2	Walker . . . . .	23
2.3.3	Cheetah . . . . .	24
2.3.4	Ant . . . . .	25
<b>3</b>	<b>Experiments</b>	<b>27</b>
3.1	Experiment Setup . . . . .	27
3.2	Hyperparameters . . . . .	30
3.2.1	Common Hyperparameters . . . . .	30
3.2.2	Specific Hyperparameters . . . . .	31

---

---

---

<b>4</b>	<b>Results</b>	<b>33</b>
4.1	General . . . . .	33
4.1.1	Hopper . . . . .	33
4.1.2	Walker . . . . .	38
4.1.3	Cheetah . . . . .	43
4.1.4	Ant . . . . .	47
4.2	Normalization . . . . .	51
4.2.1	Hopper . . . . .	51
4.2.2	Walker . . . . .	54
4.2.3	Cheetah . . . . .	57
4.2.4	Ant . . . . .	59
4.3	Network Size . . . . .	62
<b>5</b>	<b>Discussion and Outlook</b>	<b>65</b>

---

---

# Figures and Tables

---

---

## List of Figures

---

2.1	The environments used for the benchmarks. Visualizations exemplary taken from the corresponding task in the Deepmind Control Suite [1]. . . . .	22
4.1	J (discounted cumulative reward) on the Hopper environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Hopper, for A2C and PPO2 on PyBullet Hopper, and for all four on-policy algorithms on DeepMind Hopper.	35
4.2	Return (undiscounted cumulative reward) on the Hopper environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Hopper, for A2C and PPO2 on PyBullet Hopper, and for all four on-policy algorithms on DeepMind Hopper. . . . .	36
4.3	Episode length on the Hopper environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Hopper, and for A2C and PPO2 on PyBullet Hopper.	37
4.4	Value function output on the Hopper environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Hopper, for A2C and PPO2 on PyBullet Hopper, and for all four on-policy algorithms on DeepMind Hopper. . . . .	37
4.5	Entropy on the Hopper environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Hopper, for A2C and PPO2 on PyBullet Hopper, and for all four on-policy algorithms on DeepMind Hopper. . . . .	38

---



4.6	J (discounted cumulative reward) on the Walker environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Walker, for TRPO, PPO1 and PPO2 on PyBullet Walker, and for all four on-policy algorithms on DeepMind Walker. . . . .	40
4.7	Return (undiscounted cumulative reward) on the Walker environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Walker, for TRPO, PPO1 and PPO2 on PyBullet Walker, and for all four on-policy algorithms on DeepMind Walker. . . . .	40
4.8	Episode length on the Walker environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Walker, and for TRPO, PPO1 and PPO2 on PyBullet Walker.	41
4.9	Value function output on the Walker environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Walker, for TRPO, PPO1 and PPO2 on PyBullet Walker, and for all four on-policy algorithms on DeepMind Walker. . . . .	42
4.10	Entropy on the Walker environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Walker, for TRPO, PPO1 and PPO2 on PyBullet Walker, and for all four on-policy algorithms on DeepMind Walker. . . . .	43
4.11	J (discounted cumulative reward) on the Cheetah environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Cheetah, for A2C, TRPO and PPO1 on PyBullet Cheetah, and for all four on-policy algorithms on DeepMind Cheetah. . . . .	44
4.12	Return (undiscounted cumulative reward) on the Cheetah environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Cheetah, for A2C, TRPO and PPO1 on PyBullet Cheetah, and for all four on-policy algorithms on DeepMind Cheetah. . . . .	45







4.13	Value function output on the Cheetah environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Cheetah, for A2C, TRPO and PPO1 on PyBullet Cheetah, and for all four on-policy algorithms on DeepMind Cheetah. . . .	46
4.14	Entropy on the Cheetah environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Cheetah, for A2C, TRPO and PPO1 on PyBullet Cheetah, and for all four on-policy algorithms on DeepMind Cheetah. . . . .	46
4.15	J (discounted cumulative reward) on the Ant environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C on Gym Ant and for all four on-policy algorithms on DeepMind Ant. .	47
4.16	Return (undiscounted cumulative reward) on the Ant environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C on Gym Ant and for all four on-policy algorithms on DeepMind Ant.	48
4.17	Value function output on the Ant environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C on Gym Ant and for all four on-policy algorithms on DeepMind Ant. . . . .	49
4.18	Episode length on the Ant environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C on Gym Ant. . .	50
4.19	Entropy on the Ant environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C on Gym Ant and for all four on-policy algorithms on DeepMind Ant. . . . .	50
4.20	A2C comparison of normalized and non-normalized observations on the Hopper environment averaged over 25 runs with 95% confidence bounds.	52
4.21	TRPO comparison of normalized and non-normalized observations on the Hopper environment averaged over 25 runs with 95% confidence bounds.	52
4.22	PPO1 and PPO2 comparison of normalized and non-normalized observations on the Hopper environment averaged over 25 runs with 95% confidence bounds. . . . .	53
4.23	A2C comparison of normalized and non-normalized observations on the Walker environment averaged over 25 runs with 95% confidence bounds. .	54





4.24 TRPO comparison of normalized and non-normalized observations on the Walker environment averaged over 25 runs with 95% confidence bounds. . . . .	55
4.25 PPO1 and PPO2 comparison of normalized and non-normalized observations on the Walker environment averaged over 25 runs with 95% confidence bounds. . . . .	56
4.26 A2C comparison of normalized and non-normalized observations on the Cheetah environment averaged over 25 runs with 95% confidence bounds. . . . .	57
4.27 TRPO comparison of normalized and non-normalized observations on the Cheetah environment averaged over 25 runs with 95% confidence bounds. . . . .	58
4.28 PPO1 and PPO2 comparison of normalized and non-normalized observations on the Cheetah environment averaged over 25 runs with 95% confidence bounds. . . . .	58
4.29 A2C comparison of normalized and non-normalized observations on the Ant environment averaged over 25 runs with 95% confidence bounds. . . . .	60
4.30 TRPO comparison of normalized and non-normalized observations on the Ant environment averaged over 25 runs with 95% confidence bounds. . . . .	60
4.31 PPO1 and PPO2 comparison of normalized and non-normalized observations on the Ant environment averaged over 25 runs with 95% confidence bounds. . . . .	61
4.32 Comparison of small and big networks on the Hopper environment averaged over 25 runs with 95% confidence bounds. . . . .	62
4.33 Comparison of small and big networks on the Cheetah environment averaged over 25 runs with 95% confidence bounds. . . . .	63
4.34 Comparison of small and big networks on the Ant environment averaged over 25 runs with 95% confidence bounds. . . . .	64

---

## List of Tables

---

2.1 Summary of the environment specifics . . . . .	21
--	----

---



---

3.1	Algorithm specific hyperparameters . . . . .	32
-----	--	----



---

# Abbreviations

---

---

## List of Abbreviations

---

<b>Notation</b>	<b>Description</b>
DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q-Networks
MDP	Markov Decision Process
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SAC	Soft Actor-Critic
TD3	Twin Delayed Deep Deterministic Policy Gradient
TRPO	Trust Region Policy Optimization

---

# 1 Introduction

---

Reinforcement Learning (RL) is a machine learning technique for the process of making informed decisions by learning from experience. Contrary to supervised learning, though, the experience is not available prior to the learning process in form of labeled data which can be used for the training. Instead, RL agents have to create their own experience by interacting with an environment through trial and error. They do so by performing actions based on current observations of the environment state and by receiving feedback from the environment in the form of a reward signal that quantifies how good the chosen actions were. Based on this feedback, the agents try to learn behaviors that maximize the received rewards in order to reach some predefined objectives for the specific tasks they are trained to solve. The introduction of deep neural networks made it possible to process high-dimensional inputs without the need of hand-crafted feature representations, which led to big successes in domains like image classification [2] or speech recognition [3]. The concept of Deep Learning ([4, 5]) was then quickly adopted for the use in RL ([6]) leading to Deep RL. From this point on, many noticeable successes of Deep RL have been made. Starting from the super-human performance in playing Atari games based only on raw pixel observations in [7], Deep RL was also used to outperform humans in the game of Go ([8, 9]) and in the field of competitive video games ([10, 11]). In addition to that, Deep RL is regularly used in the domain of Robot Learning ([12, 13]) to solve tasks like grasping objects ([14]), various locomotion tasks ([15]) and many other complex manipulation or control tasks ([16, 17, 18]).

In recent years, many different deep RL algorithms have emerged that all have different advantages and shortcomings. Naturally, the need of benchmarks for the plethora of new approaches became bigger and bigger to be able to compare the effectiveness of different strategies and quantify their quality. The Arcade Learning Environment ([19]) was created as a platform for evaluating different algorithms on a wide range of Atari games. However, this testbed was not suitable for the domain of continuous control since the different tasks only had discrete actions. Existing benchmarks only included low-dimensional continuous control tasks like inverted pendulum, mountain car or acrobot,

---

---

which is why [20] introduced a new set of benchmark tasks including more challenging and higher-dimensional problems.

Even using those proposed benchmarks, it can be very hard to reproduce the results of deep RL algorithms, because of many sources of possible instability and variance [21]. The variance can be caused by random stochasticity inherent to the algorithms themselves or incorporated in the dynamics of the environments. This issue was investigated by [21] who reported that the results of some works could be roughly reproduced, but others were wildly varying from their findings. They suggest to average the results of many different trials and report them together with the observed standard deviations to counteract the high variance. Furthermore, they emphasize the importance of reporting all hyperparameters used for the experiments to make the results more reproducible. Both recommendations are often only partly followed (only subsets of hyperparameters, low number of trials), which is also why different works often report different baseline results [21].

Another problem that hinders meaningful comparisons between algorithm performances is the diversity of evaluation procedures and used metrics, which can lead to misleading reporting of results [22]. Specifically, some works only show a subset of their evaluations that only includes the best performing trials. Others are either only using environments for which their algorithm is especially suited, or tailor their algorithm and its hyperparameters specifically for one particular task, which is described as environment overfitting [23].

Looking at all these issues, many of the modern RL algorithms have not been properly evaluated on the proposed suite of benchmark environments for continuous control. We focus on the currently most well-known model-free RL algorithms that are often serving as baselines for novel work, namely A2C, TRPO, PPO, DDPG, TD3 as well as SAC, and provide evaluations for the most commonly used benchmarking tasks (Hopper, Walker, Cheetah, Ant) using the environment implementations from three different benchmark libraries (OpenAI Gym [24], PyBullet [25], DeepMind Control Suite [1]). Therefore, we are not only able to compare overall algorithm performances, but also investigate differences between the same task in different libraries.

We follow the before mentioned recommendations by providing the full set of hyperparameters, averaging our results over many trials, displaying confidence bounds and using several meaningful and non-biased evaluation metrics. Therefore, this work provides a summary of several algorithm performances using well carried out and much more reproducible experiments enabling direct comparisons, all in one place.

The thesis is structured as follows. Chapter 2 provides the needed background explaining the concept of RL followed by a description of the algorithms and tasks used for the

---

---

benchmarks. The setup of the experiments is layed out in Chapter 3 including the evaluation procedure as well as the used metrics and hyperparameters. In Chapter 4, we examine the experiment results by evaluating and comparing the different performances while highlighting particularities. The final Chapter 5 serves as a summary of the gained insights and concludes the thesis by adding final remarks and describing possible future works.

### **1.0.1 The importance of Benchmarking**

To motivate the procedure in this thesis we use this subsection to explain why accurate and sound benchmarking is fundamental in the field of RL.

With the introduction of several new RL algorithms in the recent years the question of how good each of the new approaches performs naturally arises. In order to quantify the quality of different techniques, benchmarks are needed. They allow comparisons between the effectiveness of different methods and make it possible to establish baseline results that can be used as a reference point for future work. Using these baselines, we can evaluate and quantify scientific progress in a more sound way. Benchmarks also make it easier to understand the effects of certain approaches by highlighting their strengths and weaknesses. We can then build on the improvements of certain techniques and do more exploration in these directions or even combine several promising approaches to create novel methods that unite the best ideas of previous work. At the same time, approaches that lead to weaker results can be neglected saving valuable research time. Furthermore, the gained insights can help identifying current limitations and areas that need improvement, which defines goals and starting points for future work. In addition to that, the existence of benchmarks increases the competition between researchers so that they feel the need to develop novel solutions and establish a culture of continuous improvement. This mindset leads to more innovation and technological progress with lower chances of stagnation in certain fields.

Benchmarks can also be used to highlight the applicability of certain approaches to different domains and can demonstrate how well the particular methods generalize to other problems. They can also measure how reliable and reproducible the results of specific algorithms are if the benchmarks are carried out properly and with a statistically significant sample size.

However, if the benchmarks are carried out poorly, the opposite effect can be achieved. As mentioned in the previous section, the experiments are often insufficiently described without a proper explanation of the evaluation process and the used hyperparameters. This lack of documentation leads to results that are not reproducible, which slows down

---

the scientific progress, because more time has to be invested to recreate the findings of other work. In addition to that, the performance for the benchmarks is often measured in different ways, which makes the reported results not comparable. Because of the high variance and instability of certain algorithms and environments, we need to report the results of a high number of independent trials to measure the consistency of the outcomes. If instead only a subset of the best results is reported, then the findings can be very misleading. The same can happen if specific environments that are especially well suited for the benchmarked algorithm are chosen as a base for the experiments or the algorithms are particularly tuned for one specific task, which makes the performance look better than it actually would be in a fair setting.

Therefore, we need standardized benchmarks and meaningful metrics that lead to consistent results which can be reproduced and used as a point of comparison for future work.



---

## 2 Background

---

### 2.1 Reinforcement Learning

---

RL describes the problem of one or more **agents** interacting with an **environment** and learning through trial and error. The environment describes the world the agent is located in and changes its state based on the behavior of the agent. The learning process is not based on labeled data like in supervised learning, but instead based on the experience the agent gains through exploring the environment by executing different **actions** and receiving feedback (**rewards**) from the environment depending on how good or bad the chosen actions were. Using this feedback the agent can learn a behavior that leads to a specific goal by repeating actions which lead to good rewards and dismissing actions which lead to bad rewards.

#### 2.1.1 Markov Decision Process

To formally describe the RL problem, we define the case of Markov Decision Processes (MDPs), which are a classical formalization of sequential decision making [26]. A decision maker called **agent** interacts with the **environment**  $\mathcal{E}$  by choosing **actions**  $a$  based on **observations**  $o$  of the current **state**  $s$ . The set of possible states (also called state space) is defined as  $\mathcal{S}$  and can have a finite or infinite amount of elements. Likewise, the set of possible actions is defined as  $\mathcal{A}$  and called action space. In the case of a finite state/ action set, the state/ action space is called *discrete*. If the states/actions correspond to continuous variables the state/ action space is called *continuous* and the amount of possible states/ actions is infinite. The environment is *fully observable* if the complete state can be seen by the agent. If the agent can only observe a part of the full state, the environment is called *partially observable*. In the literature, the symbol  $s$  is often used for observation

---

---

even though  $o$  would technically be more correct if the environment is partially observed. For simplicity, we do the same in the following sections.

The interaction between the agent and the environment is divided into different **timesteps**. At each timestep  $t$  the agent makes a decision based on the current state  $s_t$  and chooses an action  $a_t$  to take. This decision making process is based on the so called **policy**  $\pi$  of the agent. If the policy is *deterministic* - meaning that an observed state  $s$  always leads to the same action  $a$  - the policy can be described as  $\pi(s)$ . If the policy is *stochastic*, it is denoted as  $\pi(a | s) = \Pr(a_t = a | s_t = s)$  describing the probability of taking action  $a$  if the state  $s$  is observed at timestep  $t$ . A stochastic environment responds with the new state  $s_{t+1}$  based on its transition probability distribution:  $s_{t+1} \sim P(\cdot | s_t, a_t)$ . If the environment is deterministic, this distribution can be simplified to a transition function.

In addition to the new state, the environment returns a reward  $r_t$ , which is based on the reward function  $R(s_t, a_t, s_{t+1})$  and describes the short-term quality of the current state-action pair  $(s_t, a_t)$ . The overall goal of the agent is to maximize the expected cumulative reward (often called *return*) over an *episode* (also called *rollout* or *trajectory*). An episode is a sequence of states and actions starting with the initial state-action pair  $(s_0, a_0)$  and continuing with  $(s_t, a_t)$  for  $t = 1$  until  $t = T$ , where  $T$  describes the timestep when the environment gets reset to an initial state [27]. An environment is reset, i.e. an episode ends, when either a terminal state or a predefined maximum amount of timesteps is reached. Because of the fact that the chosen actions at each timestep do not only influence the immediate rewards, but also the subsequent states and therefore also future rewards, there needs to be a trade-off between short-term (immediate) and long-term reward maximization. This is done by *discounting* future rewards by a certain factor  $\gamma \in [0, 1]$ , the discount factor. Therefore, the agent tries to maximize the *expected (finite-horizon) discounted return*:

$$\mathcal{J}(\pi) = \mathbb{E}_\pi [G] = \mathbb{E}_\pi \left[ \sum_{t=0}^T \gamma^t r_t \right].$$

For non-episodic tasks, that means tasks that go on continually without a natural limit on the episode length (so  $T = \infty$ , also called *infinite-horizon*), this also solves the problem of maximizing a potentially infinite sum, because if  $\gamma < 1$  the sum has finite value as long as the reward sequence is bounded [26]. Usually, a value close to 1 is chosen for  $\gamma$  to have future rewards still be impactful for the decision making of the agent. If  $\gamma = 1$  the return is called *undiscounted*. The overall optimization problem of RL then lies in finding the *optimal policy*  $\pi^*$  that maximizes  $\mathcal{J}(\pi)$ .

To summarize, an MDP is defined by the tuple  $(\mathcal{S}, \mathcal{A}, P, R, d_0, \gamma)$  where  $\mathcal{S}$  is the set of

---

possible states,  $\mathcal{A}$  is the set of possible actions,  $P$  denotes the transition probability distribution over the next state given state-action-pairs, and  $R(s_t, a_t, s_{t+1})$  the reward function. The initial state distribution is defined by  $d_0$ , and  $\gamma \in [0, 1)$  denotes the chosen discount factor used for calculating the future discounted return [26].

### 2.1.2 Model-Free vs Model-Based Reinforcement Learning

RL algorithms can be divided into two major subcategories.

The first category consists of **model-free** RL algorithms such as A2C [28], TRPO [29] or PPO [30] (see Section 2.2 for more details). The main characteristic that distinguishes them from model-based approaches is the non-existence of a model of the environment. The agent has no explicit understanding of what the environment it interacts with looks like and, more importantly, does not know how it will change after performing an action. Accordingly, the agent typically learns either a value function for every state of the state space, forming a policy derived from these values, or directly the policy itself only by interacting with the environment and observing the received rewards and next states.

In contrast to model-free RL approaches, **model-based** algorithms rely on an available model of the environment which, if not already existing, has to be learned before applying the actual RL algorithm. By exploiting the model, the algorithm can predict the state transitions and rewards. The method then computes, similar to model-free methods, some kind of value function and eventually obtains a policy. The difference is that, instead of learning values by directly interacting with the environment, the values are updated based on the environment model only, which is called planning. This approach allows the agent to think ahead. When the model-based planning is done, the agent may interact with the environment. The experience gained in this way can be used by either resume training the model of the environment or directly improve the value function [26]. An obvious problem of model-based reinforcement learning approaches is that in most cases there is no perfect model given and learning a model often leads to suboptimal models. Planning on such a model then obviously can lead to suboptimal policies. On the other hand, a well trained model forms the basis for a good chance of obtaining an optimal policy which can also be superior to model-free approaches in planning over long time periods [31]. In addition to that, model-based algorithms often have a better sample efficiency than model-free methods since they can rely on the model instead of having to do a lot of environment interactions. This can be very beneficial, especially in real world environments like those used for robot learning tasks, where it is essential to reduce the amount of environment interactions to a minimum, not only because of the time needed

for the system to respond, but also because the mechanical parts may wear down or even break in the course of the training process resulting in increased financial expenses [12].

### 2.1.3 Q-Learning and Policy Search

In order to explain these two approaches for model-free RL, we first need to introduce the concept of **(Q)-Value Functions**, which are commonly used in both variants. These functions approximate the expected infinite-horizon discounted return, which measures the value of a state or state-action pair. The *On-Policy Value Function* is denoted as

$$V^\pi(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right]$$

representing the value of starting in state  $s$  and following a policy  $\pi$  afterwards. Similar to that, we define the *On-Policy Action-Value Function* (also *Q-Value Function*) as

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a \right],$$

which describes the value of starting in state  $s$ , taking action  $a$  and thereafter following a policy  $\pi$  [26]. We can then draw the connection from the value function to the action-value function with  $V^\pi(s) = \mathbb{E}_{a \sim \pi} [Q^\pi(s, a)]$  with  $a \sim \pi$  meaning that action  $a$  is chosen according to policy  $\pi$ . Finding the optimal policy  $\pi^*$  is often done by using the *Optimal Value Function*  $V^*(s) = \max_\pi V^\pi(s)$  or the *Optimal Action-Value Function*  $Q^*(s, a) = \max_\pi Q^\pi(s, a)$ .

