FRANKFURT UNIVERSITY OF APPLIED SCIENCES

MASTER THESIS

Deep Reinforcement Learning: Overcoming the Challenges of Deep Learning in Discrete and Continuous Markov Decision Processes

Author:
Marius WICHTNER
wichtner@stud.fra-uas.de
1051472

Examiner:
Prof. Dr. Thomas GABEL
Co-examiner:
Prof. Dr. Christian BAUN

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

October 23, 2019

Declaration of Authorship

I, Marius WICHTNER, declare that this thesis titled, "Deep Reinforcement Learning: Overcoming the Challenges of Deep Learning in Discrete and Continuous Markov Decision Processes" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Abstract

Faculty of Computer Science and Engineering

Master of Science

Deep Reinforcement Learning: Overcoming the Challenges of Deep Learning in Discrete and Continuous Markov Decision Processes

by Marius WICHTNER

In the last couple of years, the reinforcement learning research community successfully picked up the utilization of deep neural networks. While the application of non-linear function approximation in reinforcement learning has been known for quite a while, the relevant methods remained relatively obscure and were unstable for a variety of domains. Recent research tries to understand the black box of deep neural networks in the context of reinforcement learning. As a result, a variety of improvements were developed to guarantee a stable and reliable learning process in high-dimensional reinforcement learning environments. These improvements enabled the extension of reinforcement learning from trivial problems to much more complicated and exciting difficulties, such as the very high-dimensional domain of robotics. Unfortunately, these algorithms have become incredibly complicated. Many of the novel methods require meticulous hyper-parameter tuning, which made the development of such notoriously hard. To sustain the recent progress in reinforcement learning, it is consequently essential to provide reproducibility and to preserve a vital degree of transparency, all while striving for algorithm simplicity. To that end, this thesis reviews the challenges of deep reinforcement learning theoretically and analyzes reproducibility and important algorithm details empirically. It explores the difficulties of combining reinforcement learning with deep learning and discusses the latest satisfactory solutions. Most importantly, it sheds new light on seemingly unimportant hyper-parameters and algorithm features, while providing unique insights and clarity with an attention to detail.

Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

A2C	**A**dvantage **A**ctor-**C**ritic
AI	**A**rtificial **I**ntelligence
ACER	**A**ctor **C**ritic with **E**xperience **R**eplay
ACKTR	**A**ctor **C**ritic using **K**ronecker-Factored **T**rust **R**egions
ANN	**A**rtificial **N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
DQN	**D**eep **Q** **N**etworks
DP	**D**ynamic **P**rogramming
DPG	**D**eterministic **P**olicy **G**radients
DDPG	**D**eep **D**eterministic **P**olicy **G**radients
DDQN	**D**ouble **D**eep **Q** **N**etworks
DDDQN	**D**ueling **D**ouble **D**eep Q **N**etworks
EMA	**E**xponential **M**oving **A**verage
ELU	**E**xponential **L**inear **U**nit
FQI	**F**itted **Q** **I**teration
GAE	**G**eneralized **A**dvantage **E**stimator
KL	**K**ullback-**L**eibler
MC	**M**onte **C**arlo
MDP	**M**arkov **D**ecision **P**rocess
MuJoCo	**Mu**lti-**Jo**int dynamics with **Co**ntact
MM	**M**inorization **M**aximization
NFQ	**N**eural **F**itted **Q** Iteration
NFQCA	**N**eural **F**itted **Q** Iteration with **C**ontinuous **A**ctions
PPO	**P**roximal **P**olicy **O**ptimization
PG	**P**olicy **G**radient
RL	**R**einforcement **L**earning
ReLU	**Re**ctifier **L**inear **U**nits
SGD	**S**tochastic (mini-batch) **G**radient **D**escent
TD	**T**emporal-**D**ifference
TRPO	**T**rust **R**egion **P**olicy **O**ptimization
TRM	**T**rust **R**egion **M**ethods
VE	Proportion of **V**ariance **E**xplained

Chapter 1

INTRODUCTION

learn·ing /ˈləːnɪŋ/ *noun*
1: *the acquisition of knowledge or skills through study, experience, or being taught*
2: *modification of a behavioral tendency by experience (such as exposure to conditioning)*
3: *an increase, through experience, of problem-solving ability*

The *sample complexity* of the human brain is truly remarkable. A human being is capable of learning lifelong behavior from only a single experience. In comparison, today's *machine learning* algorithms are horribly sample inefficient. They often require millions of carefully selected learning samples before any sophisticated behavior may be learned. Meanwhile, the increase of computational power seems inexorable, and traditional machine learning approaches currently experience a new heyday, called *deep learning*, promising more applicable and better-performing data science solutions. *Reinforcement learning*, a sub-field of machine learning, appears to benefit in particular from this new trend. Figure 1.1 shows how quickly reinforcement learning jumped on the success train of deep learning, establishing the topic of this thesis: *Deep Reinforcement Learning*.

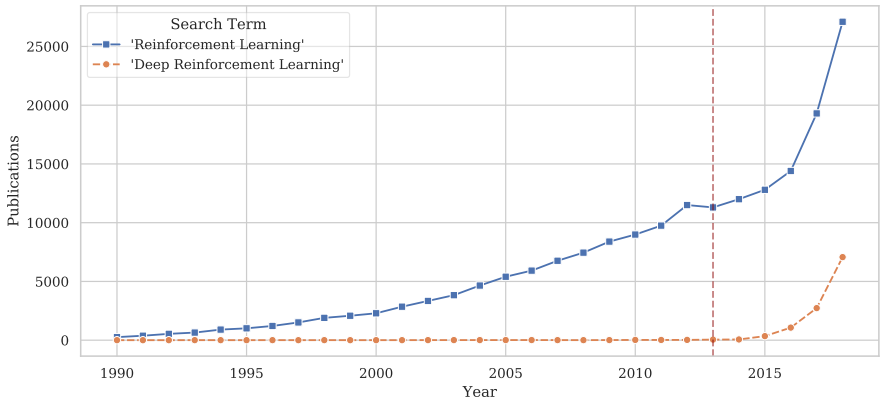


FIGURE 1.1: We see the total number of publications (y-axis) found for the exact search terms "Reinforcement Learning" and "Deep Reinforcement Learning" by year (x-axis), scraped from *Google Scholar*. The vertical line in the year 2013 marks the starting point of publications, which this thesis is interested in. Notice the explosive increase of publications beyond this line.

1.1 Reinforcement Learning

Reinforcement learning (RL) [114] is an area of machine learning in which an interactive algorithm, formally called *agent*, [114] interacts with an *environment*. The agent must find the correct set of interactions with the environment to solve a formalized problem. In RL, the agent, therefore, must learn a *policy* that describes which action the agent performs in the environment. Decisions are made based on a scalar reward. At each timestep, the agent performs an action and observes its successor state and reward, given in response to the selected action. Throughout this iterative trial-and-error process, the agent seeks to find a preferably *optimal return* of the system's dynamics over time [8]. In biology, a reward *reinforces* the action that caused its delivery [20]. For human beings or animals, positive or negative consequences of an action influence future behavior. It has been shown that there are notable similarities between the phasic signals of the dopaminergic neurons in the prefrontal cortex [107] of the human brain and reinforcement learning [75]. Motivated by this idea, researchers composed a set of algorithms to learn behavior from a given reward signal.

1.2 Deep Learning

Conventional machine learning was previously limited in the ability to process raw, untransformed data. For many years painstakingly handcrafted feature representations were required to train a *classifier* or to perform *non-linear regression*. *Deep learning* [63] enables machine learning models, composed out of multiple processing layers, to learn *underlying functional correlations* of complicated data representations. Such a model is referred to as a *deep neural network*. Each computational layer transforms its input data into a higher and more abstract representation, providing the ability to handle previously unseen data by *generalization*. Given the composition of an adequate amount of such transformations, complex functional relations can be learned. Consider, for example, the image classifier *ResNet* [43], which was trained with 152 layers and 60.2×10^6 free parameters to classify a wide variety of different images of the *ImageNet* [58] data set. It is remarkable how quickly deep learning progressed [63], given that traditional wisdom of model complexity [78] is not able to explain the generalization ability of large non-linear function approximations. As a result, it requires rethinking [131] and new theoretical constructions to understand this current trend of machine learning thoroughly.