The relationship between the value of a state and the value of its successor states is expressed by the **Bellman equation** for  $V$ :

$$V^\pi(s) = \mathbb{E}_{a \sim \pi, s' \sim P(\cdot | s, a)} [r(s, a) + \gamma V^\pi(s')].$$

Hereby,  $s' \sim P(\cdot | s, a)$  means that the next state  $s'$  is sampled according to the environment's transition probability distribution  $P$  [27]. Similarly, the Bellman equation can also be written for the action-value function:

$$Q^\pi(s, a) = \mathbb{E}_{s' \sim P(\cdot | s, a)} \left[ r(s, a) + \gamma \mathbb{E}_{a' \sim \pi} [Q^\pi(s', a')] \right].$$

---

Using the concept of Q-Value functions leads to the approach of **Q-Learning** [32] which is the basis of **value-based** RL methods. Here, a learned action-value function,  $Q_\phi(s, a)$ , is used to approximate the optimal action-value function  $Q^*(s, a)$ . The Bellman equation for  $Q^*$

$$Q^*(s, a) = \mathbb{E}_{s' \sim P(\cdot | s, a)} \left[ r(s, a) + \gamma \max_{a'} Q^*(s', a') \right]$$

can be used to update the estimate of  $Q^*$

$$\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha \left[ r(s, a) + \gamma \max_{a'} \hat{Q}(s', a') - \hat{Q}(s, a) \right]$$

with  $\alpha$  being a positive learning rate that determines the extent of the update step [26]. The agent can then choose the actions in the following way:

$$a(s) = \arg \max_a Q_\phi(s, a).$$

One big deficit of value-based methods is that they update their value function estimates based on other estimates, which is called *bootstrapping*. While this approach can lead to a faster learning behavior, it introduces bias and leads to the fact that value-based methods do not have general convergence guarantees if they use function approximations for  $Q^*$ . Another problem of Q-Learning methods is that the max operator for the action choosing process is very expensive in the case of big action spaces and it becomes impossible in the case of continuous action spaces. Here, the action space needs to be discretized to a finite amount of possibilities, which can limit the options of the agents and therefore is hard to do appropriately [26].

The other very popular approach to RL problems is called **Policy Search**. Here, a parameterized policy  $\pi_\theta(a | s)$  is learned directly which can then be used for the action selection process without the need for a value function. The updates for the policy are carried out by doing gradient ascent on the performance objective  $\mathcal{J}$  or some local approximation of it:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla \mathcal{J}(\theta_t)}.$$

Methods that follow this general schema are called *policy gradient methods* and belong to the class of **policy-based** methods [26]. The policy gradient can be simplified to the general form of:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\pi_\theta} \left[ \sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_t | s_t) \Phi_t \right].$$

---

---

where  $\Phi_t$  can appear in different valid forms. One of the first established forms was

$$\Phi_t = \sum_{t'=t}^T r_{t'} - b(s_t),$$

where  $b(s)$  is an arbitrary function called the *baseline*. The baseline serves as a comparison for the quality of the chosen action value and was introduced as it lowers the variance of the update without changing its expected value. This fact holds true as long as the baseline does not use bootstrapping, because that would then introduce bias. One of the first established policy gradient algorithms of this form was REINFORCE [33], which used  $\Phi_t = Q^{\pi_\theta} - b(s_t)$  with any unbiased baseline  $b(s)$ . The advantages of policy gradient methods lie in stronger convergence guarantees, a natural form of exploration by using a stochastic policy and the possibility to handle continuous action spaces without discretization.

Even when using a baseline function, policy gradient methods suffer from high variance and therefore slower convergence, because of their reliance on Monte Carlo rollouts. For this reason, an estimate of the state value function  $V^\pi(s_t)$  (also called the *Critic*) is often learned in parallel to the policy (the *Actor*) and is used as the baseline, which highly decreases the variance of the process while increasing the sample efficiency. In return, a bias is introduced, which leads to the loss of convergence guarantees. Algorithms using this approach are called **Actor-Critic** algorithms.

#### 2.1.4 On-Policy vs Off-Policy

There are two ways for RL algorithms to explore the environment in order to get new experiences that can then be used to improve the current policy. They differ in which kind of policy is used to make decisions and generate new data.

The first way creates the family of **on-policy** methods. Here, the algorithms use the same policy that they try to optimize also for the data gathering process. Therefore, on-policy methods learn action values not for the optimal policy, but for a near-optimal policy that still explores [26]. The advantage hereby lies in the fact that the data for the policy updates comes from the same state visitation distribution that is encountered when following the currently assumed best policy, which makes on-policy methods generally more stable than their counterpart. In return, on-policy methods are generally less sample

---

---

efficient since older samples can not be reused if they were not generated by the current policy.

**Off-policy** methods use a different policy for exploration than the one they are currently optimizing. The policy that is learned and that becomes the optimal policy is called the *target policy* whereas the policy to generate behavior is called the *behavior policy* [26]. The possibility to reuse data from other policies (often in the form of a replay memory, see DDPG) makes off-policy methods generally much more sample efficient. This benefit comes at the price of fewer guarantees for performance improvements, which is why off-policy methods are often considered more unstable than on-policy methods. Since the behavior policy might look entirely different than the target policy, it might lead to new experiences that are not beneficial or even disadvantageous for the improvement of the current target policy. Value-based methods often use a common policy as an  $\epsilon$ -greedy behavioral policy. Here, a random action is chosen with probability  $\epsilon$ ; otherwise the current target policy is followed [26].

State-of-the-art on-policy and off-policy algorithms try to counteract their specific deficits (sample efficiency/ instability) by several techniques, which we see in Section 2.2.

Finding a good balance between choosing optimal actions and exploring the environment to potentially find better options is not trivial, neither for off-policy nor for on-policy methods. This problem is often referred to as the *exploration-exploitation dilemma* [26]. Typical approaches are decreasing the exploration over time or letting the agent decide when to explore as it is done for stochastic policy gradient methods.

---

## 2.2 Algorithms

---

This section provides a short description of each of the used algorithms in the benchmarks. All algorithms fall into the category of *model-free* approaches and can handle continuous state and action spaces. The code base of all algorithms used for the benchmarks is built upon the Stable Baselines [34] implementations, which is an improved version of the OpenAI Baselines repository [35]. We made slight modifications to the code for evaluating the performances correctly.

---

### 2.2.1 A2C

A2C stands for Advantage Actor Critic and is a synchronous version of the A3C algorithm first introduced in [28]. It is a policy gradient algorithm that uses an on-policy value function  $V_{\phi}^{\pi}(s_t)$  (the *Critic*) that is learned during the training process as its baseline. This baseline is biased, but reduces the variance of the policy gradient estimate, which makes the training process faster and more stable.

Therefore, A2C uses the *Advantage* value

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V_{\phi}^{\pi}(s_t) = r_t + \gamma V_{\phi}^{\pi}(s_{t+1}) - V_{\phi}^{\pi}(s_t)$$

instead of just the Q-Value.

The stochastic policy  $\pi$  (*Actor*) is updated via stochastic gradient ascend with the following gradient estimator:

$$\nabla_{\theta} \mathcal{J}(\pi_{\theta}) = \mathbb{E}_{\pi_{\theta}} \left[ \sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) A^{\pi}(s_t, a_t) \right]$$

$$\theta_{k+1} = \theta_k + \alpha \nabla_{\theta} \mathcal{J}(\theta)$$

### 2.2.2 TRPO

Trust Region Policy Optimization (TRPO) was first introduced in [29]. It is an on-policy algorithm that uses a constraint on the policy updates in order to restrict the change in policy space to achieve guaranteed monotonic improvements.

Since small differences in the policy parameters can have big impacts on the performance of the policy, it is problematic to use large step sizes for policy gradient algorithms. This leads to bad sample efficiency which TRPO tries to improve by using *importance sampling*. Here, samples from rollouts of previous policy versions are reused to estimate the rewards of the current policy version. Those samples are reasonable estimates as long as the current and old version of the policy are similar enough. This is why TRPO uses a *trust region* for the policy update so that the policy update steps can not get too big. Staying inside the trust region is ensured by bounding the average KL-divergence between two subsequent policies:

$$\overline{D}_{KL}(\pi_{\theta_{old}} || \pi_{\theta}) \leq \delta$$



---

Putting this together, we get the following new objective function, which is maximized with respect to the policy parameters  $\theta$  and while satisfying the KL constraint:

$$\mathcal{J}(\theta) = \mathbb{E}_{s \sim \rho^{\theta_{old}}, a \sim \pi_{\theta_{old}}} \left[ \frac{\pi_{\theta}(a | s)}{\pi_{\theta_{old}}(a | s)} A^{\pi_{\theta_{old}}}(s, a) \right],$$

where  $\rho^{\theta_{old}}$  is the state visitation distribution for the policy  $\pi_{\theta_{old}}$ .

### 2.2.3 PPO

Proximal Policy Optimization (PPO) is an on-policy RL algorithm that was first introduced in [30]. It builds on the same ideas as TRPO (2.2.2), but does not use a hard constraint on the KL divergence between two subsequent policies. Instead, it has two other ways of ensuring the trust region.

The **clipping** variant of PPO specifies how far the new policy can diverge from the old policy by clipping the probability ratio  $r(\theta) = \frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)}$  between the old and the new policy in the range of  $[1 - \epsilon, 1 + \epsilon]$  with  $\epsilon$  being a new hyperparameter which was set to 0.2. This leads to the following objective function without any additional constraints:

$$\mathcal{J}^{\text{CLIP}}(\theta) = \mathbb{E}_{s \sim \rho^{\theta_{old}}, a \sim \pi_{\theta_{old}}} [\min(r(\theta)A^{\pi_{\theta_{old}}}(s, a), \text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon)A^{\pi_{\theta_{old}}}(s, a))]$$

The **penalty** variant of PPO adds the KL-divergence between the old and new policy as a penalty by subtracting it from the objective:

$$\mathcal{J}^{\text{PENALTY}}(\theta) = \mathbb{E}_{s \sim \rho^{\theta_{old}}, a \sim \pi_{\theta_{old}}} \left[ \frac{\pi_{\theta}(a | s)}{\pi_{\theta_{old}}(a | s)} A^{\pi_{\theta_{old}}}(s, a) \right] - \beta \overline{D}_{KL}(\pi_{\theta_{old}} || \pi_{\theta})$$

The hyperparameter  $\beta$  controls how much impact the KL penalty should have and can be adjusted during the training process.

Stable baselines provides two versions of PPO (PPO1 and PPO2) which both use the clipping variant of PPO, but differ in how multiprocessing takes place. PPO1 uses MPI [36] whereas PPO2 uses vectorized environments. We provide benchmarks for both implementations.

---

---

## 2.2.4 DDPG

Deep Deterministic Policy Gradient (DDPG) [37] is an off-policy algorithm that combines Q-Learning and Policy Search using the ideas of DPG [38] and Deep Q-Networks (DQN) [39], which is being used as its critic part.

In contrast to DQN, DDPG is suited for continuous action spaces. While DQN chooses its actions with  $a(s) = \arg \max_a Q_\phi(s, a)$ , which is only feasible for finite and low dimensional action spaces, DDPG learns an approximator for  $a(s)$ , which is denoted as  $\mu_\theta(s)$  and used as a *deterministic actor* (the policy). Since this policy is not of stochastic nature anymore as it is for most other actor-critic algorithms, DDPG needs a different form of exploration, which is the reason why noise is added to the policy at training time. This noise can come in different forms, e.g. mean-zero Gaussian noise, time-correlated Ornstein-Uhlenbeck noise ([40]) or parameter space noise ([41]).

The actor  $\mu_\theta(s)$  is being trained by trying to maximize the objective

$$\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta} [Q(s, \mu(s|\theta)|\omega)]$$

using the performance gradient, which is described by

$$\nabla_\theta \mathcal{J}(\theta) \approx \nabla_a Q(s, a|\omega)|_{s=s_t, a=\mu(s_t)} \nabla_\theta \mu(s|\theta)|_{s=s_t} \quad ,$$

where  $\omega$  denotes the parameters of the critic and  $\theta$  the parameters of the actor [37].

As it is often done in connection with DQN, a *target network*  $Q'$  is used to obtain the targets for the updates of the Q-network so that they do not depend on the same parameters that are being updated anymore, which has lead to oscillations and/or divergence [26]. The parameters  $\omega'$  of the target network  $Q'$  are close to the ones of  $Q$ , but time-delayed. Instead of just periodically copying the parameters of the main network to the target network as it is done in DQN, DDPG uses *soft target updates* in the form of

$$\omega' \leftarrow \tau\omega + (1 - \tau)\omega'$$

after each update of the main network ( $\tau$  is hyperparameter that has to be tuned, usually close to 0). This way of updating the target network leads to slower changes of the target values, which increases the stability of learning [37]. Another difference to DQN is that DDPG also uses a target network for the action prediction, the *target policy network*  $\mu'$ , which is updated in the same time-delayed way as  $Q'$ . This results in the target equation

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta')|\omega')$$

---

with target functions  $Q'$  and  $\mu'$  and their respective parameters  $\omega'$  and  $\theta'$  [37]. Using these targets for the updated Q value, we get the loss

$$\mathcal{L}(\omega) = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i | \omega))^2,$$

which is minimized with stochastic gradient descent.

$N$  is the number of transitions  $(s_i, a_i, r_i, s_{i+1})$  sampled from a so called *replay buffer*, which is used to store previous experiences so that they can be reused later for updating the policy and value function without querying the environment again. This procedure of experience replay increases the sample efficiency of DDPG while also reducing the variance of the updates, because successive updates are not correlated with one another as they would be with standard Q-Learning [26].

### 2.2.5 TD3

Twin Delayed Deep Deterministic Policy Gradient (TD3) [42] is an off-policy algorithm that builds on DDPG. It tries to minimize the effects of function approximation errors, which lead to overestimated values and suboptimal policies, because the policy tries to exploit errors in the Q-function. Three different improvements are made in comparison to DDPG that improve this issue.

The first improvement is *Target Policy Smoothing*. This regularization technique is applied to the action that is used to compute the Q target, but not when interacting with the environment. Clipped noise (from  $-c$  to  $c$ ) is added to each dimension of the action and the resulting action is then clipped to ensure that it stays in the range of valid actions, i.e.  $a_{Low} \leq a \leq a_{High}$ :

$$a'(s_{i+1}) = \text{clip}(\mu'(s_{i+1}|\theta') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}), \quad \epsilon \sim \mathcal{N}(0, \sigma).$$

This approach smoothes out the Q-function over similar actions so that the Q-function approximator does not develop incorrect narrow peaks, which are then exploited by the deterministic policy and lead to target values with high variance [42]. The second improvement is called *Double Q-Learning*, which means that TD3 concurrently learns two Q-functions,  $Q_1$  and the *twin*  $Q_2$ , and uses the one that outputs a smaller value in the target for both Q-functions:

$$y_i = r_i + \gamma \min_{k=1,2} Q'_k(s_{i+1}, a'(s_{i+1}) | \omega'_k).$$

By doing so, we get the following loss functions for  $Q_1$  and  $Q_2$ :

$$\mathcal{L}(\omega_1) = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i | \omega_1))^2,$$

$$\mathcal{L}(\omega_2) = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i | \omega_2))^2.$$

This approach is used to counteract the often reported problem of overestimation bias of the Q-functions, which is shown to lead to safer policy updates with stable learning targets [42].

The third thing that TD3 changes in comparison to DDPG is the usage of *Delayed Policy Updates*. Here, the policy is updated less frequently than the Q-functions (the critics), i.e. not at every timestep. The policy updates can then rely on a lower variance value estimate, which results in higher quality policy updates [42]. The objective is defined just as it is for DDPG, but in terms of  $Q_1$ :

$$\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta} [Q_1(s, \mu(s|\theta)|\omega)].$$

## 2.2.6 SAC

Soft Actor-Critic (SAC) ([43]) is an off-policy actor-critic RL algorithm that combines the ideas of stochastic policy optimization with the DDPG-way of learning. It uses the clipped Double Q-Learning approach just like TD3, but adds *entropy regularization* as one of its central features. Entropy can be seen as a measurement of uncertainty in the outcome of a random variable. If the outcome is very predictable, the entropy is small and the more uncertain the outcome gets the higher the entropy of the random variable is. Formally, entropy is defined as

$$H(P) = \mathbb{E}_{x \sim P} [-\log P(x)],$$

where  $x$  is a random variable and  $P$  describes its probability mass or density function.

Instead of only trying to maximize the expected cumulative reward, SAC also tries to maximize the entropy of the policy at the same time, which leads to a new definition of the objective (infinite-horizon formulation):

$$\mathcal{J}(\pi) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t (r_t + \alpha H(\pi(\cdot | s_t))) \right],$$

where  $\alpha > 0$  is an explore-exploit trade-off coefficient and  $H(\pi(\cdot | s_t))$  describes the entropy of the policy  $\pi$  at timestep  $t$ .

The Q-value function formulation is adjusted in the same way to include the entropy bonuses from every timestep but the first:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_t + \alpha \sum_{t=1}^{\infty} \gamma^t H(\pi(\cdot | s_t)) \mid s_0 = s, a_0 = a \right],$$

so that we get the connection to the value function in this way:

$$V^\pi(s) = \mathbb{E}_{a \sim \pi} [Q^\pi(s, a)] + \alpha H(\pi(\cdot | s)) = \mathbb{E}_{a \sim \pi} [Q^\pi(s, a) - \alpha \log \pi(a | s)].$$

Just as TD3, SAC uses two target Q-networks which are softly updated as explained in Section 2.2.4. SAC uses no target policy and therefore does not make use of target policy smoothing. Instead, it uses a stochastic policy, which incorporates noise because of its stochasticity, to achieve a similar effect [27].

The loss functions for  $Q_k$  look similar to the ones for TD3:

$$\mathcal{L}(\omega_k) = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i | \omega_k))^2,$$

but with a target that includes the entropy bonus:

$$y_i = r_i + \gamma \left( \min_{k=1,2} Q'_k(s_{i+1}, \tilde{a}_{i+1} | \omega'_k) - \alpha \log \pi_\theta(\tilde{a}_{i+1} | s_{i+1}) \right), \quad \tilde{a}_{i+1} \sim \pi_\theta(\cdot | s_{i+1})$$

where the next action  $\tilde{a}_{i+1}$  is sampled from the current policy and not taken from the replay buffer [27].

In order to learn the policy, a *reparameterization trick* is used. Sampling from the policy is done by computing a deterministic function which is dependant on the state, the policy parameters and independent noise  $\xi$ :

$$\tilde{a}_\theta(s, \xi) = \tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \xi), \quad \xi \sim \mathcal{N}(0, I).$$

The log standard deviations are not state-independent parameter vectors like they were for A2C, PPO and TRPO, but are instead represented as outputs from the neural network making them dependent on the specific state. In addition to that, a tanh squashing function is added to bind the actions to a finite range [27].

---

---

Using these adaptations, SAC can then maximize the objective as an expectation over noise instead of as an expectation over actions:

$$\mathcal{J}(\theta) = \mathbb{E}_{\pi_\theta} \left[ \min_{\xi \sim \mathcal{N}} \left[ \min_{k=1,2} Q_k(s, \tilde{a}_\theta(s, \xi)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s, \xi) | s) \right] \right].$$

Different to TD3, SAC uses the minimum of the two Q-function outputs instead of always using the first approximator  $Q_1$ .

The different adjustments for SAC lead to faster learning speed and a more stable learning behavior [43].

---

## 2.3 Environments

---

In this section, we provide a description of the environments used for the benchmarks. We make use of three big libraries which provide several tasks for benchmarking RL algorithms. Each of those libraries includes a set of continuous control tasks. The first library we use is *Open AI Gym* ([24]), which was one of the first libraries that provided a wide range of RL tasks. *PyBullet* ([25]) was introduced later and provided own implementations of various tasks. Our third library, the *DeepMind Control Suite* ([44]), focuses exclusively on continuous control tasks and has a unified reward structure that offers interpretable learning curves and aggregated suite-wide performance measures [44]. Both OpenAI Gym and the DeepMind Control Suite are based on the MuJoCo physics engine [45] while PyBullet has its own physics engine. Some general statements about the reward functions of each library can be made:

### Open AI Gym:

The exact reward functions look like the following:

$$r = b + v_x - w_c \cdot (\|a\|)^2 - w_f \cdot (\|f\|)^2$$

with  $b$  being a bonus for being alive (+1 for all tasks except Cheetah),  $v_x$  being the forward velocity of the robot,  $a$  being the action vector and  $f$  being the vector of raw contact forces (clipped from  $-1$  to  $1$ ). Consequently, the reward is negatively impacted by the squared euclidean norm of the action vector and the same happens for the raw contact forces. The values of the weights  $w_c$  for the control cost and  $w_f$  for the contact cost (only used for

---

Ant) are specific to the particular environment. The same is the case for the episode-end criteria of each environment.