1.3 Deep Reinforcement Learning

The term deep reinforcement learning describes the combination of *supervised learning* and *reinforcement learning*, as illustrated in Figure 1.2. The idea of such a combination was not entirely new when the term emerged with the increasing popularity of *deep learning*. *TD-Gammon* [117] was an early contender for the first successful interdisciplinary combination of non-linear function approximation and RL. The algorithm managed to learn the game *backgammon* by self-play and explored unforeseen unorthodox strategies that had a significant impact on the way professional human players play today. Interestingly enough, *TD-Gammon* has not initiated a comparable increase

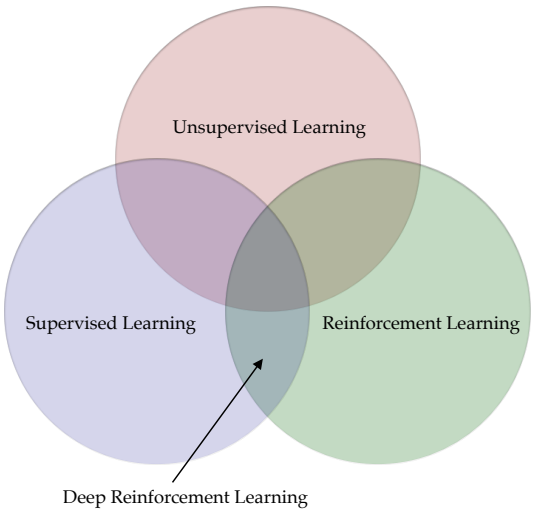


FIGURE 1.2: Overlaps of different types of machine learning.

of deep RL publications as we find it nowadays in the research community. Even influential RL textbooks, such as Sutton and Bartos "*Reinforcement learning : an introduction*" [113], have not dedicated much attention to non-linear function approximation at that point in time. The application of such remained rather obscure, and the success of TD-Gammon had not been very well understood [89]. Later, after the result of several deep learning breakthroughs, the research community has picked up the idea of deep reinforcement learning once again and achieved astonishing results with it. A reader may now ask the question: "How has recent research managed to overcome the challenges of deep learning?". The answer to this question can be seen as the central underlying *leitmotiv* of this thesis.

1.4 Motivation

RL has made significant progress in many applicable problem domains. While traditional RL was previously limited to simple toy problems, the application of deep learning has enabled a wide variety of complex sequential decision-making tasks. Probably most noteworthy are the results in video games such as Atari [75], StarCraft [121], and Dota2 [83], board games like Go [111] and Chess [110], as well as robotic control tasks [84, 105, 68].

Despite the different success stories, it became difficult to understand which parts of deep RL matter, and which do not [46, 130, 1, 91]. For larger projects, as mentioned above, the results are produced by large numbered teams on costly cloud environments. The source code of theses projects is usually kept private, which makes it hard to reproduce or understand the success from a technical perspective thoroughly. Also, accessible publicly available repositories, released alongside their corresponding white-paper publications, remain difficult to understand and to reproduce. In many cases, both intrinsic (environment hyper-parameters, random seeds), as well as extrinsic reasons

(codebase changes, hyper-parameters) for non-determinism, can contribute to inconclusive results of the different deep RL baseline algorithms even with their original repositories [46].

In the domain of software engineering, the term *technical debt* [27] refers to the implied cost of additional rework by choosing a software implementation built with a lack of knowledge, standards, collaboration, and documentation. Similar to a *monetary debt*, technical depth becomes in particular dangerous when the level of technical complexity surpasses that of the domain foundation, and we begin losing compatibility to further changes. One may transfer the same metaphor into the domain of deep reinforcement learning: If we do not conduct the necessary bookkeeping, such as maintaining standard-benchmarks, providing full algorithm insights, and ensuring reproducibility, we will very likely not be able to sustain the rapid development speed of the current state of research. To that end, this thesis contributes as a literature review by providing *transparency* and by counteracting against the arising technical debt of reinforcement learning.