**PyBullet:**

The PyBullet library adds two costs to the reward, which leads to the reward formulation of

$$r = b + v_x - c_e - c_j$$

with  $b$  being the bonus for being alive (+1 for each timestep,  $-1$  if the robot dies) and  $v_x$  being the forward velocity of the robot. An electricity cost  $c_e$  for using motors is calculated by

$$c_e = 2 \cdot \frac{\sum_{i=1}^N |a_i \cdot v_i|}{N} + 0.1 \cdot \frac{(\|a\|)^2}{N}$$

where  $a$  is the action vector,  $v$  is the vector of the joint velocities,  $a_i$  stands for the action for joint  $i$  and  $N$  is the total amount of joints. The joints-at-limit cost  $c_j = 0.1 \cdot n_s$  discourages the amount  $n_s$  of stuck joints. This exact reward function is used for all PyBullet environments, but the episode-end criteria are different for each.

**DeepMind Control Suite:**

The environments of the DeepMind Control Suite all have rewards in the unit interval  $r(s, a) \in [0, 1]$ . They also run for a fixed length of exactly 1000 time steps per episode without any early stopping criteria due to out-of-bounds states or bad torso tilts. Therefore, the maximum achievable cumulated undiscounted reward is 1000 for all tasks. However, this score is not achievable in practice since it takes some time for the robots to reach a state in which they can reliably get the maximum reward of 1.0 per time step [44]. The agent can get rewarded for torso height and forward velocity depending on the specific task.

Instead of providing benchmarks for all available tasks, we focus on a subset of continuous control tasks that have become the de-facto benchmark in continuous RL [44]. In particular, we use the domains of Hopper, Walker, Cheetah and Ant. These tasks are challenging, because they have high degrees of freedom and also make it easy to get stuck in local optima. Therefore, a high amount of exploration is needed to learn optimal control policies that do not only try to not let the robot fall over, e.g. by staying at the origin, but instead focus on moving forward as fast as possible [20]. The dynamics of the tasks are subject to second-order equations of motion, which means that the states are built based on position-like and velocity-like variables, while the state derivatives are acceleration-like [44]. These properties make the chosen tasks appropriate for benchmarking algorithms

Table 2.1: Summary of the environment specifics

	Env ID	Obs dim	Action dim
<b>Gym Hopper</b>	Hopper-v3	11	3
<b>Gym Walker</b>	Walker2d-v3	17	6
<b>Gym Cheetah</b>	HalfCheetah-v3	17	6
<b>Gym Ant</b>	Ant-v3	111	8
<b>PyBullet Hopper</b>	HopperBulletEnv-v0	15	3
<b>PyBullet Walker</b>	Walker2DBulletEnv-v0	22	6
<b>PyBullet Cheetah</b>	HalfCheetahBulletEnv-v0	26	6
<b>PyBullet Ant</b>	AntBulletEnv-v0	28	8
<b>DeepMind Hopper</b>	Hopper; Hop	15	4
<b>DeepMind Walker</b>	Walker; Walk	24	6
<b>DeepMind Cheetah</b>	Cheetah; Run	17	6
<b>DeepMind Ant</b>	Quadruped; Walk	78	12

in a simulated way with results that can be compared with the expected performances for real-world robotics or other physical control tasks.

The exact identifier, the observation space dimensionality and the action space dimensionality of each environment in each library can be found in Table 2.1. All environments use continuous states and actions.

### 2.3.1 Hopper

This environment consists of a two-dimensional one-legged robot (see Fig. 2.1a) which has to move forward by doing hops.

The received reward can include a bonus for being alive and is often positively impacted by the forward velocity of the robot. Negative impacts can come from stuck joints and the characteristics of the action vector depending on the specific library.



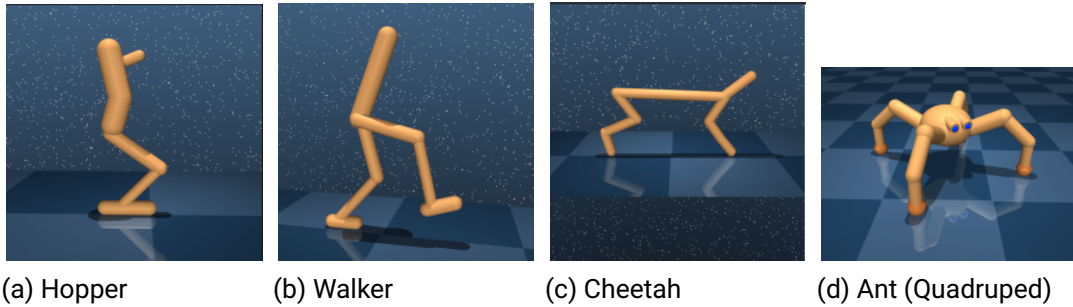


Figure 2.1: The environments used for the benchmarks. Visualizations exemplary taken from the corresponding task in the Deepmind Control Suite [1].

An episode can end when the robot’s torso is tilted too much, when an out-of-bounds state is reached or when 1000 timesteps are reached.

The dynamics of this task are very unstable, which leads to agents often getting stuck in local optima by only trying to keep the robot alive without focusing on moving forward quickly.

**Gym:**

The control cost weight  $w_c$  is set to 0.001 and  $w_f$  equals 0. The alive bonus  $b$  is 1 for each timestep.

Uniform noise from  $-0.005$  to  $+0.005$  is added to the vector of the initial state.

The episode-end criterion is reached in four different ways. The first way includes the  $z$ -coordinate of the body ( $z_{body}$ ) getting smaller than or equal to 0.7. The episode is also ended when  $|\theta| \leq 0.2$ , when the state leaves the range of  $(-100, 100)$  in any dimension or when 1000 timesteps are reached with  $\theta$  being the forward pitch of the body.

**PyBullet:**

The total amount of actuated joints  $N$  is 3, which is why also the action is of dimension 3. The initial state is  $[0. \sin \theta \cos \theta 0. 0. 0. 0. 0. j_1 v_1 j_2 v_2 j_3 v_3 0.]$  with the relative joint positions  $j_i$  that are derived by adding uniform noise from  $-0.1$  to  $+0.1$  to their initial absolute positions, the joint velocities  $v_i$  that are initially all equal to zero and  $\theta$ , which represents the robot’s initial angle to the target position.

An episode can end on three different occasions. The first criterion is reached when the sum of the body height above ground and the initial torso height (1.25 for Hopper) is smaller than or equal to 0.8. The other ways for an episode to end are either the absolute value of the body pitch being greater than or equal to 1 or the full 1000 timesteps being reached.

---

---

**DM Control:**

The reward is calculated based on the forward velocity  $v$  of the robot:

$$r = \begin{cases} \max(0, \min(\frac{v}{2}, 1)), & \text{if } 0.6 \leq z \leq 2 \\ 0, & \text{otherwise} \end{cases},$$

where  $z$  is the body height of the robot.

As for all DeepMind environments, the episodes always run for exactly 1000 timesteps.

### 2.3.2 Walker

This environment consists of a two-dimensional bipedal robot (see Fig. 2.1b) which has to walk forward as fast as possible. The task is more difficult than Hopper as it has more degrees of freedom and is even more prone to falling [20].

The received reward can include a bonus for being alive and can also be positively impacted by the forward velocity of the robot. Negative impacts can come from stuck joints, touching legs and the characteristics of the action vector depending on the specific library.

An episode can end when the robot's torso is tilted too much, when an out-of-bounds state is reached or when 1000 timesteps are reached.

**Gym:**

The control cost weight  $w_c$  is set to 0.001 and  $w_f$  equals 0. The alive bonus  $b$  is 1 for each timestep.

Uniform noise from  $-0.005$  to  $+0.005$  is added to the vector of the initial state.

The episode-end criterion is reached in three different ways. For Walker, the  $z$ -coordinate of the body ( $z_{body}$ ) is not only lower bounded, but also upper bounded. An episode ends if  $z_{body}$  leaves the range of  $(0.8, 2.0)$ , when  $|\theta| \leq 1.$ , where  $\theta$  is the forward pitch of the body, or when 1000 timesteps are reached. There are no specified state bounds that have to be fulfilled as it is the case for Hopper.

**PyBullet:**

The total amount of actuated joints  $N$  is 6, which is why also the action is of dimension 6.

The initial state is  $[0. \sin \theta \cos \theta 0. 0. 0. 0. 0. j_1 v_1 j_2 v_2 j_3 v_3 j_4 v_4 j_5 v_5 j_6 v_6 0. 0.]$  with the relative joint positions  $j_i$  that are derived by adding uniform noise from  $-0.1$  to  $+0.1$  to their initial absolute positions, the joint velocities  $v_i$  that are initially all equal to zero and  $\theta$ , which represents the robot's initial angle to the target position.

The episode-end criteria are defined in the exact same way as they are for PyBullet Hopper

---

(see Sec. 2.3.1). That means, if the sum of the body height above ground and the initial torso height (1.25 for Walker) is smaller than or equal to 0.8, the episode ends. The same happens if the absolute value of the body pitch is greater than or equal to 1 or when the full 1000 timesteps are reached.

**DM Control:**

The reward is calculated based on the forward velocity  $v$  of the robot and a factor  $s$ :

$$r = s \cdot \frac{5 \cdot \max(0, \min(v, 1)) + 1}{6},$$

where

$$s = \frac{3 \cdot \max(0, \min(1.5z - 0.8, 1)) + (0.5 + 0.5u)}{4}$$

describes the standing factor with  $z$  being the torso height of the robot and  $u$  being a projection from the z-axes of the torso to the z-axes of the world.

As for all DeepMind environments, the episodes always run for exactly 1000 timesteps.

### 2.3.3 Cheetah

This environment consists of a two-dimensional bipedal cheetah robot (see Fig. 2.1c) which has to run forward as fast as possible.

The dynamics of this task are much more stable compared to tasks like Hopper, which means that failures (abrupt episode endings) occur less often and with less variance [21].

The reward is build differently depending on the specific library.

An episode only ends if the full 1000 timesteps are reached.

**Gym:**

The control cost weight  $w_c$  is higher for this environment (0.1) and  $w_f$  equals 0. Contrary to Hopper and Walker, the received reward includes no bonus for being alive ( $b = 0$ ) so all focus is laid on the robot's forward velocity.

For Cheetah, more uniform noise (from  $-0.1$  to  $+0.1$ ) than for Hopper and Walker is added to the vector of the initial state.

**PyBullet:**

The total amount of joints  $N$  is 6, which is why also the action is of dimension 6.

The initial state is  $[0, \sin \theta, \cos \theta, 0, 0, 0, 0, 0, j_1, v_1, j_2, v_2, j_3, v_3, j_4, v_4, j_5, v_5, j_6, v_6, 0, 0, 0, 0, 0, 0]$  with the relative joint positions  $j_i$  that are derived by adding uniform noise from  $-0.1$  to  $+0.1$  to

---

---

their initial absolute positions, the joint velocities  $v_i$  that are initially all equal to zero and  $\theta$ , which represents the robot’s initial angle to the target position.

The alive bonus  $b$  is 1 if both the absolute body pitch is smaller than 1.0 and also no body part other than the feet touches the ground or another body part. If that is not the case, the episode is not ended, but the alive bonus becomes  $-1$  instead.

**DM Control:**

The reward is calculated based on the forward velocity  $v$  of the robot:

$$r = \max(0, \min(\frac{v}{10}, 1)),$$

which means that  $r$  is linearly proportional to  $v$  up to a maximum of 10m/s [1].

As for all DeepMind environments, the episodes always run for exactly 1000 timesteps.

### 2.3.4 Ant

This environment consists of a three-dimensional quadruped robot (see Fig. 2.1d) which has to walk forward as fast as possible. The third dimension and therefore bigger observation and action spaces make this environment the hardest to learn of the four described environments, in theory.

The received reward can include a bonus for being alive and can also be positively impacted by the forward velocity of the robot. Negative impacts can come from stuck joints and the characteristics of the action vector depending on the specific library.

An episode can be ended when the robot’s torso height gets too low or when 1000 timesteps are reached.

**Gym:**

The control cost weight  $w_c$  is set to the highest value of all environments with 0.5. Also, Ant is the only environment where  $w_f$  is not zero, but 0.0005 instead.

Just like for the Cheetah environment, the Ant environment makes use of the higher uniform noise of  $-0.1$  to  $+0.1$ , which is added to the vector of the initial state.

The episode-end criterion is reached in only two ways. The  $z$ -coordinate of the body ( $z_{body}$ ) has to stay in the range of  $(0.2, 1.0)$ . If that is not the case or 1000 timesteps have been reached, the episode is ended.

**PyBullet:**

The total amount of joints  $N$  is 8, which is why also the action is of dimension 8.

The initial state is  $[0. \sin \theta \cos \theta 0. 0. 0. 0. 0. j_1 v_1 j_2 v_2 j_3 v_3 j_4 v_4 j_5 v_5 j_6 v_6 j_7 v_7 j_8 v_8 0. 0. 0. 0.]$  with

---

---

the relative joint positions  $j_i$  that are derived by adding uniform noise from  $-0.1$  to  $+0.1$  to their initial absolute positions, the joint velocities  $v_i$  that are initially all equal to zero and  $\theta$ , which represents the robot's initial angle to the target position.

For Ant, there is no body pitch to be considered so the episode is ended when the central sphere scrapes the ground, which is expressed by the sum of the body height above ground and the initial torso height (0.75 for Ant) becoming smaller than or equal to 0.26. An episode is also ended when 1000 timesteps are reached.

**DM Control:**

The reward is calculated based on the forward velocity  $v$  of the robot and a factor that is proportional to how upright the torso is:

$$r = (0.5u + 0.5) \cdot \max(0, \min(v + 0.5, 1)),$$

where  $u$  is a projection from the z-axes of the torso to the z-axes of the world.

As for all DeepMind environments, the episodes always run for exactly 1000 timesteps.

---

## 3 Experiments

---

In this chapter, we describe the experiments that are carried out including the setup, the evaluation metrics and also the chosen hyperparameters.

---

### 3.1 Experiment Setup

---

**Experiments:** We make use of all algorithms described in Section 2.2 (A2C, TRPO, PPO 1 and 2, DDPG, TD3, SAC) building on the implementations of Stable Baselines [34]. All algorithms are evaluated on the four tasks described in Section 2.3, namely Hopper, Walker, Cheetah and Ant, to cover a diverse range of continuous control tasks with different dynamics. Moreover, we include different variations of the same task by using three different libraries that all provide their own specific environment implementations (*Open AI Gym* [24], *DeepMind Control Suite* [44] and *PyBullet* [25]). Consequently, we cannot only investigate how the algorithm performances differ from task to task, but also which impact different environment implementations of the same task have for the evaluations in order to give a more insightful overview. This procedure is important, because parameter tuning may overfit on a single environment and a better performance in one environment does not mean that the algorithm is more suited or that the parametrization is reasonable. In addition to a general overview about the algorithm performances on the different environments (Section 4.1), we investigate the impact of input normalization for the algorithms A2C, TRPO and both versions of PPO (Section 4.2) as well as compare the results between small and big network architectures for the algorithms DDPG, TD3 and SAC (Section 4.3).

The python requirements look like the following: *python* 3.7.4, *tensorflow-gpu* 1.15.0, *stable-baselines* 2.10.0, *gym* 0.17.2, *pybullet* 3.0.4 and *dm-control* 0.0.322773188.

**Training and evaluation process:** During the learning process, the performance of RL algorithms is not very stable, which is why it is not recommended to track a running

---

average performance based on the most recent training samples as this might average the performance across several different policy updates and therefore lead to misleading and highly fluctuating results. In addition to that, exploratory behavior cannot be factored out so that, even if the policy was fixed, it would not be possible to distinguish exploratory, noisy actions from greedy actions sampled on-policy, which means we are not actually evaluating the behavior the agent currently considers as best. In order to circumvent these issues, we perform our evaluations periodically after batches of update steps (one epoch) have been carried out. After each of those epochs we do one evaluation phase for which we hold the current policy fixed and predict actions in greedy fashion. One epoch includes  $2^{15} = 32768$  training steps and the evaluation is carried out over  $2^{14} = 16384$  episode steps in order to derive a meaningful average performance after each training epoch.

Recent work has shown that the variance in benchmarking results due to environment randomness or stochasticity in the learning process is a major concern for deep RL [22]. That means that the randomness in the initial model weights, the randomness in the exploration and also in the dynamics of the environments causes drastic differences in the actual outcome of each run. They also show that even averaging the results over too few runs can lead to misleading results by splitting 10 different runs into two sets of 5. The variance between runs is enough to create statistically different distributions for the two sets just from varying random seeds [22]. What makes this problem even worse is that often only the top- $N$  runs of several are selected to show seemingly good results which may only be caused by lucky random seeds (e.g. [46] and [28]).

That is why we average our results for each experiment over 25 different independent runs. To visualize the variance across the runs, we also report the 95% confidence intervals of our results,

$$\left(\bar{x} - t_{n-1} \cdot \frac{s}{\sqrt{n}}, \bar{x} + t_{n-1} \cdot \frac{s}{\sqrt{n}}\right),$$

with  $n$  describing the sample size,  $\bar{x} = 1/n \cdot \sum_{i=1}^n x_i$  describing the sample mean,  $s$  describing the sample standard deviation and  $t_{n-1}$  being the critical t-value from the t-distribution with  $n - 1$  degrees of freedom [47]. So we can say that the true mean of each metric lies within the confidence bounds for 95% of all experiments carried out in this fashion.

**Evaluation metrics:** In the literature, the algorithm evaluations are often reported with highly biased performance measures which lead to misleading results. One of such measures is the *Maximum Return*, which is considered to be unsuitable, because single outlying trials may yield a vastly larger maximum return than what can usually be achieved,

---

---

especially for high-variance policies and environments [21]. The *Maximum Average Return* is better in that regard, but still has the same problem of high performing sets of trials being cherry picked for good results.

For those reasons, we track the **Average Undiscounted Return**

$$R = \frac{1}{n} \sum_{i=1}^n R_i = \frac{1}{n} \sum_{i=1}^n \sum_{t=0}^T r_t$$

over all  $n$  episodes that fit in the 16384 evaluation steps as it is suggested in [22].

In addition to that, we depict the **Average Discounted Return**

$$J = \frac{1}{n} \sum_{i=1}^n G_i = \frac{1}{n} \sum_{i=1}^n \sum_{t=0}^T \gamma^t r_t$$

calculated in the same fashion. This metric is not shown very frequently in related work even though it represents the *expected discounted return*, which is the value that is approximated by the value functions.

Speaking of which, we also keep track of the **(Q-)Value Function Output** averaged over all initial states of the evaluation episodes. By doing so, we have a measure of the *expected discounted return* at the start of an episode. We can then compare this value with the *actual* discounted return  $J$  to examine how accurately the value function approximation works. In the case of TD3 and SAC we take the minimum value of the two Q-value functions.

Another metric that is documented is the **Average Episode Length**. The benchmark environments often include a bonus reward for staying alive at each timestep so that just trying to keep the episode running forms a local optimum for the behavior of the agents, which is why it makes sense to investigate how the average episode lengths change with more and more epochs. We exclude this metric for environments with fixed episode horizons that are not dependent on the policy (e.g. Cheetah Gym).

The final metric that is kept track of is the **Entropy** of the policies of A2C, TRPO, PPO1 and 2 as well as SAC. This metric measures the uncertainty of the action choice of the current policy and can be used to examine the exploratory behavior over the course of the training process.

For all five described metrics we take the arithmetic mean over all 25 runs and then calculate the 95% confidence intervals as described in the paragraph above. The figures



---

---

use a different color for each evaluated algorithm and depict the confidence bounds by using a transparent version of the respective color.

---

## 3.2 Hyperparameters

---

In this section, we provide the hyperparameter values used for the benchmarks. In order to create reproducible results we include all customized hyperparameters as suggested in [22]. All unmentioned hyperparameters are left at the default values of the algorithm’s Stable Baselines implementation (release 2.10.0). The hyperparameters are manually tuned on the *Gym Hopper* environment using different sources ([48], [20], [30]) as starting values for the tuning process to produce comparable results. The found hyperparameter settings are then used for all tasks of all libraries in order to investigate the robustness of hyperparameters across different environments.

### 3.2.1 Common Hyperparameters

**Network Architecture:** We use feed-forward neural networks with identical network sizes as approximators for the policy and value function. There are *no shared layers* between them so that they are independent from each other. This approach makes sure that changes in the policy are not affecting the value function and, therefore, also the gradients are kept untouched, which results in a mathematically more correct way of approximating the policy and value functions.

The *networks sizes* are the same for A2C, TRPO as well as PPO1 and 2 with two hidden layers of 32 units. DDPG, TD3 and SAC are shown to perform better with bigger networks [21], which is why we use two hidden layers with 400 and 300 hidden units for the general experiments (Sec. 4.1). In Section 4.3, we explore the difference between the performance of small and big networks for those algorithms.