1.5 Objectives

The utilization of deep neural networks in the domain of reinforcement learning opens up a new window of possible research. In this window, innovative empirical successes have significantly increased the interest of reinforcement learning research, as we may interpret from Figure 1.1. Concurrently, for many of these empirical successes, it has become unclear why these approaches work so well in practice. On top of that, these new methods often only show fragile convergence properties, require meticulous hyper-parameter tuning, and are hard to reproduce. These challenges limit the quick pace of the development of new algorithms. In this thesis, we resist these emerging concerns with three central objectives:

First, the central underlying theoretical objective of this thesis is to explore the challenges of deep learning in the context of discrete and continuous RL domains in more depth. We discuss poorly but also well-understood challenges along with possible solutions, identify gaps in the current literature, but, most importantly, provide a robust layer of transparency for the complex construct of deep reinforcement learning.
Second, we argue that only little research [46] has been conducted to investigate the impact of recently emerged hyper-parameters. Hence, we experiment with the magnitude of the effect of such. On top of that, this work investigates if the attention given to specific hyper-parameters in the current literature is adequate, and performs experiments on a currently state-of-the-art deep reinforcement learning algorithm to review the reproducibility and clearness of recent RL methods.
Third, we aim to research the influence of the neural network size on algorithm performance and explore the interrelation to the learning rate of the model. The neural network has been a controversial [131, 66] research topic in supervised learning. However, we find that too little research [46, 41] has been conducted in the context of deep reinforcement learning, even though the utilization of deep learning methods in RL differs considerably [75, 41], and the size of the neural network constitutes a frequently

recalled difficulty in many deep RL publications [130, 132, 41, 100, 46].

1.6 Outline

This thesis consists of five chapters, followed by two supplementary appendixes. The topic of deep reinforcement learning mainly revolves around two major categories: *Value-based* and *policy gradient-based* RL. After the current customary introduction chapter, we establish the theory of these two divisions in Chapters 2 and 3. Each of these theoretical chapters give the reader the necessary foundations, discuss the central difficulties of combining deep learning with reinforcement learning, and establish novel solutions provided by the literature. Chapter 4 supports the theoretically acquired properties of deep reinforcement learning in two practical collections of experiments. Finally, Chapter 5 closes this thesis by revisiting the central research questions, gives an indication for future work, and summarizes the core contributions of this work. Additionally, the supplementary material provides a mathematical encyclopedia in Appendix A and discloses a full set of experiments in Appendix B. Figure 1.3 represents a graphical overview of this thesis.

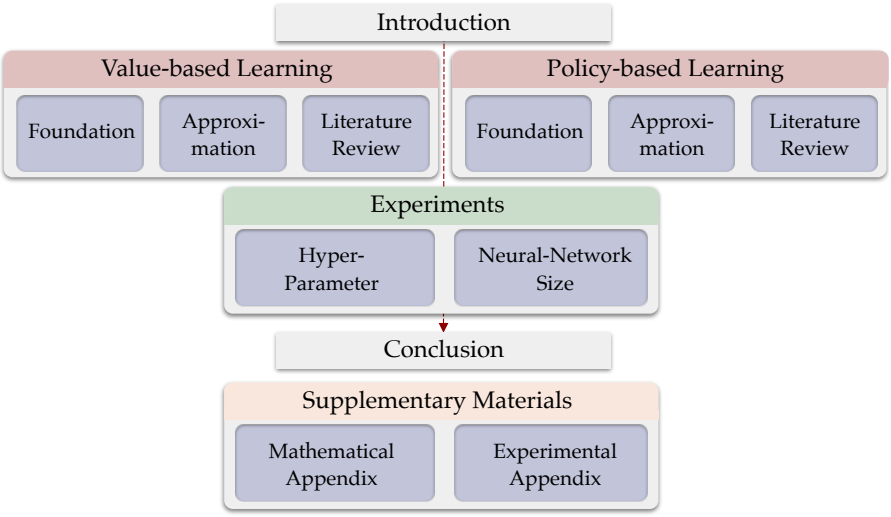


FIGURE 1.3: The big picture of this thesis.

Chapter 2

VALUE FUNCTION-BASED APPROACHES

The ability to learn a sophisticated policy from a noisy scalar reward constitutes a challenge in itself. A standard approach to deal with this problem was developed many years ago. The essential idea behind such a solution was to learn a function that represents *experience* by mapping a representation of the environment-state $\mathbb{S}$ to a *value* representation:

$$V : \mathbb{S} \rightarrow \mathbb{R} \tag{2.1}$$

and then derive a sophisticated policy from it. What Richard Bellman once cautiously called a "theory" [7] has meanwhile developed into one of the most important milestones of future artificial intelligence (AI): *Value Function-Based Reinforcement Learning*.