ReLU [49] is chosen as our *activation function* throughout all experiments, which is commonly used in the literature and is shown to perform best across environments and algorithms [22].

**Discount Factor:** Our *discount factor*  $\gamma$  is set to 0.99 for all experiments. This is a commonly used value which ensures that the long term effects are properly taken into account when optimizing the policy.

---

---

**Buffer Size:** The replay memory for DDPG, TD3 and SAC can contain 500000 samples, which is a size that is big enough to contain a diverse range of experiences while still fitting into the memory.

**Noise type:** We use Gaussian action space noise with mean 0 and a standard deviation of  $\sigma = 0.2$  for DDPG and  $\sigma = 0.1$  for TD3. Parameter space noise is not used, but might improve the results for some environments as described in [41], because it is state-dependant and, thus, can adjust the exploratory behavior specifically based on the current state.

**Reward Scale:** This hyperparameter can be used to potentially improve the stability of the training process by multiplying the gained rewards by a factor  $RS$ . We use no reward scaling ( $RS = 1$ ), because the results of this hyperparameter are reported to be inconsistent between different environments and scaling values [22]. However, some related works found a value of 0.1 to produce the best overall results [20], [50].

**Input Normalization:** We test the impact of normalized observations in Section 4.2 for the algorithms A2C, TRPO as well as PPO1 and 2 by using the VecNormalize wrapper of Stable Baselines. It will be specified whether those algorithms used normalized inputs in the specific experiments or not. For the remaining algorithms we do not normalize the inputs with the exception of DDPG.

### 3.2.2 Specific Hyperparameters

The hyperparameters that are different for each of the benchmarked algorithms can be found in Table 3.1. We also include hyperparameters that are exclusive for some algorithms and label these fields as "-" for algorithms that do not use this hyperparameter.

Table 3.1: Algorithm specific hyperparameters

	<b>A2C</b>	<b>TRPO</b>	<b>PPO</b>	<b>DDPG</b>	<b>TD3</b>	<b>SAC</b>
<b>Batch Size</b>	32	2048	2048	64	128	256
<b>Actor LR</b>	0.001	0.01 (max KL)	0.0003	0.0001	0.001	0.0003
<b>Critic LR</b>	0.001	0.00025	0.0003	0.001	0.001	0.0003
<b>Ent Coef</b>	0.0	0.0	0.0	-	-	'auto'
<b>VF Coef</b>	0.5	-	0.5	-	-	-
<b>tau</b>	-	-	-	0.001	0.005	0.01
<b>vf_iters</b>	-	5	-	-	-	-
<b>noptepochs</b>	-	-	10	-	-	-
<b>lr_schedule</b>	'linear'	-	-	-	-	-
<b>GAE <math>\lambda</math></b>	-	0.98	0.95	-	-	-
<b>nminibatches</b>	-	-	64	-	-	-
<b>cliprange</b>	-	-	0.2	-	-	-
<b>learning_starts</b>	-	-	-	-	8192	10000
<b>train_freq</b>	-	-	-	-	1024	1
<b>gradient_steps</b>	-	-	-	-	1024	1

---

## 4 Results

---

This chapter shows and describes the outcomes of the experiments mentioned in Chapter 3. We compare the results of the different algorithms with results that can be found in related work and also show how the algorithm performance varies depending on the chosen environment and library. We first show the results of the settings found to be best for each individual task and investigate the impact of input normalization and network size in the later sections.

---

### 4.1 General

---

This section includes the results for the best performing settings within the selected environments. We show the progress of the learning process in terms of the undiscounted cumulative reward  $R$ , the discounted cumulative reward  $J$ , the value function output  $V$ , the entropy of the policy and the episode length when it is meaningful. All experiments are run for 100 epochs, which corresponds to 3276800 total training steps.

#### 4.1.1 Hopper

Looking at the algorithm performances on the **OpenAI Gym** Hopper environment we can clearly see that SAC is the best performing algorithm in terms of the *discounted cumulative reward*  $J$  (see Figure 4.1). It not only has the fastest learning behavior, but also the highest peak performance at the later epochs. Furthermore, the SAC learning behavior can be described as very stable because of the low margin of error throughout the learning process over the 25 different runs. The same can be said for TRPO, PPO1 and PPO2 on Hopper Gym, but those algorithms reach a lower peak performance and also converge a bit slower (especially PPO2). TD3 reaches a similar performance while having a somewhat more unstable learning process leading to higher error margins. This instability is even

---

---

more visible for A2C and DDPG. In addition to that, those two algorithms also perform considerably worse in terms of average  $J$ . This instable behavior of DDPG is also reported in [21] and explained by the high variance of DDPG itself, but also by the high stochasticity of the Hopper environment. We later show that the performance of DDPG is indeed a lot better on other tasks, e.g. Cheetah (see Section 4.1.3). In [20] they improve the stability of DDPG by scaling the reward by a factor of 0.1 which is not tested here. Different reward scaling factors are benchmarked in [22], who show that this approach can indeed have a big impact on the algorithm performance. However, the results are inconsistent between different environments and scaling factors, which is why a more principled approach like the adaptive reward target rescaling in [51] is suggested.

The benchmarks on the **PyBullet** Hopper environment look similar in terms of overall algorithm rankings when looking at  $J$  with SAC again performing the best. TD3 scores a bit better on the PyBullet variant of Hopper reaching values nearly as good as SAC and outperforming the on-policy algorithms TRPO, PPO1 and PPO2 by a decent margin. Especially TRPO achieves results that are considerably worse comparatively reaching only 81% of the final SAC performance (cf. 91% on Gym Hopper). These results show that even on the same task the algorithm performances can differ significantly depending on the chosen library and its exact implementation of the environment. Another discrepancy that can be observed for all algorithms on PyBullet Hopper is expressed by the size of error margins. They are significantly larger than on Hopper Gym, which means that the individual random seed has a very big impact on the algorithm performance - even with the same hyperparameters. This observation is also made by [22] and is the reason why benchmarks should always average the results over a sufficient number of independent runs to ensure that the reported scores are not only based on lucky random seeds. The initial learning behavior of DDPG looks a bit faster on PyBullet Hopper than it does on Gym Hopper, but it still cannot reach a good performance after 100 epochs and neither can A2C.

The evaluation of **DeepMind Hopper** paints a completely different picture than what can be observed for the other two libraries. SAC is the only algorithm that shows a noticeable performance increase, but still can not reach very high  $J$  values. All other algorithms can not learn a behavior that notably increases the discounted return, which emphasizes how hard the DeepMind variant of Hopper is compared to its counterparts.

The fact that several evaluation metrics should be used to describe the performance of an algorithm can be seen in Figure 4.2. The results in terms of the *undiscounted cumulative reward*  $R$  do not match the picture that emerges when only looking at  $J$  (Fig. 4.1). On Gym Hopper, TD3 appears to be the best performing algorithm. The large confidence bounds show that it is even more important to perform enough runs when comparing

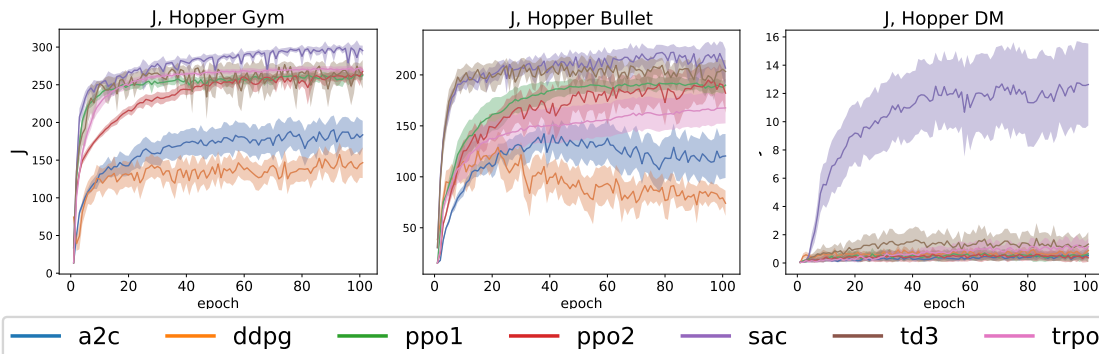


Figure 4.1:  $J$  (discounted cumulative reward) on the Hopper environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Hopper, for A2C and PPO2 on PyBullet Hopper, and for all four on-policy algorithms on DeepMind Hopper.

benchmarks in terms of  $R$ . SAC shows a very fast initial learning behavior (as is the case when looking at the  $J$  curve), but then seemingly drops in performance and ends up with the third worst final undiscounted return at the 100 epoch mark. This behavior can be explained by looking at the course of the episode lengths during the training process. The plots in Figure 4.3 are closely related to the progress in terms of  $R$ . The reason for that is that the Hopper environment adds a bonus for being alive to the reward at each timestep so that learning a policy that focuses on keeping the robot alive at each timestep and therefore maximizes the average episode length automatically also increases  $R$ . SAC, however, in this case, learns a policy that focuses on maximizing the immediate (short-term) reward at each timestep (e.g. by trying to reach a very high forward velocity), which leads to shorter episodes, because out-of-bounds states are reached faster, and, consequently, less undiscounted cumulative reward. This discrepancy can not be observed for SAC on the PyBullet Hopper environment where the algorithm leads in performance for both  $J$  and  $R$  so the exact reward function of the specific environment implementation plays a large role in how the evaluation of different agents looks. Looking at the DeepMind variant of Hopper in terms of  $R$  adds proves this point. Apart from SAC, no algorithm can achieve notable return values as it is the case for  $J$ , which can be explained by the missing bonus for just staying alive. The policy is only rewarded for forward velocity for the DeepMind implementation so that only real hopping behaviors increase the gained reward.

Comparing the other algorithms learning curves in terms of  $R$  as opposed to  $J$  leads to

roughly the same results with some notable exceptions. On the Gym Hopper environment, the PPO2 performance appears worse than the performance of PPO1 and TRPO although it is roughly on par in terms of  $J$ . The same phenomenon can be observed for PPO2 on PyBullet Hopper with TRPO reaching better  $R$  values, but PPO2 achieving better values for  $J$ . A2C and DDPG still have the worst performances also when looking at  $R$ , but they appear a bit closer on the PyBullet variant.

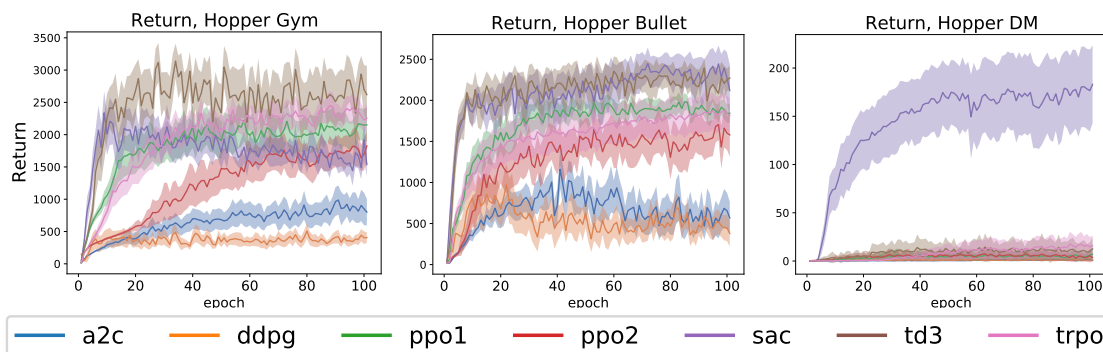


Figure 4.2: Return (undiscounted cumulative reward) on the Hopper environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Hopper, for A2C and PPO2 on PyBullet Hopper, and for all four on-policy algorithms on DeepMind Hopper.

The before mentioned instability of DDPG is also clearly depicted by its exploding value function estimates which can be seen in Figure 4.4. The off-policy nature of the algorithm leads to sudden failures if the environment dynamics are unstable, which is the case for Hopper. That means, the exploration noise can lead to abrupt episode terminations which make it hard to learn a good Q-value estimation [22]. The value function estimates of the other algorithms roughly correspond to the course of their achieved  $J$  values. Notably, SAC and TD3 seem to underestimate the value of the starting states on the PyBullet Hopper environment, because of their Double Q-Learning method. An exception to this behavior can be seen when looking at the course of the value estimate of SAC on DeepMind Hopper. After around 50 epochs with stable estimates, the outputs explode and highly fluctuate for the remaining 50 epochs. We have no explanation of why this always happens at roughly the same point in the training process.

The course of the entropy of the algorithms' policies is shown in Figure 4.5. The entropy of SAC lowers much faster than it does for the other algorithms on both the Gym and the PyBullet variants of Hopper and subsequently stays on roughly the same value. This

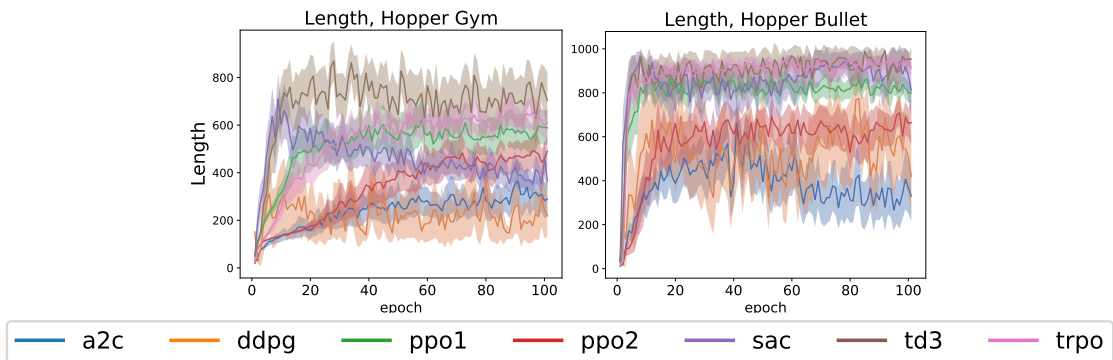


Figure 4.3: Episode length on the Hopper environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Hopper, and for A2C and PPO2 on PyBullet Hopper.

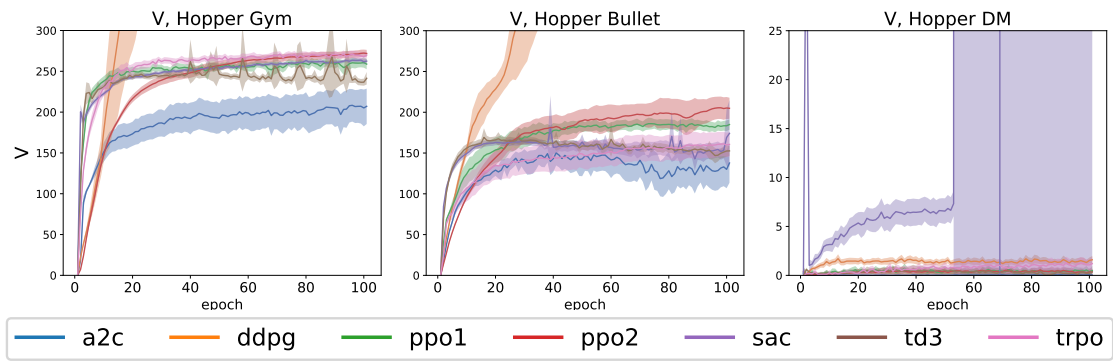


Figure 4.4: Value function output on the Hopper environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Hopper, for A2C and PPO2 on PyBullet Hopper, and for all four on-policy algorithms on DeepMind Hopper.



observation can not be made for DeepMind Hopper where SAC only slightly decreases its entropy at the start and stays there throughout the remaining epochs. In contrast to that, PPO1 and especially PPO2 lower their entropy values much further though initially not as fast. For A2C and TRPO, nearly no decrease in the entropy values is visible, which can be explained by the fact that no performance gains are made and the agents want to keep exploring. For Gym and PyBullet Hopper, the policies of the batch algorithms gradually decrease their entropy over the training process and reach the minimum value after 100 epochs. PPO1 on PyBullet forms an exception to this with increasing entropy values after around 30 epochs.

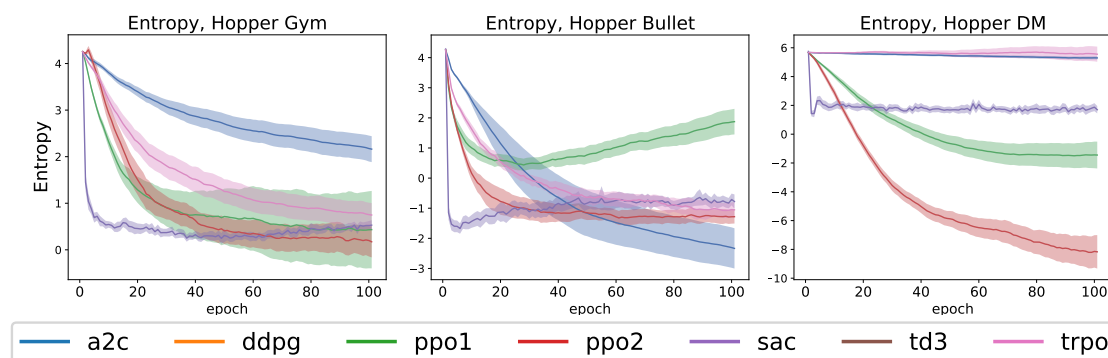


Figure 4.5: Entropy on the Hopper environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Hopper, for A2C and PPO2 on PyBullet Hopper, and for all four on-policy algorithms on DeepMind Hopper.

### 4.1.2 Walker

For both the **OpenAI Gym** and also the **PyBullet** implementation of the Walker task, SAC achieves the best performance regarding the *discounted cumulative reward*  $J$  (see Figure 4.6) closely followed by the results of TD3. On PyBullet Walker, those two algorithms also have a much faster initial learning behavior compared to the other algorithms breaking the mark of  $J = 100$  convincingly after only 10 epochs, which is already in the range of what the other algorithms reach after the full 100 epochs. The same can be said for the DeepMind version of Walker where SAC and TD3 significantly outperform the other algorithms both in terms of initial learning speed and also in terms of asymptotic performance. What is notable is that after the initial big rise in observed  $J$  values, the

---

---

performance stays at about the same level throughout the rest of the training process suggesting that no further improvements can be made. The performances of TRPO, PPO1 and PPO2 are very similar, especially on PyBullet where all three algorithms show nearly the exact same learning behavior only differing in a small drop in performance of PPO1 at the later epochs. On Gym Walker, PPO1 starts off with a bit faster initial learning speed, but ends up at roughly the same final values as PPO2 and TRPO at the 100 epoch mark. It has to be noted that the error margins are a lot bigger comparatively on PyBullet Walker just like for the Hopper task (cf. Section 4.1.1). For DeepMind Walker, PPO1 seems to outperform TRPO and PPO2 significantly, but still does not come close to the performance of SAC and TD3.

The only major difference in algorithm performances between Gym Walker and PyBullet Walker can be seen when looking at the behavior of DDPG. While it is the worst performing algorithm on Gym Walker reaching only less than half the amount of discounted cumulative reward that TRPO, PPO1 and PPO2 achieved, it comes decently close the numbers of those three algorithms on the PyBullet implementation of the Walker environment after 100 epochs. The difference is even bigger for the DeepMind variant of Walker where DDPG even manages to outperform TRPO and PPO2 at the later epochs. The initial learning progress is slightly slower though and the variance of the results is significantly higher as well. The generally much better return values for the DeepMind environment compared to DeepMind Hopper (cf. Section 4.1.1) can be attributed to the fact that the reward function for Walker also is impacted by the torso height of the robot and not only by its forward velocity so that even just standing leads to some reward. A2C is again one of the worst performing algorithms, which is the case for all three libraries.

The results in terms of the *undiscounted cumulative reward*  $R$  (see Fig. 4.7) look very similar to what can be observed for  $J$ . TD3 manages to reach results that are just as good as the ones of SAC on the Gym and PyBullet variant, but still lacks behind for DeepMind Walker. The other algorithms mostly follow their performances in terms of  $J$  with the exception of PPO2 that is achieving slightly worse results for  $R$  on PyBullet compared to TRPO and PPO1, which is not the case when looking at  $J$ . For Gym Hopper, we can see the non-optimality of the DDPG learning process even more drastically when looking at the course of  $R$ . It seems to have nearly no improvement for this evaluation criterion making it clearly the worst performing algorithm on this specific environment implementation. For DeepMind Walker, the undiscounted returns nearly look exactly the same as the discounted ones. We can also see that SAC achieves a near-optimal performance getting very close to the theoretical maximum return of 1000. The remaining gap is most likely not closable, because the agent first needs to get into a state where always receiving the full reward of 1 is possible.