Throughout this chapter, we explore different forms of value function representations and understand how to derive an optimal decision policy from them. Policy gradient-based RL has been developed on top of the framework of value-based RL. We, consequently, use Section 2.1 as an opportunity to introduce the basic terminology and foundation of traditional RL. It would carry things too far, saying that this collection of foundations is complete; however, it covers the essential parts of RL to understand any subsequent work of this thesis. Section 2.2 is concerned with the approximation of the value function using supervised machine learning approaches, while elaborating challenges and appropriate solutions. Finally, a literature review will be provided to offer the reader a more comprehensive overview of the latest value-based deep RL methods.

2.1 Foundations

In this section, we cover the foundations of value-based RL. We start by introducing the basic terminology of traditional reinforcement learning theory and then establish essential *model-based* and *model-free* value-based methods.

2.1.1 Markov Decision Process

A Markov Decision Process (MDP) [90] is a classical formalization of a sequential decision-making process. Consider a sequence of multiple states in which a formal agent [114] can interact with the environment. We modeled such an interaction in Figure 2.1. The interaction between the agent and the environment is an iterative process. After each interaction, the global state of the environment changes. Observations can be *non-deterministic*. Say, an agent has performed an action a in a state s, it can observe the successor state s'_a with a reward r_a. If an environment behaves non-deterministic, it

may happen that the agent executing the same action a in a state s will result in a different successor state s'_b with a reward r_b.

> Note that both, cost and reward, are equivalently used in literature to express the same: $-reward \equiv cost$. An optimal reward is maximal, whereas an optimal cost is minimal.

An agent must, therefore, observe multiple times to estimate the non-deterministic part of the environment state. [114]

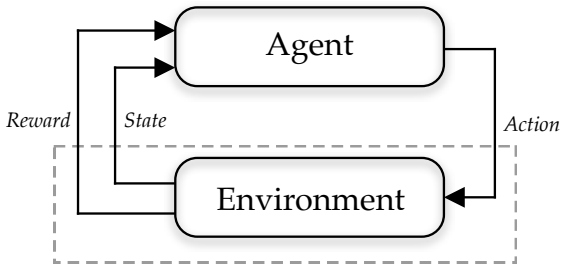


FIGURE 2.1: The interaction between the agent and the environment.

An MDP may be formalized as a 4-tuple [94, 90]: $(\mathbb{S}, \mathbb{A}, p, R)$. Let $\mathbb{S}$ be the space of possible states and $\mathbb{A}$ be the space of possible agent actions, such that a state transition function $p_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$, which maps $p_a : \mathbb{S} \times \mathbb{A} \times \mathbb{S} \rightarrow [0, 1]$, denotes the probability of ending in a successor state $s' \in \mathbb{S}$ when taking an action $a \in \mathbb{A}$ in state $s \in \mathbb{S}$. The reward transition function $R : \mathbb{S} \times \mathbb{A} \times \mathbb{S} \rightarrow \mathbb{R}$ describes the arising deterministic reward when moving from one state s to a successor state s' [90]. In order to achieve its purpose, an agent must maximize its expected long-term reward in an environment. Thus it must learn or improve a decision policy [90] $\pi : \mathbb{S} \rightarrow \mathbb{A}$, a deterministic function or stochastic probability distribution that maps the current state to a set of possible actions [94, cf.]. An example MDP formalization can be found in Figure 2.2.

2.1.2 Learning with the Bellman equation

An essential idea behind value function-based reinforcement learning is to estimate the state value function $V(s)$ in an iterative update [76]. In reinforcement learning, the *Bellman equation* [7] can be used to estimate the value function V in order to find an optimal strategy π^*. The *Bellman Principle of Optimality* [7] claims that if an agent has to perform k steps, the optimal reward for a state i is defined by the sum of the direct transition return and the optimal trajectory reward of the following state when, from there, the remaining amount of steps to be taken is: $k - 1$ [114, p.71]. Based on the existence of a Markov decision process, the principle, therefore, claims that the value of a single state can express the values of other states. The Bellman equation enabled a variety of iterative algorithms to calculate a value function, as we will see later on.