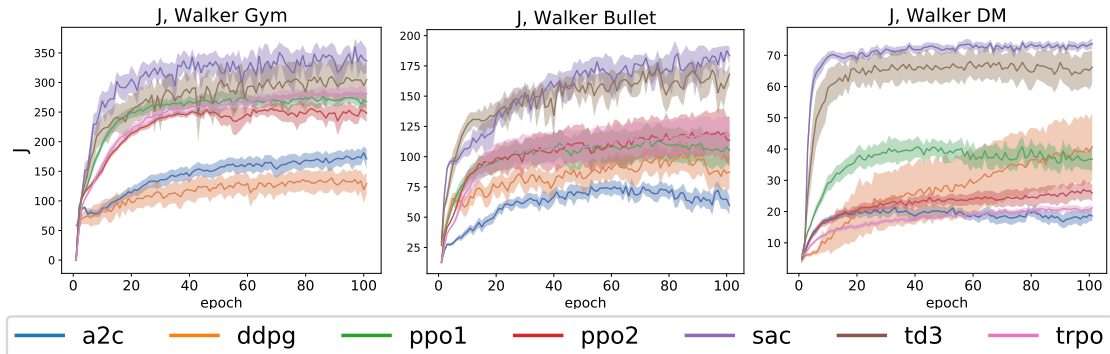


Figure 4.6: J (discounted cumulative reward) on the Walker environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Walker, for TRPO, PPO1 and PPO2 on PyBullet Walker, and for all four on-policy algorithms on DeepMind Walker.

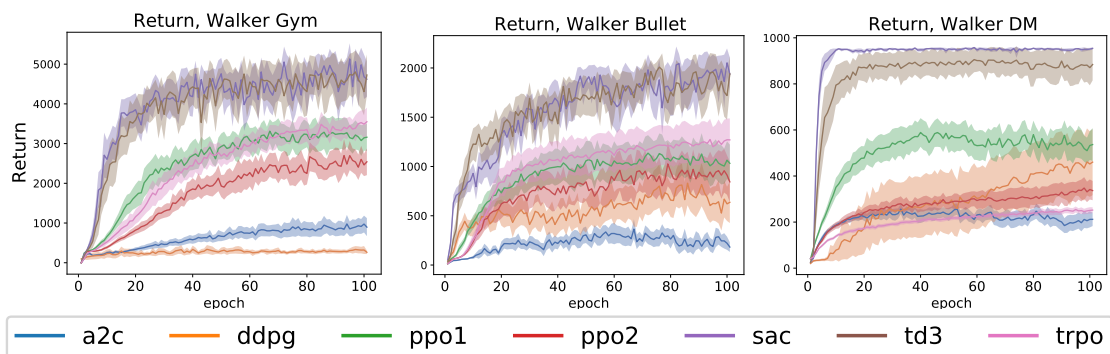


Figure 4.7: Return (undiscounted cumulative reward) on the Walker environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Walker, for TRPO, PPO1 and PPO2 on PyBullet Walker, and for all four on-policy algorithms on DeepMind Walker.

Figure 4.8 shows the average episode lengths during the training process. For the Gym variant, there are no particular differences compared to the results for  $R$ . While SAC and TD3 manage to consistently reach nearly the maximum amount of timesteps per episode after 100 epochs, DDPG does not even find a policy that regularly leads to the robot walking for more than 200 timesteps per episode before falling over. For the PyBullet variant, it is notable that DDPG, PPO1 and especially TRPO achieve much better results in terms of average episode length than they are when comparing  $R$  - with TRPO reaching about the same value at epoch 100 as SAC and TD3. The fact that the values for  $R$  look so much worse comparatively can be explained by the fact that those algorithms get stuck in local optima trying to stabilize the robot as long as possible without finding ways to concurrently maximize the forward velocity of the robot. The episode lengths for DeepMind Walker are not shown, because each episode runs for exactly 1000 timesteps.

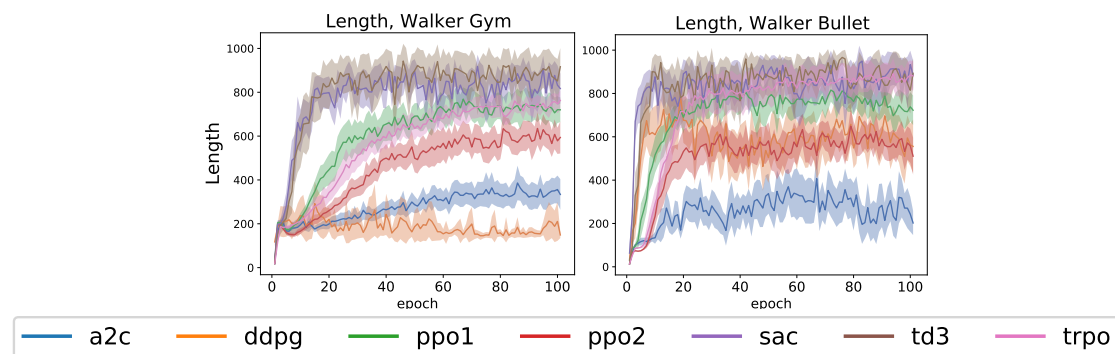


Figure 4.8: Episode length on the Walker environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Walker, and for TRPO, PPO1 and PPO2 on PyBullet Walker.

Figure 4.9 depicts the course of the value function estimates on the Walker environments. While the estimated values are very close to the achieved  $J$  values for PPO1, PPO2, TRPO and A2C, there are apparent differences for SAC, TD3 and DDPG. SAC and TD3 tend to underestimate the value of the initial states by a considerable margin for all libraries, especially at the later stages of the learning process. In contrast to that, DDPG tends to overestimate this value by nearly 100% of the actual observed  $J$  values for the Gym and PyBullet implementations. For DeepMind Walker, though, the value estimates of DDPG seem very accurate, which is also why the performance of DDPG is comparatively high here.

The entropy development shown in Figure 4.10 can be compared with the one that is

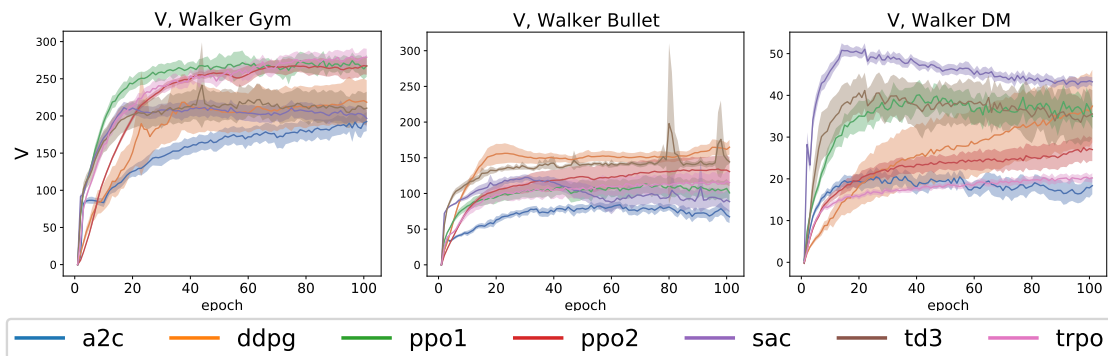


Figure 4.9: Value function output on the Walker environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Walker, for TRPO, PPO1 and PPO2 on PyBullet Walker, and for all four on-policy algorithms on DeepMind Walker.

observable for the Hopper task (cf. Section 4.1.1). The entropy of the policy of SAC again rapidly decreases at the start of the training process. For PyBullet Walker, it then slowly increases again while it keeps slowly decreasing on the Gym environment. The other algorithms follow roughly the same trend as on the Hopper environment (cf. Fig. 4.5). An exception to this forms PPO2 on DeepMind Walker, which lowers its entropy much slower than it is the case for Hopper. The opposite can be said for SAC which reaches much lower final entropy values corresponding to the fact that a near-optimal policy is learned.

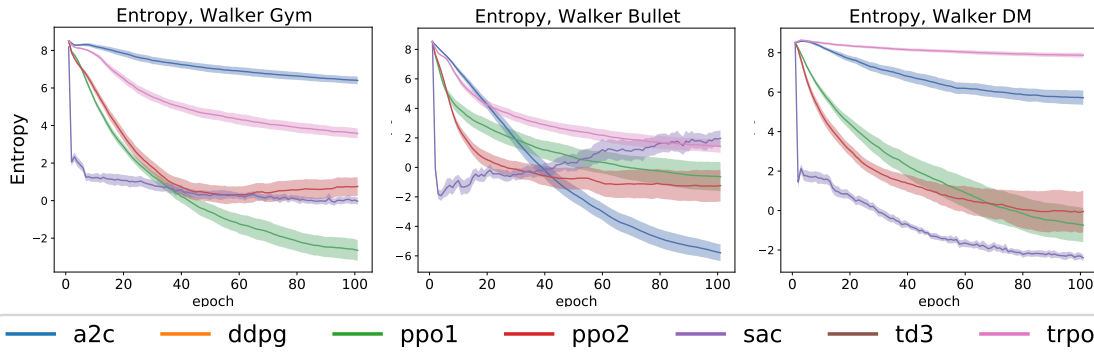


Figure 4.10: Entropy on the Walker environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Walker, for TRPO, PPO1 and PPO2 on PyBullet Walker, and for all four on-policy algorithms on DeepMind Walker.

### 4.1.3 Cheetah

Looking at the performances for the Cheetah task in terms of *discounted cumulative reward*  $J$  (Figure 4.11), we can see several similarities between the three task libraries. A2C always achieves the worst result overall, which mirrors what can be observed for the other benchmark tasks. The algorithms TRPO, PPO1 and PPO2 all perform very similarly. However, they differ in their relative performance compared to the other algorithms. While achieving a comparatively weak performance on Gym Cheetah (only slightly better than A2C), the results on DeepMind Cheetah look a lot better. For PyBullet Cheetah, PPO1 and PPO2 even manage to achieve similar  $J$  values as the top performing algorithms for this library (SAC, DDPG and TD3). TRPO lacks a bit behind though. TD3 and SAC also reach the best results for the Gym and DeepMind versions of Cheetah. Not only do they have the best sample efficiency with a fast initial learning speed, but also do they reach by far the best final performances after 100 epochs for those environments. For DDPG, the results vary a lot between the three task libraries. On DeepMind Cheetah, it performs worse than every other algorithm except A2C. For Gym Cheetah, DDPG has the third best performance, but is far from reaching the values of SAC and TD3. On the PyBullet version, however, it reaches one of the best results and even outperforms SAC on the later epochs. These varying results show that, even though all libraries technically provide environments for the same task (Cheetah), the actual performance can be significantly different, because of different reward functions and different exact dynamics and episode-end criteria. New

emerging algorithms might be presented using the library that fits best without mentioning that the results are worse for other implementations of the same environment, which is why those results always have to be taken with caution.

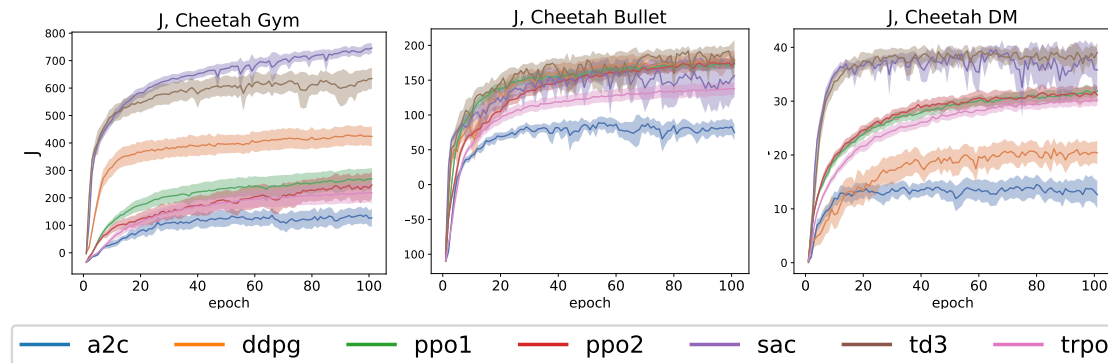


Figure 4.11:  $J$  (discounted cumulative reward) on the Cheetah environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Cheetah, for A2C, TRPO and PPO1 on PyBullet Cheetah, and for all four on-policy algorithms on DeepMind Cheetah.

In Figure 4.12, we can see that the results in terms of the *undiscounted cumulated reward* follow the ones for  $J$  very closely. The only notable differences can be observed for DDPG on the PyBullet version of Cheetah. Here, the agent seems to learn a policy that focuses more on immediate rewards than on long-term rewards, because the performance for the undiscounted return formulation looks worse than the one for the discounted case, especially at the final epochs where DDPG ends up in the second to last spot only followed by A2C. Roughly the same is the case for the DeepMind version of Cheetah, although the variance for the undiscounted return is lower here.

Since the episode lengths are fixed at 1000 timesteps for all three versions of Cheetah, there are no figures provided for this metric here.

The course of the *value function estimates* is depicted in Figure 4.13. We can see that TD3 and SAC heavily underestimate the expected values for Gym Cheetah, which can be attributed to the Double Q-Learning trick. This behavior can also be observed for DeepMind Cheetah except for the evaluations on some of the later epochs where the value function estimate is exploding for SAC for a short time. The same happens for SAC on PyBullet Cheetah and also in a less drastic way for TD3 on PyBullet and Gym Cheetah.

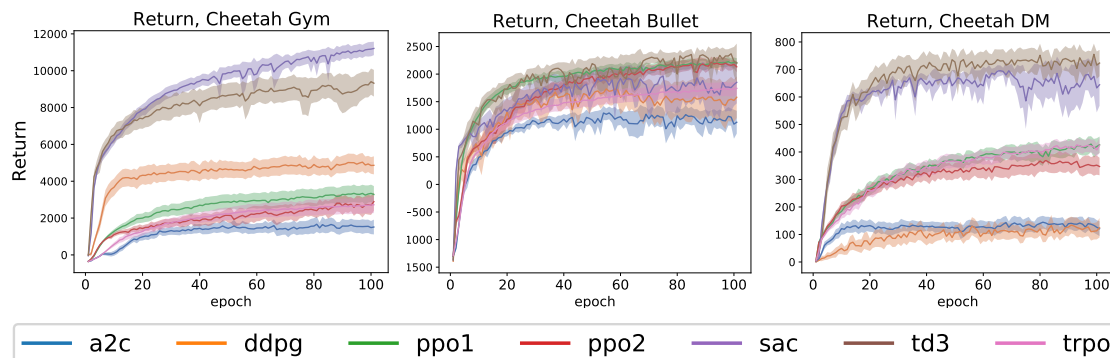


Figure 4.12: Return (undiscounted cumulative reward) on the Cheetah environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Cheetah, for A2C, TRPO and PPO1 on PyBullet Cheetah, and for all four on-policy algorithms on DeepMind Cheetah.

For DDPG, however, the value function estimates seem very stable and closely follow the observed  $J$  values for all three benchmark libraries. The stable dynamics of the Cheetah task lead to estimates that are significantly more accurate than the ones for DDPG on Hopper and Walker (cf. Section 4.1.1 and 4.1.2). For the remaining four algorithms, the value function output is very accurate as well.

Figure 4.14 shows the course of the *entropy* values for the Cheetah task. For A2C, TRPO, PPO1 and PPO2 the entropy gradually decreases with more epochs. The PPO algorithms generally decrease their entropy faster than the other two batch algorithms, which is the case for all three benchmark libraries. For DeepMind Cheetah, it is notable that the entropy of A2C is only decreasing very slowly compared to how much it is decreasing on the PyBullet and Gym counterparts. SAC has a very fast drop at the starting epochs, followed by a steep but short rise and then a slow decline for the remaining epochs (Gym and DeepMind Cheetah). For the PyBullet variant, the entropy drops even lower at the start, but does not have the sudden rise after that and instead slowly increases over time.



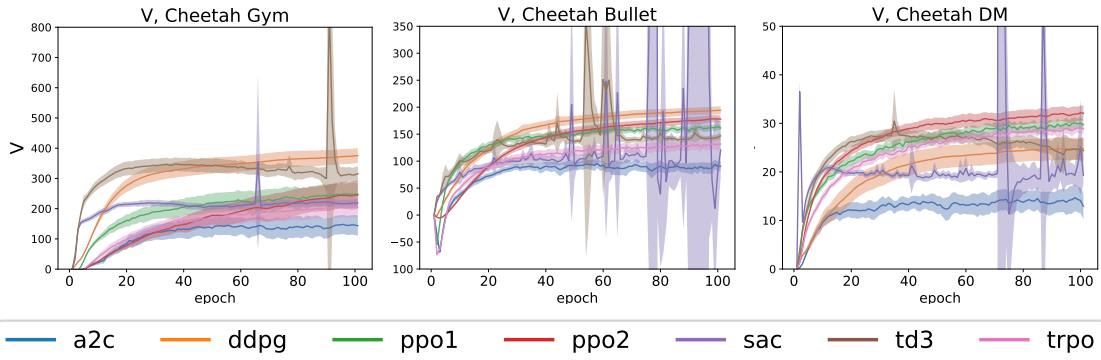


Figure 4.13: Value function output on the Cheetah environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Cheetah, for A2C, TRPO and PPO1 on PyBullet Cheetah, and for all four on-policy algorithms on DeepMind Cheetah.

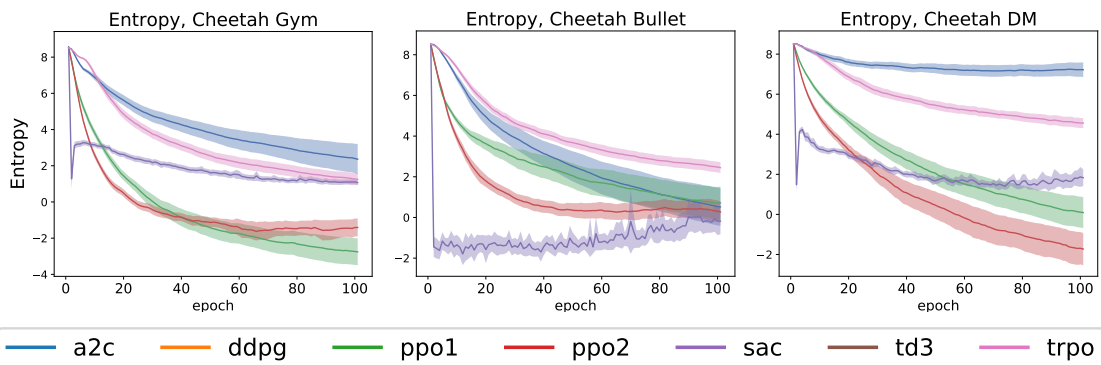


Figure 4.14: Entropy on the Cheetah environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C, TRPO, PPO1 and PPO2 on Gym Cheetah, for A2C, TRPO and PPO1 on PyBullet Cheetah, and for all four on-policy algorithms on DeepMind Cheetah.

#### 4.1.4 Ant

Figure 4.15 shows the algorithm performances with respect to the *discounted cumulative reward*  $J$  on the different versions of Ant. The results of TD3 vary between the three libraries. While it only achieves moderate discounted returns for the DeepMind implementation, it reaches by far the best performance on Gym Ant and is also among the top algorithms for the PyBullet version only challenged by SAC, which also has varying results. It is the second best performing algorithm on Gym Ant, but considerably trails behind TD3. For the DeepMind implementation, however, it accomplishes the highest  $J$  values by a wide margin. SAC has high variance in its results, which can be seen by the large confidence intervals. This observation can also be made for DDPG, which overall reaches rather mediocre results. What is notable is that its learning curve for Gym Ant starts very promising with a very fast initial rise in performance. However, after that, the observed  $J$  values slowly decline and reach their lowest point at the final epochs. This behavior of dropping performance is not present for any of the other algorithms. The two PPO implementations have comparable learning curves and reach a decent performance for all three libraries. For TRPO, the achieved discounted returns are generally lower, especially at the early epochs. Later during the training process, it catches up to the other algorithms and even outperforms all of them except SAC on DeepMind Ant. A2C accomplishes the worst overall results, but can keep up with the other algorithms for the Gym version of Ant. PPO, TRPO and A2C have very low variance in their results compared to the off-policy algorithms.

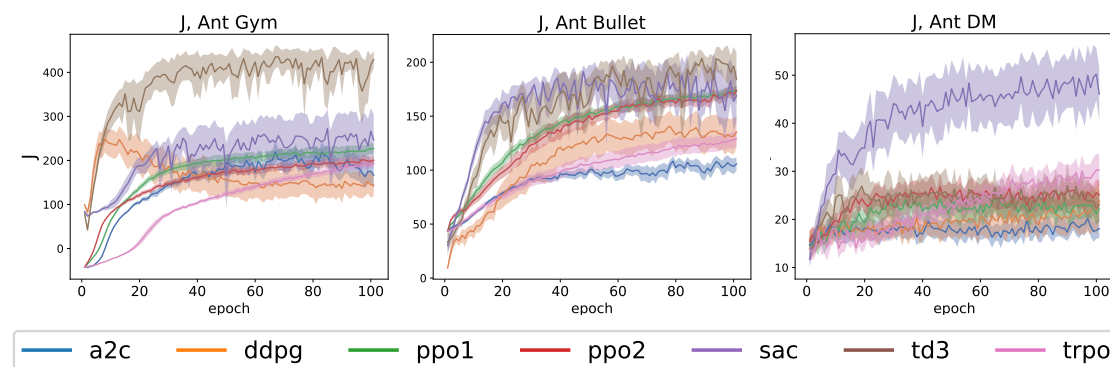


Figure 4.15:  $J$  (discounted cumulative reward) on the Ant environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C on Gym Ant and for all four on-policy algorithms on DeepMind Ant.