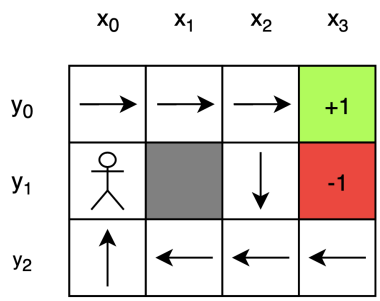


FIGURE 2.2: A classical Markovian grid-world environment, in which an agent can perform an action $a \in \mathbb{A} = \{a_{up}, a_{down}, a_{right}, a_{left}\}$ in order to change its state $s = (x_n, y_p)$. The green field indicates a successful terminal state with a reward of $+1$, while the red field indicates a negative terminal state with a reward of -1. We speak of a *dense* reward function if direct transfer costs occur (say, 0.1), and consider the reward to be *sparse* if rewards are experienced only in terminal states. An optimal decision policy π^* for a *dense* reward setting is visualized by the arrows.

2.1.3 Model

Reinforcement learning differentiates between model-free and model-based environments. In an environment with a known model, the agent is in knowledge or partial knowledge of the underlying dynamics of its environment. It will know the successor states of a current state when executing a specific action as well as their probabilities and their expected rewards before experiencing it. In a model-free environment, the agent has to experience the available successor states, their probability, and the expected rewards first. The agent primarily navigates through the environment by "trial-and-error" [114, p.7]. The exploitation of environment domain knowledge can significantly increase the learning performance of the algorithm [25, 36]. However, for many practical RL applications, this may be impossible, especially if the cost of simulation is high or the dynamics of the environment are utterly unknown. Model-free methods are discussed further in Section 2.1.10. As this work progresses, we will increasingly concentrate on model-free learning approaches.

2.1.4 Deterministic Environments

Generally, we must differentiate between two types of environments: Deterministic and stochastic environments [114]. Deterministic environments guarantee to change from one state s to another state s' when performing a particular action a. In stochastic environments, the state transition is connected with a specific probability distribution. This distribution can either describe noise in the environment or may be a defined probability of a stochastic part in the environment. We can, hence, define an MDP *deterministic* if the reward function R and the state transition function p are defined *uniquely* for every state s and action a [33]. Stochastic environments do not necessarily limit the agent from improving its policy as the underlying stochastic model of expected probabilities

converges to *equilibrium* [90], given an infinite state visitation frequency. This effect allows the agent to, nevertheless, maximize its performance by following the maximal expected cumulative reward of the environment.

2.1.5 Offline and Online Learning

In the context of *machine learning*, offline/online learning [114] describes the point in time when learning is executed. *Online learning* describes the process of learning while collecting training data from experiences, meaning the policy π is updated after a new sample from the environment is collected. *Offline learning*, in contrast, is accomplished after all data has been collected from a learning trajectory. Offline learning is considered to be slower, as the agent cannot learn from experiences while moving through the trajectory. Thus, the more time an agent requires for a trajectory, the less effective becomes offline learning.

2.1.6 Horizon

An optimal agent behavior will improve its performance. In order to judge *optimality*, we adopt two types [85] of trajectory horizon models, which are used by the majority of RL publications. In a *finite-horizon* model, an agent optimizes its expected reward r_t at any timestep t for the next n timesteps: $\mathbb{E}[\sum_{t=0}^{n} r_t]$. With any non-stationary policy, an agent will always choose its *(n-th)-step optimal action* until a *terminal state* has been reached. If the exact length of the trajectory horizon is unknown, a better horizon-model may be the *discounted infinite-horizon model*: $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]$, in which the discount factor $\gamma \in [0, 1)$ is a hyper-parameter. γ can be interpreted [85] in several ways: As a mathematical trick to analyze optimality, to discount later rewards indifferent from earlier rewards or as a variance reduction technique, which we shall see later in Section 3.1.5.

2.1.7 Dynamic Programming

The term Dynamic Programming (DP) [7] describes a collection of reinforcement learning algorithms that can be applied to environments with a perfect model. They come with a substantial computational expense; however, they provide an essential foundation for most value- and policy-gradient-based RL methods. Prior to more practically applicable value-based reinforcement learning, we shall, thus, briefly cover a primary DP method: *Policy Iteration* [33]. The general idea behind dynamic programming is to break down a complicated problem into multiple simplified sub-problems; and then to search for the *optimal substructure* within the space of already solved sub-problems. To that end, we can calculate a sequence of length k value functions $V_1, V_1, ..., V_k$ in an iterative manner, where each function $V_k(s) \rightarrow \mathbb{R}$ maps each state s of the environment at timestep k of the calculation into a real-valued quality of state measure [7]. The optimal solution of the complicated problem is then found by recursively solving the *Bellman equation* [7]:

$$V^*(s) = \max_{a \in \mathbb{A}} \sum_{s', r} p_a(s, s')(r + \gamma V^*(s')), \tag{2.2}$$

where $s \in \mathbb{S}, s' \in \mathbb{S}$, $r \in \mathbb{R}$, $a \in \mathbb{A}$ and $\gamma \in [0,1)$ is the discount factor. When using a tabular form for the value function, we store a value for each state in an expedient memory-structure. The optimal policy can then be derived after an optimal value function V^* has been found, satisfying the Bellman equation.

Policy Iteration

Policy Iteration grasps the way to the optimal solution as a monotonically improving process, in which the algorithm iterates between *Policy Evaluation* (1) and *Policy Improvement* (2). Firstly, we are concerned with the calculation of the next value function (1) $V' = V_{k+1}$, where a given current value function under the current policy $V = V_k$ may be chosen arbitrarily at the state $k = 0$. By applying the *Bellman equation* as the update rule, we calculate [114] preceding value functions recursively:

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} p_a(s,s')(R(s,a,s') + \gamma V_k(s')). \tag{2.3}$$

Here, $\pi(a|s)$ represents the conditional probability of taking an action a in a state s under policy π. Equation 2.3 can be shown [114] to converge to the *Bellman equation* for $k \to \infty$. Next, we are concerned with the improvement (2) of a policy π into a better policy π' by *greedily* evaluating [114] the approximated *optimal substructure* V^* :

$$\pi'(s) = \arg\max_{a \in \mathbb{A}} \sum_{s'} p_{\pi(s)}(s,s')(R(s,\pi(s),s') + \gamma V^*(s')). \tag{2.4}$$

The agent action a can be found with a successor state prediction [114, p. 96], by merely looking ahead on all successor states and choosing the one which leads to the best combination of state and value as described in Equation 2.4. As a result, the classical policy iteration does require a model and, consequently, expects available state transition knowledge of the environment. The *policy improvement theorem* [114] proves that each policy improvement is guaranteed to increase the policy return, given that each state of the environment is visited infinitely often. The proof behind the theorem expands the Bellman optimality equation and reapplies the policy improvement for $k \to \infty$ until $V_{\pi'}(s) \geq V_\pi(s)$. A structured proof can be found in [114]. On every policy evaluation, we obtain a new, improved V', followed by a policy improvement yielding a new, improved policy π'. By repeating this process iteratively, we obtain a monotonically improving set of subsequent policies and value functions:

$$\pi_0 \to V_{\pi_0} \to \pi_1 \to V_{\pi_1} \to \cdots \to \pi^* \to V_{\pi^*} \tag{2.5}$$

until the optimization converges to the *Bellman equation* V^* with the optimal policy V^*, as visualized in Figure 2.3. Even though the policy improves with every step, the number of repetitions k necessary until the optimal policy is reached can be $k \to \infty$ [114]. The learning time of V^{π_k} is, therefore, to be weighed up against the necessary policy performance.