Looking at the *undiscounted cumulative reward*, we can see that there are nearly no major differences to what can be observed when looking at  $J$  (see Figure 4.16). For the Gym and DeepMind implementations, the curves very closely follow the trends that could be seen in the discounted case. The same is the case for PyBullet Ant with the only exception being DDPG, which accomplishes the worst undiscounted return values although it seemingly outperformed A2C and TRPO when looking at the discounted returns.

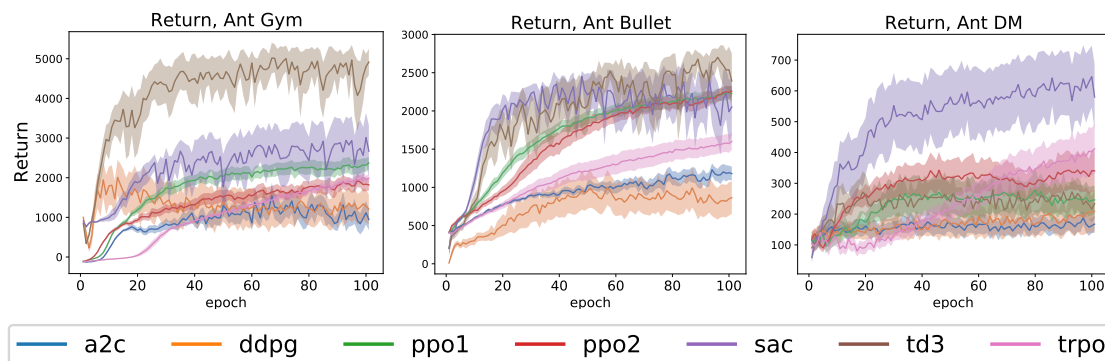


Figure 4.16: Return (undiscounted cumulative reward) on the Ant environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C on Gym Ant and for all four on-policy algorithms on DeepMind Ant.

The *value function estimates* for A2C, TRPO, PPO1 and PPO2 closely follow the achieved discounted returns on all three Ant implementations with only A2C slightly overestimating the expected values on Gym Ant (see Figure 4.17). TD3 tends to underestimate the values instead, especially on Gym Ant. For the PyBullet version, the estimates seem very unstable with high variance, but are roughly accurate in the mean. SAC, as the other algorithm that uses the Double Q-Learning trick, also shows tendencies of underestimation, but sometimes has big spikes in the value function output. While this phenomenon only happens twice during the learning process for DeepMind Ant, it is present throughout the whole learning process for the Gym Ant, which leads to tremendous confidence intervals. This instability in the value function estimates might be the reason for the comparatively low performance of SAC for this specific implementation of Ant. DDPG shows considerably less overestimation than it did for the Hopper and Walker tasks (see Section 4.1.1, 4.1.2). Only for DeepMind Ant does it show a big spike in the estimates at the early epochs.

The episode lengths are fixed at 1000 timesteps for the DeepMind implementation and also on PyBullet Ant nearly all algorithms reach the maximum of 1000 timesteps per episode very fast (see Figure 4.18). The only exception to this forms DDPG, which has

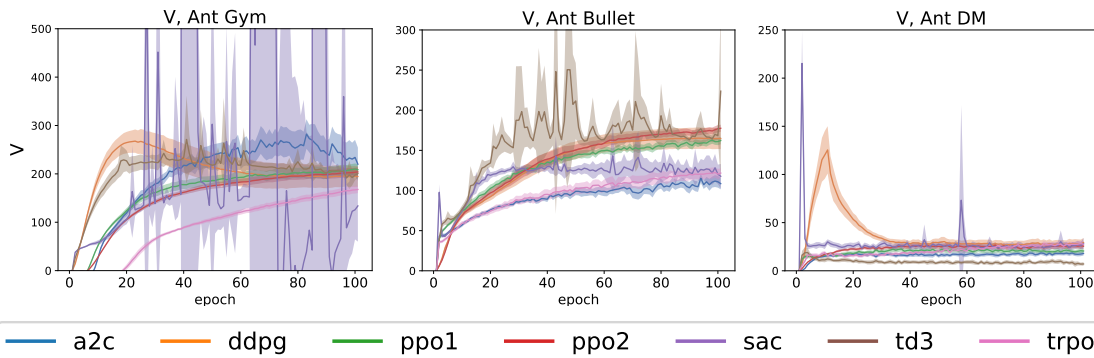
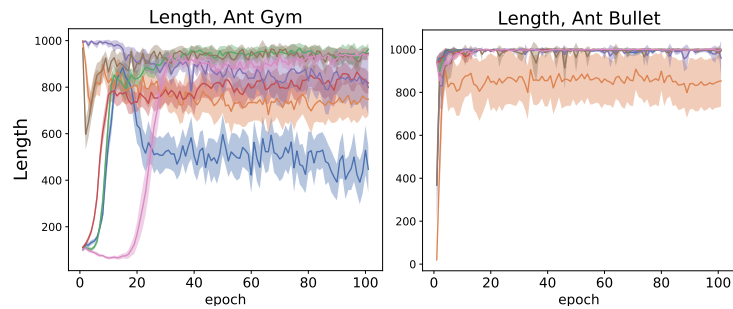


Figure 4.17: Value function output on the Ant environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C on Gym Ant and for all four on-policy algorithms on DeepMind Ant.

a much more unstable behavior with lower average episode lengths with high variance. For the Gym implementation of Ant, we can see that PPO1 converges to a local optimum that achieves high episode lengths very fast, but does not lead to high returns as they can be observed for TD3. TRPO takes a long time to learn a policy that leads to long episode lengths, which is related to its previously observed slow learning speed in terms of discounted and undiscounted return. A2C initially reaches high average episode lengths very fast, but subsequently experiences a drop that leads to much shorter lengths for the remaining epochs. However, the observed returns are not dropping in a similar way, so the learned policies favour actions that lead to higher quality immediate rewards while accepting potentially faster episode fails for that.

The behavior of DDPG for PyBullet and A2C for Gym is an example of where reporting results in terms of average reward would be very misleading since the algorithms may achieve a decent average reward, but do not manage to keep the robot alive for the whole episode leading to worse cumulative rewards. This effect can to some extent already be seen when comparing the discounted and undiscounted returns as described in the previous paragraphs, because shorter episode lengths have less impact in the discounted case as the missing rewards at the later timesteps are discounted anyway.

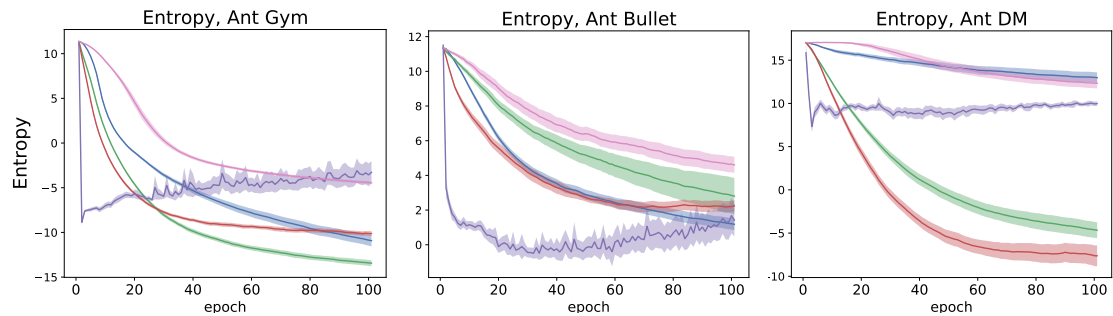
The *entropy* development looks normal for A2C, TRPO and PPO on Gym and PyBullet Ant with slowly declining values over the course of the training process (see Figure 4.19). For the DeepMind version, A2C and TRPO decrease the entropy of their policies much slower than the two PPO versions and therefore still do a lot of exploration at the later epochs. For TRPO this behavior might lead to much higher returns if the algorithm was to



— a2c — ddpq — ppo1 — ppo2 — sac — td3 — trpo

Figure 4.18: Episode length on the Ant environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C on Gym Ant.

be trained for more than 100 epochs (cf. Fig. 4.30). SAC also keeps its entropy very high throughout the learning process for DeepMind Ant, which is not the case for the Gym and PyBullet variants where the entropy drops really fast at the start and then slowly increases with further epochs.



— a2c — ddpq — ppo1 — ppo2 — sac — td3 — trpo

Figure 4.19: Entropy on the Ant environment averaged over 25 runs with 95% confidence bounds. Input normalization was used for A2C on Gym Ant and for all four on-policy algorithms on DeepMind Ant.

---

## 4.2 Normalization

---

In this section, we investigate the impact of *input normalization* for the on-policy algorithms A2C, TRPO, PPO1 and PPO2. The algorithms are trained for 350 epochs (5734400 training steps) as opposed to only 100 epochs in the previous section, because the training process is considerably faster for these algorithms. The normalization takes place by using the *VecNormalize* wrapper of Stable Baselines with the clipping parameter *clip\_obs* set to 10.0. The rewards are not normalized so that differences in performance can fully be attributed to the input normalization.

### 4.2.1 Hopper

Figure 4.20 shows how A2C is affected by input normalization for the Hopper task. We can see that the performance is considerably improved for the Gym environment, but the variance is also increased in the process. For PyBullet Hopper, the results are similar in terms of the undiscounted return, but seem to improve for the discounted case. The observed returns on the DeepMind version of Hopper are very low for A2C with and without input normalization, although the normalization leads to very slight improvements with high variance.

Looking at the normalization impact for TRPO, we can see clear improvements for Gym Hopper (see Figure 4.21). On the PyBullet environment it is not clear whether input normalization really improves the performance because of the high variance, but the undiscounted return values seem worse in the mean by a small margin. For DeepMind Hopper, the normalization leads to better mean returns, but the variance is drastically increased so that those results have to be taken with caution.

The impact of normalized inputs for PPO1 and 2 is depicted in Figure 4.22. For Gym Hopper, no clear conclusion can be drawn. The normalization seems to improve the learning behavior in the earlier epochs, but leads to lower final values at the later epochs. Looking at the PyBullet variant of Hopper, the results look extremely close for PPO1. For PPO2, however, normalized inputs lead to moderately higher returns, but again with high variance. For DeepMind Hopper, the performance can be increased tremendously both for PPO1 and 2. Without normalization, the algorithms show close to no learning behavior while the normalized variants can achieve good results.

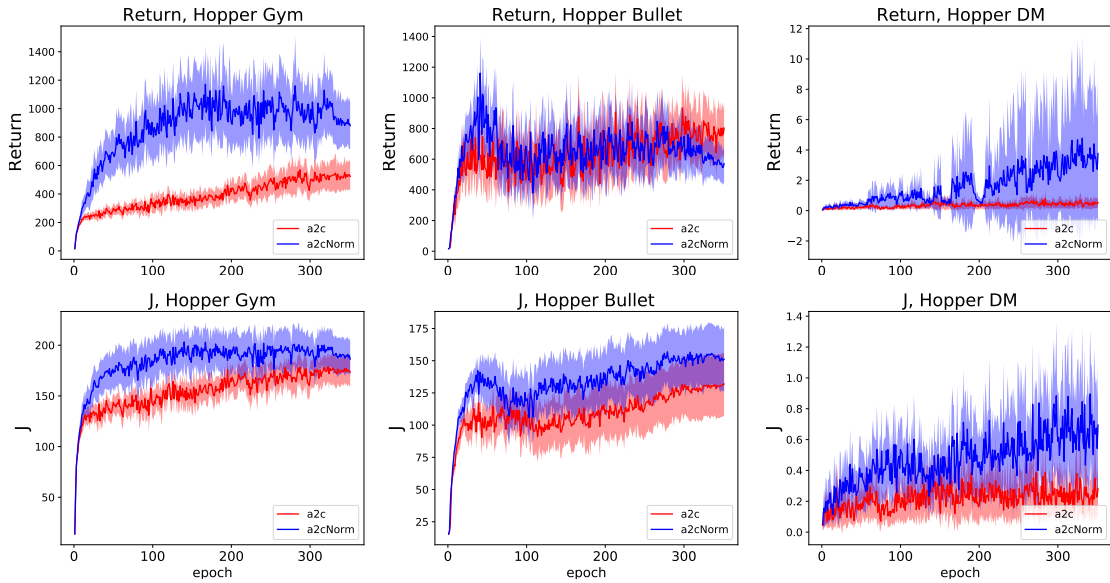


Figure 4.20: A2C comparison of normalized and non-normalized observations on the Hopper environment averaged over 25 runs with 95% confidence bounds.

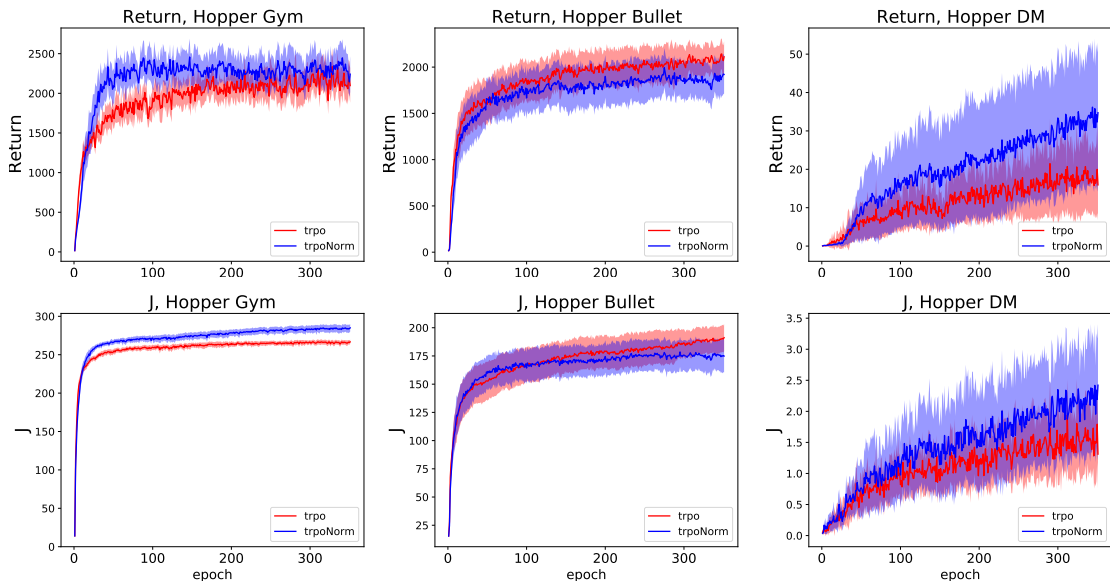


Figure 4.21: TRPO comparison of normalized and non-normalized observations on the Hopper environment averaged over 25 runs with 95% confidence bounds.

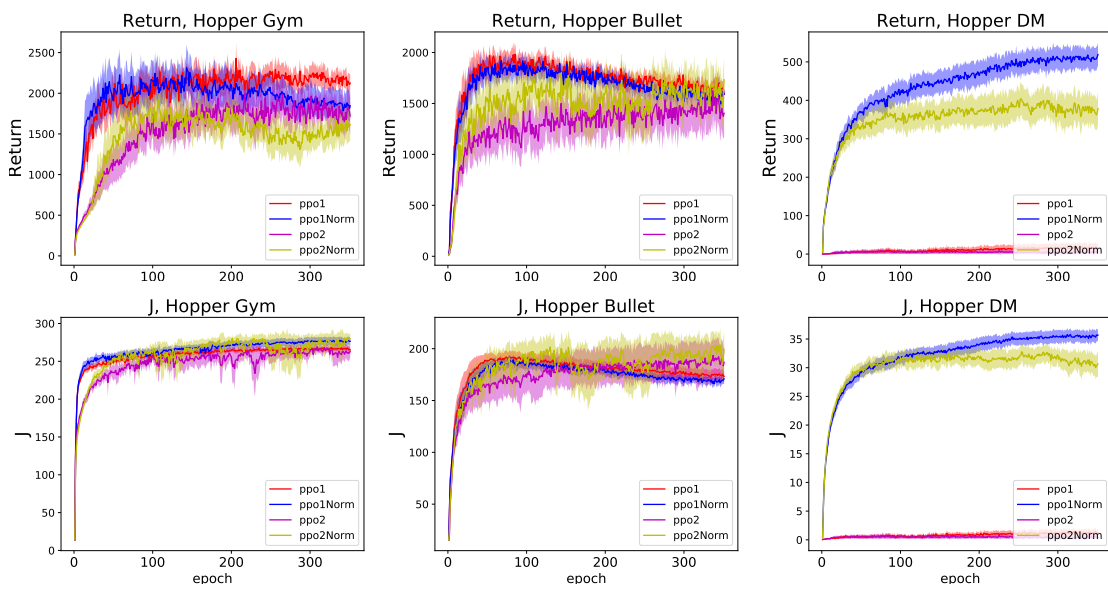


Figure 4.22: PPO1 and PPO2 comparison of normalized and non-normalized observations on the Hopper environment averaged over 25 runs with 95% confidence bounds.



## 4.2.2 Walker

On the Walker task, we can see that A2C can achieve considerable performance gains by using input normalization (Figure 4.23), both for Gym and DeepMind Walker. The PyBullet variant paints a different picture when looking at the discounted return. Here, the normalization seems to worsen the results. In the undiscounted case, there is no clear advantage for either approach.

TRPO can also substantially increase its achieved returns by using normalized inputs on the Gym variant of Walker (see Figure 4.24). For PyBullet Walker, this effect is not as apparent with the results laying a lot closer and being of higher variance. Even less differences between the two approaches are present for DeepMind Walker where the results are almost identical.

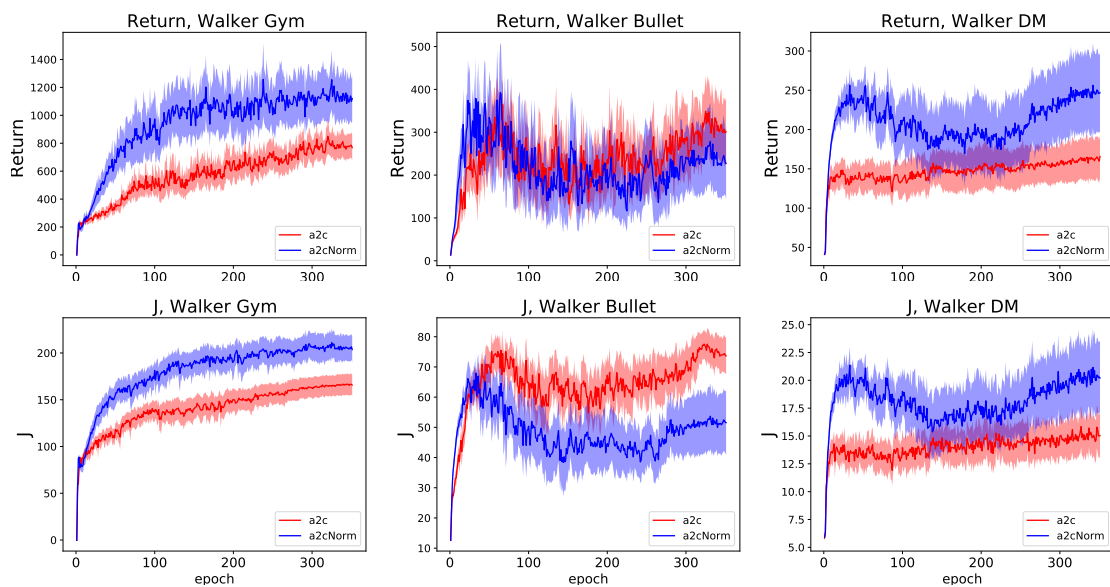


Figure 4.23: A2C comparison of normalized and non-normalized observations on the Walker environment averaged over 25 runs with 95% confidence bounds.

When looking at the returns of the two PPO variants on Gym and PyBullet Walker (Figure 4.25), we can see that input normalization leads to higher performances, but also higher variances. For the DeepMind version of Walker, we can say the same for PPO1 where not only the final returns are superior, but even the initial learning speed is a lot faster than

---

without normalized inputs. PPO2 does not show the same behavior. Here, the results of normalization are marginally worse.

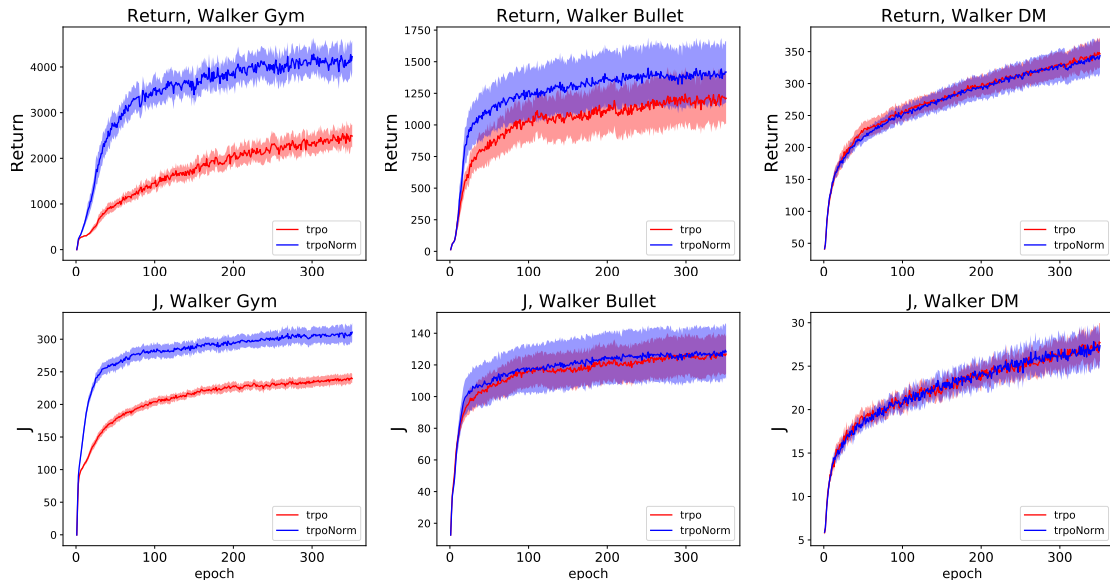


Figure 4.24: TRPO comparison of normalized and non-normalized observations on the Walker environment averaged over 25 runs with 95% confidence bounds.

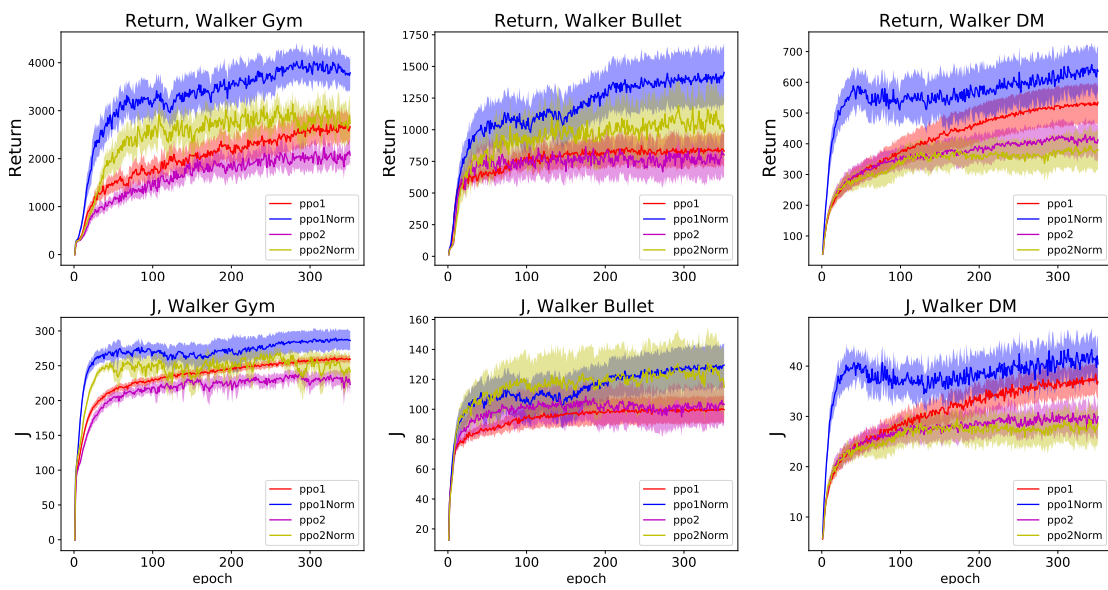


Figure 4.25: PPO1 and PPO2 comparison of normalized and non-normalized observations on the Walker environment averaged over 25 runs with 95% confidence bounds.

### 4.2.3 Cheetah

A2C achieves higher returns on the Cheetah task for all three benchmark libraries when using input normalization (see Figure 4.26). This is the case for both the discounted and undiscounted case. For PyBullet Cheetah, using normalized inputs even lowers the variance of the return values.

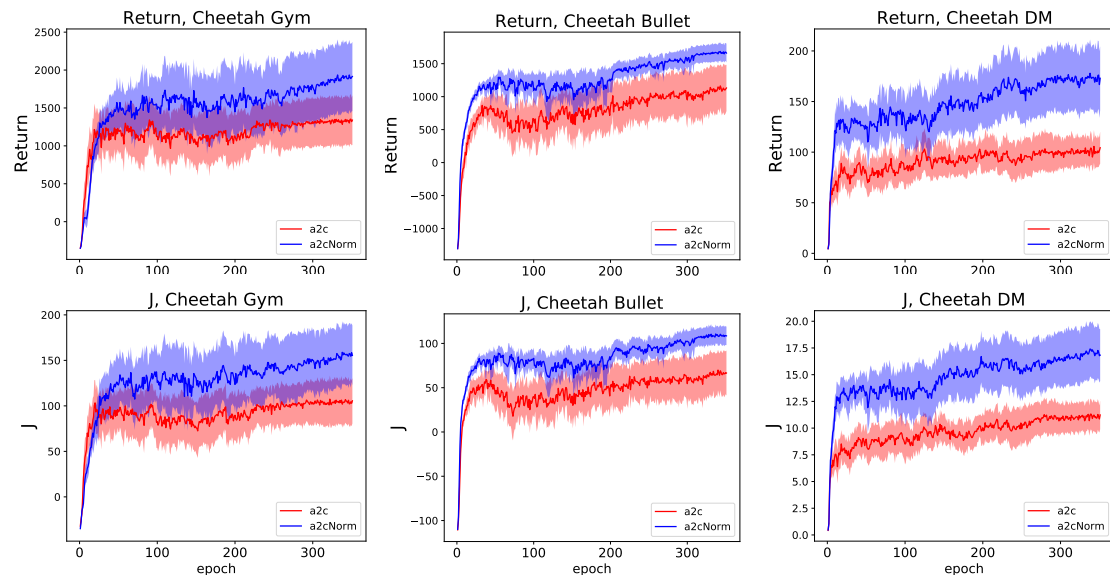


Figure 4.26: A2C comparison of normalized and non-normalized observations on the Cheetah environment averaged over 25 runs with 95% confidence bounds.

TRPO also seems to considerably benefit from normalizing the inputs for this task. The advantage becomes apparent when looking at Figure 4.27 where the achieved returns are substantially higher for the DeepMind and Gym environments. For PyBullet Cheetah, the gains are very slim for both the discounted and also the undiscounted case.

Figure 4.28 shows the normalization impact for PPO on Cheetah. Both versions of PPO reach higher discounted and undiscounted returns on the Gym and DeepMind environments. PyBullet Cheetah forms an exception again. While normalized inputs work slightly better for PPO1, they lead to worse results for PPO2 with higher variance.

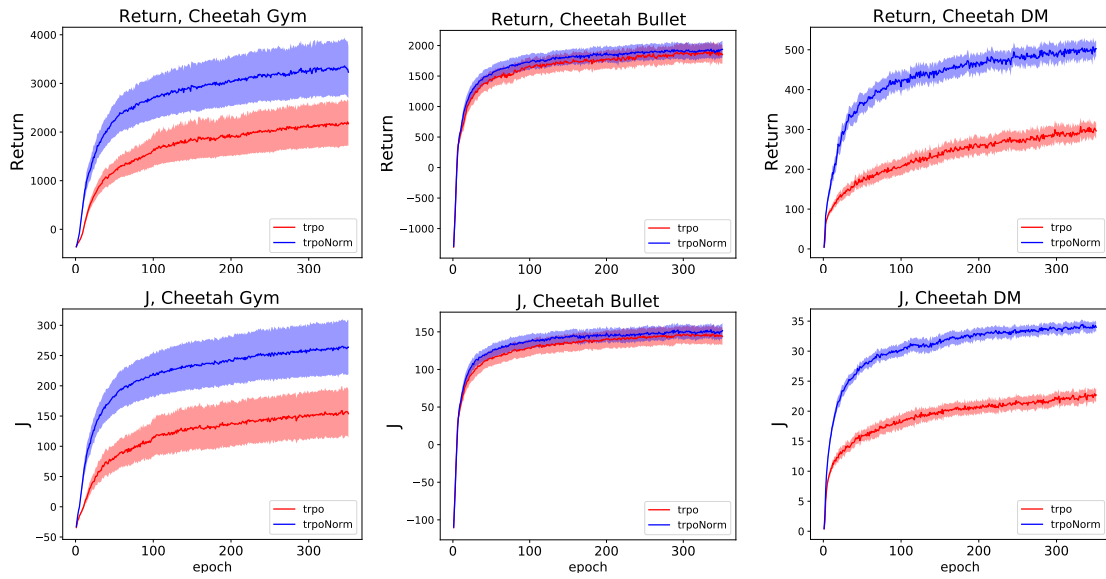


Figure 4.27: TRPO comparison of normalized and non-normalized observations on the Cheetah environment averaged over 25 runs with 95% confidence bounds.

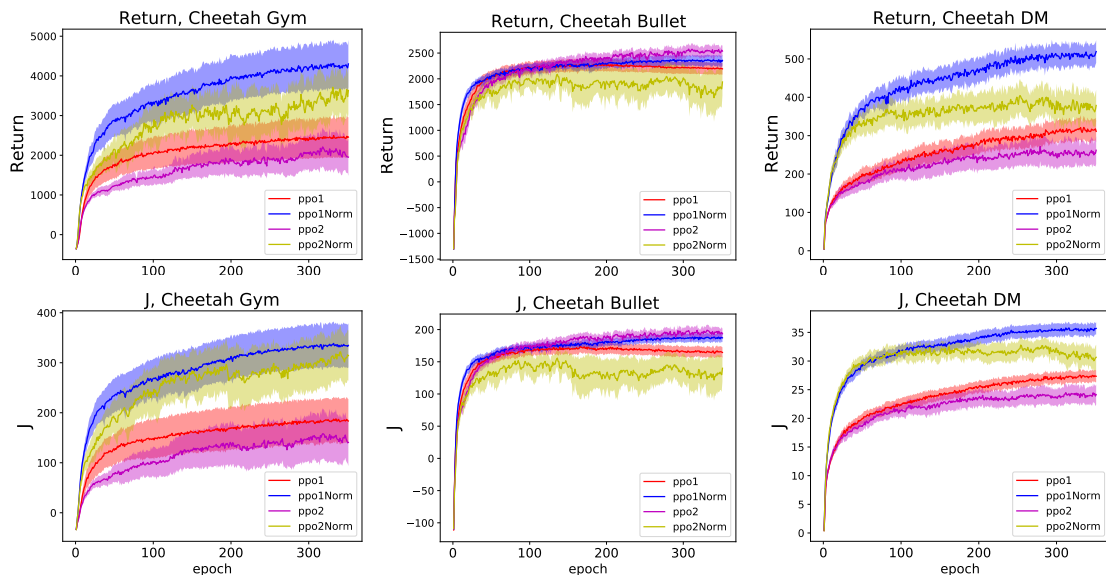


Figure 4.28: PPO1 and PPO2 comparison of normalized and non-normalized observations on the Cheetah environment averaged over 25 runs with 95% confidence bounds.

---

## 4.2.4 Ant

The impact of using normalized inputs for the Ant task greatly differs between the different algorithms and the different benchmark libraries.

For A2C, the achieved  $J$  values are higher for normalized Gym Ant, but lower for normalized PyBullet Ant (see Figure 4.29). On the DeepMind implementation, it is hard to spot a difference, also because of the high variance. When looking at the undiscounted return, there are also no clear conclusions possible since input normalization seems better for the DeepMind version, but worse for PyBullet and about equal for Gym Ant.

For TRPO, the results on normalized Gym and PyBullet Ant not only are of higher variance, but also have lower mean. On DeepMind Ant, however, input normalization leads to significantly better discounted and undiscounted returns, which can be seen in Figure 4.30. Especially at the earlier epochs, the learning seems to happen a lot faster on this specific environment.

The two PPO algorithms also differ in how they react to input normalization on the Ant task (see Figure 4.31). For Gym Ant, non-normalized inputs outperform normalized ones later during the training process in the case of PPO2, but for PPO1 there is no clear better variant. On the PyBullet environment, input normalization leads to drastically worse results for PPO2 and the same is the case for PPO1 to a lesser but still significant extent. For DeepMind Ant, a contrary behavior can be observed, especially for the undiscounted return where normalized inputs increase the performance for both PPO versions. However, this improvement comes with the prize of largely increased variance in the observed returns.

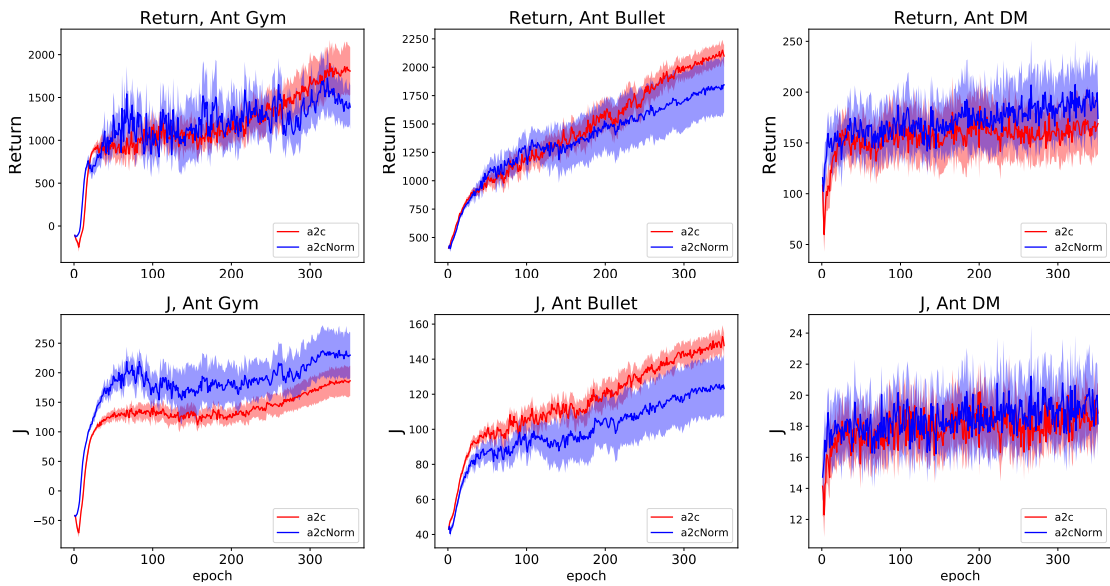


Figure 4.29: A2C comparison of normalized and non-normalized observations on the Ant environment averaged over 25 runs with 95% confidence bounds.

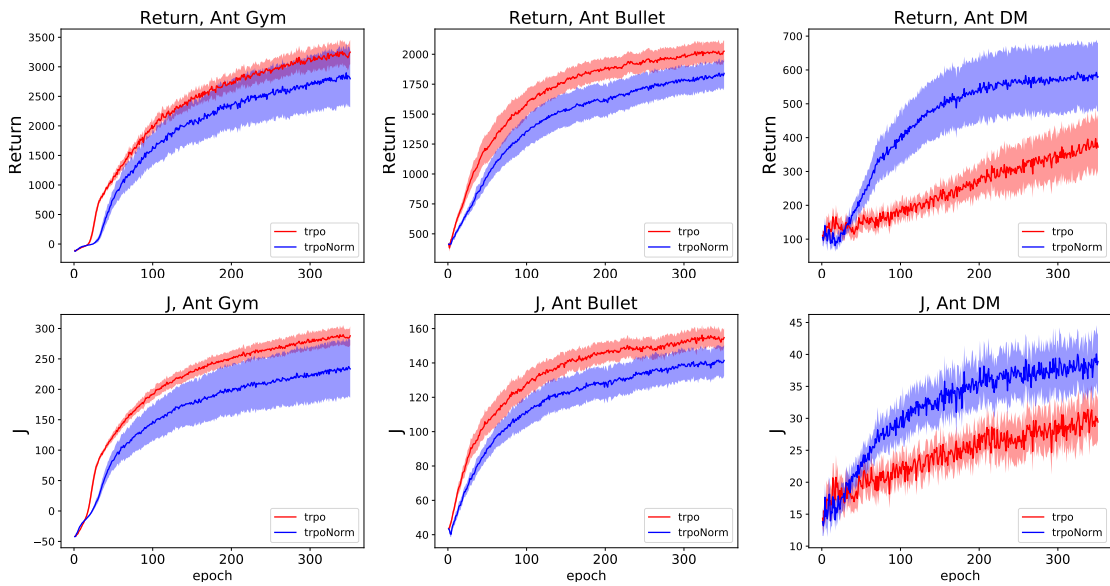


Figure 4.30: TRPO comparison of normalized and non-normalized observations on the Ant environment averaged over 25 runs with 95% confidence bounds.

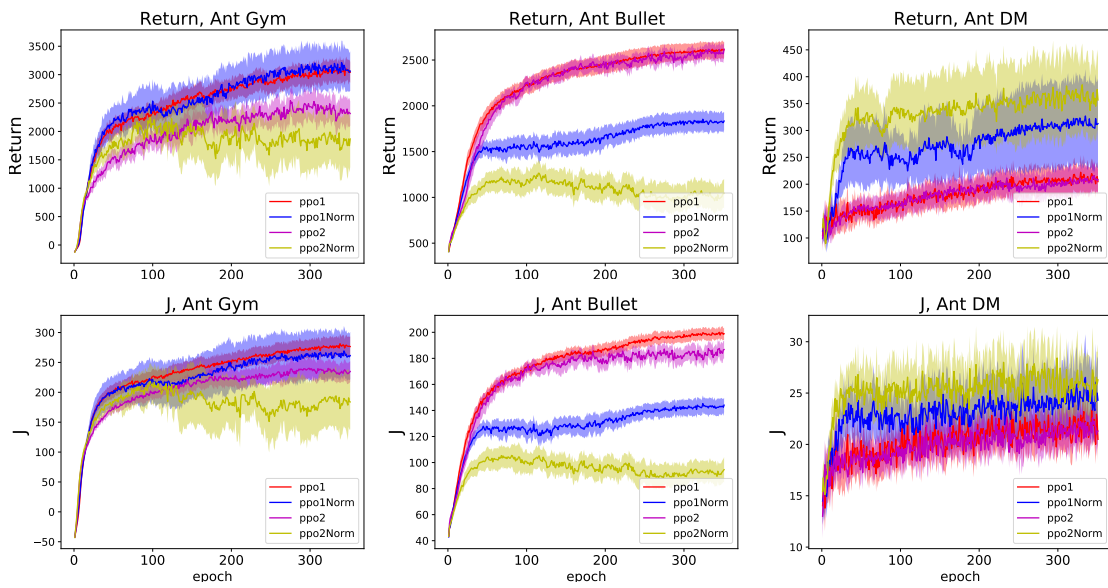


Figure 4.31: PPO1 and PPO2 comparison of normalized and non-normalized observations on the Ant environment averaged over 25 runs with 95% confidence bounds.



---

### 4.3 Network Size

---

In [21] and [22], it is shown that bigger network architectures could significantly improve the results of DDPG on certain environments, because of the amount of information that can be stored in the network. We investigate to which extent the same behavior can be observed for TD3 and SAC by evaluating all three off-policy algorithms with small and big network sizes. We use feed forward neural networks with hidden layer sizes of (32, 32) as well as (400, 300) and compare the results in terms of average discounted and undiscounted return of both setups (referenced as *SmallNet* and *BigNet*). ReLU is used as the activation function across all experiments and all other hyperparameters are held fixed. We exemplarily show the results on the OpenAI Gym implementations of Hopper, Cheetah and Ant as those three tasks have the most diverse dynamics while Walker behaves similarly to Hopper to a certain degree.

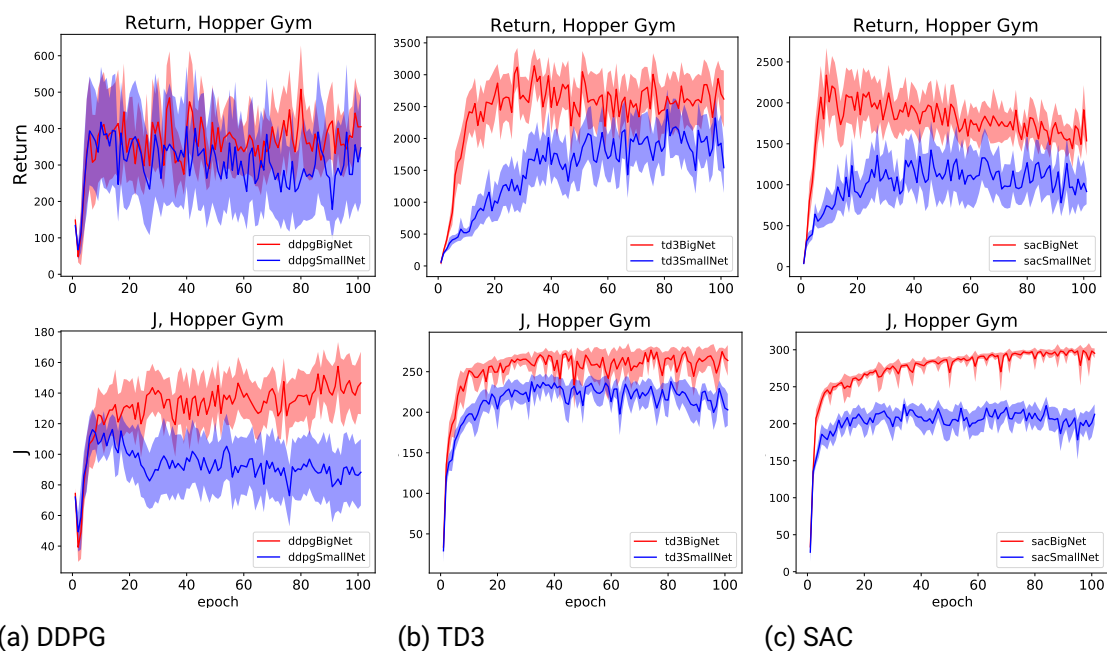


Figure 4.32: Comparison of small and big networks on the Hopper environment averaged over 25 runs with 95% confidence bounds.

On **Hopper**, we can see that the improvements are significantly more apparent for TD3

and SAC than they are for DDPG (see Figure 4.32). Both the discounted as well as the undiscounted return values are considerably higher for TD3 and SAC using the bigger network setup. This improvement is not only expressed in better peak performances at the later epochs, but also in higher sample efficiency with faster initial learning speeds. For DDPG, we can only see noticeable improvements in terms of average  $J$  while there is no clear better setup recognizable in the undiscounted case. However, the bigger network setup seems to considerably reduce the variance of the observed returns, which leads to smaller confidence intervals.

The results of different network architectures on the Gym **Cheetah** environment can be seen in Figure 4.33. As it is also reported in [21], the network size seems to have big effects on the performance of DDPG, which is improved both in terms of discounted and undiscounted return. The same statement can be made about the other two algorithms which also show significantly higher  $J$  and  $R$  values over the whole training process. The variance is staying similar for all three algorithms no matter which architecture is used.

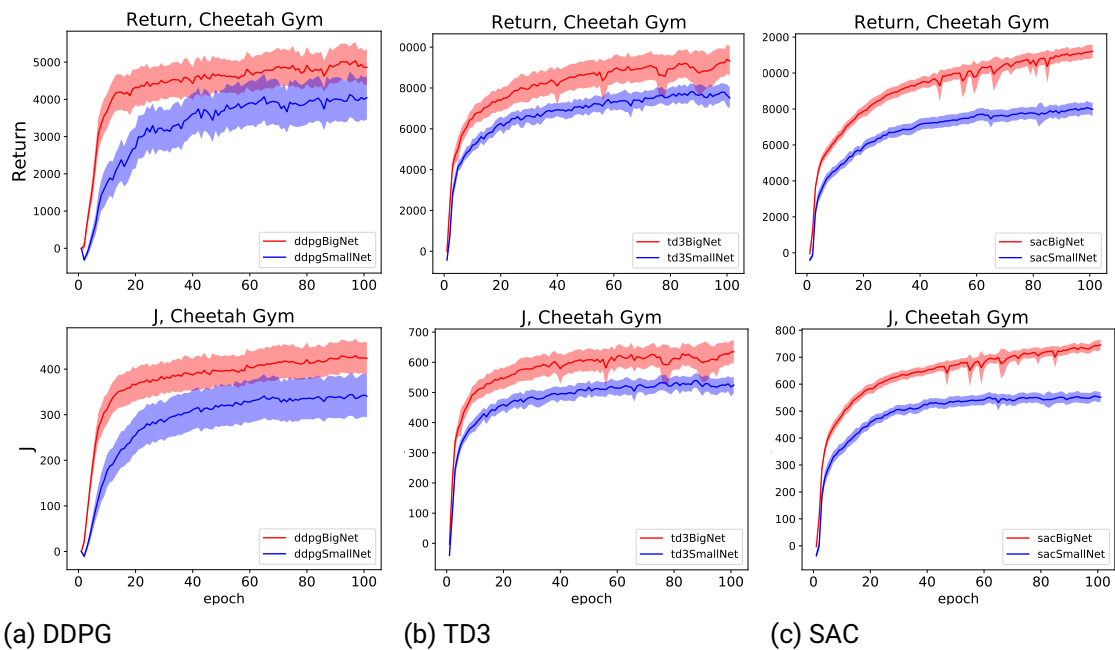


Figure 4.33: Comparison of small and big networks on the Cheetah environment averaged over 25 runs with 95% confidence bounds.

The biggest differences in the results of the two network architectures can be observed on

the Gym **Ant** environment. While nearly no learning behavior is achieved for neither of the three algorithms when using small networks, the bigger setup leads to decent results in terms of  $J$  and  $R$  (see Figure 4.34). The small networks seem to not be able to hold enough information to extract the properties of the observed states to the full extent. This can be attributed to the fact that the observation space dimensionality of the Ant environment (111) is decisively larger than the one of Hopper or Cheetah (11 and 17 respectively, see Fig. 2.1).

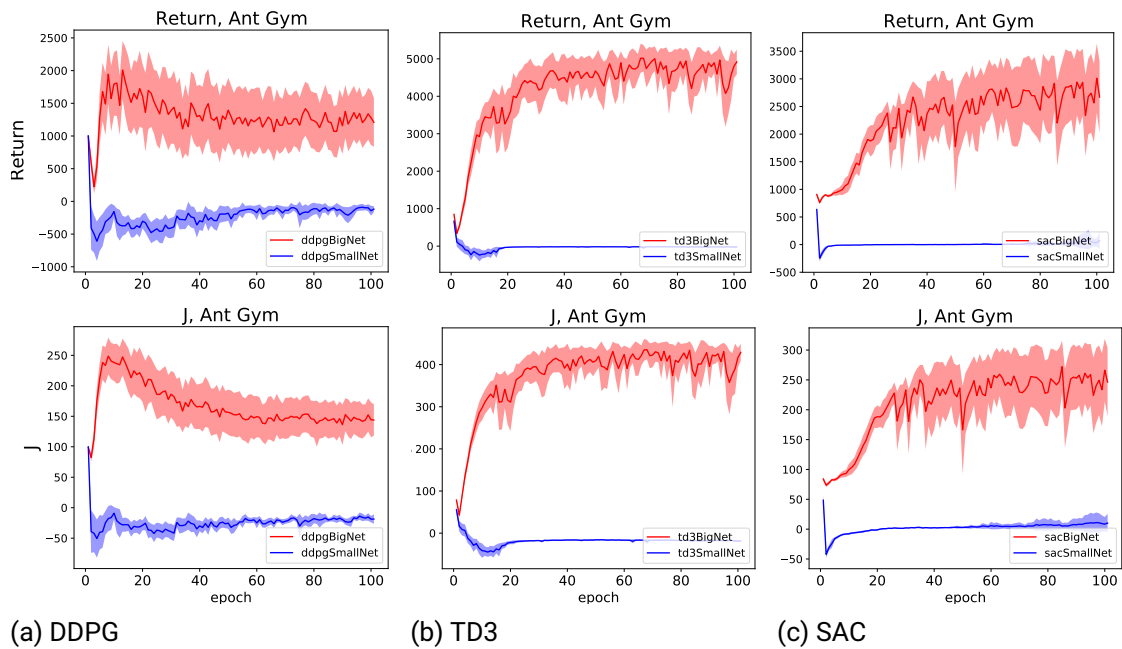


Figure 4.34: Comparison of small and big networks on the Ant environment averaged over 25 runs with 95% confidence bounds.

---

## 5 Discussion and Outlook

---

Coming back to the general benchmark results in Section 4.1, we can draw several conclusions.

SAC generally seems to be the best performing algorithm across the board both in terms of initial learning speed and also in terms of the final return values at the later epochs. For some environments (e.g. DeepMind Hopper, see Fig. 4.2), it even turns out to be the only algorithm that shows any relevant learning behavior. In addition to that, the learning is very stable in most cases without big variances in the observed results. An exception to this property is formed by the Gym Ant environment where SAC performs comparatively weaker and has higher variance than it normally does.

TD3 also achieves good results on most environments and even outperforms SAC in some cases (e.g. Gym Ant, see Fig. 4.16). However, it usually is less stable and more variant in its results (e.g. Gym Hopper, cf. Fig. 4.2).

The remaining off-policy algorithm, DDPG, achieves significantly worse results than its two successor algorithms. Due to not making use of the Double Q-Learning approach, it often overestimates the Q-values and, for tasks with unstable dynamics, like Hopper, the value estimates can even completely explode (see Gym and PyBullet Hopper in Figure 4.4). This behavior results in very low returns or no learning at all (e.g. Gym Walker, see Fig. 4.7). DDPG also generally has very large confidence intervals for the returns, which means that the outcome of single runs can be significantly better or worse depending on the random seed. However, on environments with more stable dynamics, like Gym Cheetah, the performance looks a lot better and much more stable (see Fig. 4.12).

TRPO, PPO1 and PPO2 achieve close results on most of the environments. The specific outcomes vary between the different tasks and the chosen library. While TRPO outperforms both PPO variants on PyBullet Walker, it scores weaker on the PyBullet versions of Cheetah and Ant. PPO1 overall achieves better results than its counterpart PPO2. For environments like DeepMind Walker, the outcomes between the two variants look exceedingly different (see Fig. 4.7). Because of the fact that we use a large amount of independent runs, we can consider this a meaningful result which shows that the specific implementation of an

---

---

algorithm can have a big effect in how it actually performs in practice.

The results of the on-policy algorithms generally seem to have less variance than the ones of the off-policy algorithms, which makes them more stable. However, in most cases, they can not keep up with the ones of SAC and TD3 in terms of peak performance. Especially A2C performs comparatively weakly in nearly all environments, which is expected due to its simplicity and the lack of the sophisticated improvements the other algorithms make use of.

Looking at the evaluation metrics we can see that the undiscounted return is closely related to the discounted return in most cases and shows similar trends. There are some exceptions, though, where improvements in terms of  $J$  do not entail rising undiscounted return values (e.g. Gym Hopper, see Fig. 4.2). This behavior shows that reporting results for both return formulations can be advantageous and can highlight particularities that would not become apparent if only the undiscounted return was measured. The large confidence intervals of certain results also show that it is utterly important to perform enough independent runs for the same experiment in order to counteract the randomness in the algorithms and environments. Otherwise, the results can become very misleading as they might only represent a lucky or unlucky set of outcomes. This problem becomes even more severe if metrics like the Maximum Return or Maximum Average Return are used instead of the Average Return.

Our reported results are wildly varying depending on the dynamics and reward functions of the specific tasks. We experience that the Hopper and Walker tasks behave similarly to a certain degree while the evaluations on Cheetah and Ant look entirely different for some algorithms. The plethora of exceptions in the algorithm performances on some tasks show that it is important to cover a wide range of benchmark tasks instead of only reporting results on single environments. The second approach becomes even worse if the algorithm hyperparameters are specifically tuned for the chosen environment and the algorithm is, therefore, overfitting on this environment, which makes the results even less suited for serving as a meaningful benchmark.

Furthermore, even if several different tasks are chosen for the benchmarks, it is important, which exact benchmark library provides the implementation of the task. We show that the algorithm performances can turn out entirely different depending on the specific environment implementation. As an example, we can reference the results of TD3 on the Hopper task. While good returns can be achieved on the OpenAI Gym and PyBullet variants of Hopper, this is not the case for the DeepMind implementation where nearly no learning behavior can be observed (see Fig. 4.2). Choosing a library that has an implementation which leads to better results on a chosen task, can also make results misleading. In general, the DeepMind environments seem to require more exploration

---

---

and are also harder to solve overall, although the state and action space dimensionalities are not necessarily bigger. Thus, the complexity of an environment does not only lie in the size of its state and action spaces, but, more importantly, in the exact dynamics and the reward function.

The comparison of normalized and non-normalized inputs in Section 4.2 shows that no clear statements can be made whether one or the other is generally preferable. While input normalization increases the performance on most DeepMind environments, this is not the case for Gym and PyBullet. For example, we can observe worse results for nearly all algorithms on the Gym and PyBullet versions of the Ant task when using normalized inputs. In contrast to that, the Walker and Cheetah environments mostly seem to benefit from input normalization. Therefore, we conclude that this method has to be tested and adjusted individually for each environment. For most of our experiments, normalized inputs increased the variance of the results, which might be a drawback.

The network size seems to have large effects on the performance of DDPG, TD3 and SAC (see Section 4.3). Bigger networks lead to both faster initial learning speed as well as higher peak returns. This improvement is present for all used environments, but most extreme for the Ant task where nearly no learning behavior can be observed when using smaller networks. In return, the training time of the algorithms is considerably increased, which might be a problem for certain use cases.

In this thesis, we perform an extensive set of experiments on a suite of established continuous control environments using the implementations of the three biggest benchmark libraries. We provide results for six of the most well-known model-free deep RL algorithms and also examine the impact of input normalization as well as bigger network architectures. Our experiments are carried out in a reproducible way by reporting all used hyperparameters as well as by providing details about the experiment setup and execution. In addition to that, we follow the suggestions of averaging our results over many trials, displaying the resulting confidence bounds and using meaningful, non-biased evaluation metrics. We point out similarities and differences in the algorithm performances and compare how the different environment implementations impact the results by highlighting peculiar results. Furthermore, we point out potential pitfalls in the benchmarking process and in the way of reporting results.

Our results show that the tuning for the environments is not equivalent, although this behavior would be expected since they are all used to benchmark the suitability of algorithms for the domain of continuous control. Instead, the performances vary from environment to environment and are highly susceptible to favourable random seeds in addition to that.

---

---


While the OpenAI Gym and PyBullet environments have been commonly used for benchmarks in related work, the environments of the DeepMind Control Suite have not been thoroughly evaluated on all common model-free RL algorithms yet. To our knowledge, we are the first to provide various results for these environments with a sufficient amount of independent runs and meaningful evaluation metrics that are presented directly together with the results for the other two benchmark libraries to enable easy comparisons. We also do not only measure the performance in terms of average undiscounted return as it is normally done, but instead also show the discounted returns, the episode lengths, the value function estimates and the entropy of the policies over the course of the training, which enables a deeper insight into the learning process.

Starting from the outcomes of our analysis, several research directions that tackle the observed problems are possible. New approaches could range from methods that lead to even more accurate value estimates (building on the Double Q method of TD3 and SAC) and ideas that explicitly control the amount of exploration to further ways of avoiding performance decreases.

Similar to our examinations regarding input normalization and network size, further experiments that explore the robustness to changes of other hyperparameters could be performed. For example, we do not evaluate the impact of other noise types like time-correlated Ornstein-Uhlenbeck noise ([40]) or parameter space noise ([41]). Investigations regarding the reward scale and layer normalization have been performed in [22], but not for the Ant and Walker task and also only using the OpenAI Gym library. Similarly, [21] looked at the impact of different batch sizes and different activation functions, but not for all algorithms and also not using all three task libraries. Furthermore, we only evaluate the six most common model-free RL algorithms. The same experiments could be performed using the state-of-the-art model-based algorithms to measure the applicability of those approaches in this field. [52] already did an extensive study for model-based algorithms, which could be extended to include results for the PyBullet and DeepMind environments.

One could also, instead of trying to find an algorithm with the overall best performance across a suite of benchmark tasks, try to identify which exact criteria of the environments makes algorithms perform better or worse. By doing so, we could specify specific application criteria for the algorithms and make an individual decision for each use case.

Since sample efficiency is an important criterion for real world robotic and continuous control tasks as samples are usually expensive to get, we plot our results in terms of training samples obtained from the environment. However, for certain use cases, it might be useful to consider the computational complexity of the different algorithms and measure the



---

performance with respect to the resulting training time efficiency. This approach might be a starting point for future work.



---

## Bibliography

---

- [1] S. Tunyasuvunakool, A. Muldal, Y. Doron, S. Liu, S. Bohez, J. Merel, T. Erez, T. Lillicrap, N. Heess, and Y. Tassa, “dm\_control: Software and tasks for continuous control,” *Software Impacts*, vol. 6, p. 100022, Nov 2020.
- [2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.
- [3] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.
- [4] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural Networks*, vol. 61, p. 85–117, Jan 2015.
- [5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [6] M. Riedmiller, “Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method,” in *Machine Learning: ECML 2005* (J. Gama, R. Camacho, P. B. Brazdil, A. M. Jorge, and L. Torgo, eds.), (Berlin, Heidelberg), pp. 317–328, Springer Berlin Heidelberg, 2005.
- [7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” 2013.
- [8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–489, 2016.

- 
- 
- [9] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, “Mastering the game of go without human knowledge,” *Nature*, vol. 550, pp. 354–359, 2017.
- [10] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, “Starcraft ii: A new challenge for reinforcement learning,” 2017.
- [11] V. do Nascimento Silva and L. Chaimowicz, “Moba: a new arena for game ai,” 2017.
- [12] 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.
- [13] J. Peters and S. Schaal, “Policy gradient methods for robotics,” in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2219–2225, 2006.
- [14] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen, “Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection,” 2016.
- [15] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. Riedmiller, and D. Silver, “Emergence of locomotion behaviours in rich environments,” 2017.
- [16] A. Rajeswaran, V. Kumar, A. Gupta, G. Vezzani, J. Schulman, E. Todorov, and S. Levine, “Learning complex dexterous manipulation with deep reinforcement learning and demonstrations,” 2018.
- [17] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel, “Asymmetric actor critic for image-based robot learning,” 2017.
- [18] S. Levine, C. Finn, T. Darrell, and P. Abbeel, “End-to-end training of deep visuomotor policies,” 2016.
- [19] 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, p. 253–279, Jun 2013.
- [20] Y. Duan, X. Chen, R. Houthoof, J. Schulman, and P. Abbeel, “Benchmarking deep reinforcement learning for continuous control,” 2016.

- 
- 
- [21] R. Islam, P. Henderson, M. Gomrokchi, and D. Precup, “Reproducibility of benchmarked deep reinforcement learning tasks for continuous control,” 2017.
- [22] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, “Deep reinforcement learning that matters,” 2019.
- [23] S. Whiteson, B. Tanner, M. E. Taylor, and P. Stone, “Protecting against evaluation overfitting in empirical reinforcement learning,” in *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pp. 120–127, 2011.
- [24] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” 2016.
- [25] E. Coumans and Y. Bai, “Pybullet, a python module for physics simulation for games, robotics and machine learning.” <http://pybullet.org>, 2016–2019.
- [26] R. S. Sutton and A. Barto, “Reinforcement learning : an introduction,” 2018.
- [27] J. Achiam, “Spinning Up in Deep Reinforcement Learning,” 2018.
- [28] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” 2016.
- [29] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust region policy optimization,” 2017.
- [30] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017.
- [31] J. Fu and I. Hsu, “Model-based reinforcement learning for playing atari games,” tech. rep., Technical Report, Stanford University, 2016.
- [32] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
- [33] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Mach. Learn.*, vol. 8, p. 229–256, May 1992.
- [34] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, “Stable baselines.” <https://github.com/hill-a/stable-baselines>, 2018.

- 
- 
- [35] 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.
- [36] M. P. Forum, “Mpi: A message-passing interface standard,” tech. rep., USA, 1994.
- [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,” 2019.
- [38] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic policy gradient algorithms,” in *ICML*, 2014.
- [39] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” 2013.
- [40] S. Finch, “Ornstein-uhlenbeck process,” 2004.
- [41] M. Plappert, R. Houthoofd, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz, “Parameter space noise for exploration,” 2018.
- [42] S. Fujimoto, H. van Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” 2018.
- [43] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” 2018.
- [44] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. de Las Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, T. Lillicrap, and M. Riedmiller, “Deepmind control suite,” 2018.
- [45] 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, 2012.
- [46] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, “Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation,” 2017.
- [47] Student, “The probable error of a mean,” *Biometrika*, pp. 1–25, 1908.
- [48] A. Raffin, “Rl baselines zoo.” <https://github.com/araffin/rl-baselines-zoo>, 2018.
- [49] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *ICML*, 2010.

- 
- 
- [50] S. Gu, T. Lillicrap, Z. Ghahramani, R. E. Turner, and S. Levine, “Q-prop: Sample-efficient policy gradient with an off-policy critic,” 2017.
  - [51] H. van Hasselt, A. Guez, M. Hessel, V. Mnih, and D. Silver, “Learning values across many orders of magnitude,” 2016.
  - [52] T. Wang, X. Bao, I. Clavera, J. Hoang, Y. Wen, E. Langlois, S. Zhang, G. Zhang, P. Abbeel, and J. Ba, “Benchmarking model-based reinforcement learning,” 2019.