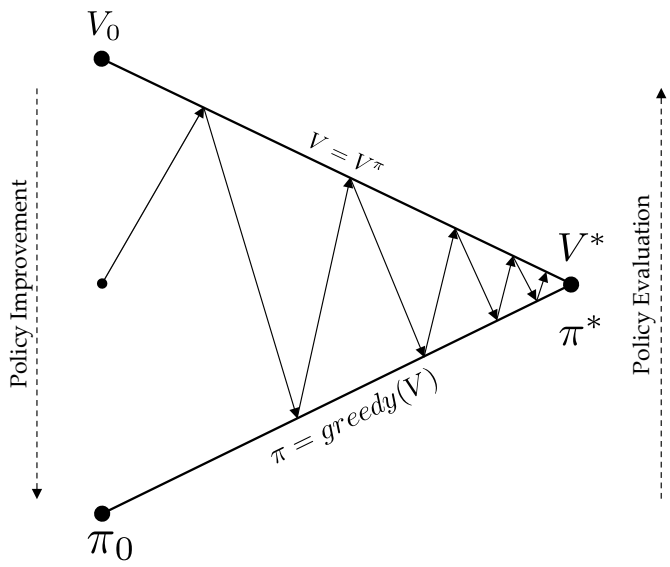


FIGURE 2.3: A graphical visualization of the continuous improvement process of a given policy π starting on the left side and converging to the right into an optimal policy π^*. The arrows pointing upwards indicate the policy evaluation, while the arrows pointing downwards show the greedy policy improvement.

2.1.8 Monte Carlo Methods

Monte Carlo (MC) methods [96] sample to estimate the state value function for a given policy. While the term *Monte Carlo* often specifies an estimation with a random component, in this context, it refers to a method that averages over the complete trajectory return [114]. The approximation of V with MC methods is model-free, as they do not need to know the probabilities and state transition rewards in advance. MC methods can replace the already discussed *policy evaluation* of the DP methods [56, cf.]. When walking through a trajectory, Monte Carlo methods collect experiences first and update the value function then afterward in accordance with [114, p. 119]:

$$V(s_k) \leftarrow V(s_k) + \alpha\Big[g(s_k) - V(s_k)\Big], \tag{2.6}$$

where α is a the step size. MC methods use the rewards of the whole trajectory from the current state s_k until the last state s_{k+n} modeled here as $g(s_k)$, in the actual return following time k. This explains why the value update can only be performed after the trajectory has been completed: $g(s_k)$ is only known after a terminal state has been reached, and all the rewards have been summed up for the particular trajectory [94].

2.1.9 Temporal-Difference Learning

Temporal-Difference (TD) Learning is another popular member of the reinforcement learning family. It is a combination of Monte Carlo approaches and uses ideas from Dynamic Programming. DP, TD, and MC methods all solve a prediction problem [114, cf.]. Their essential difference is how and when they calculate the value function. TD and

MC based value estimations use experience to solve their prediction problem. One of the drawbacks of MC based approaches for the calculation of the value function is that the actual learning happens *offline* after the trajectory has been experienced. Especially for problems with very long trajectories, this deduces the performance of the learning process. In comparison, TD methods must not wait until the end of the episode. Instead, they learn *online* by using the reward information along the trajectory to update V after each timestep.

$$V(s) \leftarrow V(s) + \alpha\Big[R(s,a,s') + \gamma V(s') - V(s)\Big] \tag{2.7}$$

The update presented in Equation 2.7 [94] is the simplest form of a TD update, called $TD(0)$, where $s_k = s$ and $s_{k+1} = s'$. It is a special case of the $TD(\lambda)$ family, where $TD(\lambda)_{|\lambda=0}$. Since the update of the current $V(s)$ uses the already existing estimate of the successor state $V(s')$, $TD(0)$ is also called *bootstrapping*. The part of the equation in the brackets is called *TD error* δ, which is used in various forms of reinforcement learning. δ is the error or difference between the old value estimate for s and the better-improved estimate calculated by [114, cf.]:

$$\delta = R(s,a,s') + \gamma V(s') - V(s). \tag{2.8}$$

The TD value function update is done iteratively and uses the sampled successor states along the rollout rather than a complete distribution of successor states. [56, cf.]

MC approaches, which are equivalent to $TD(\lambda)_{|\lambda=1}$, calculate V with the experienced rewards from the whole trajectory [94]. The $TD(\lambda)$ approach with $\lambda = 0$ calculates V only by taking into account a single step TD error [94]. In order to smoothly interpolate between both extrema, the basic RL theory suggests different methods. *Eligibility traces* interpolate the length of the backward view by truncating the total experienced reward backward [114] in λ-steps as:

$$g_k^\lambda \doteq (1-\lambda)\sum_{n=1}^{K-k-1} \lambda^{n-1} g_{k:k+n} + \lambda^{K-k-1} g_t \tag{2.9}$$

in a sequence of multiple n-step returns with length K, where $\lambda \in [0,1]$. For $\lambda = 0$ the return reduces to $g_{k:k+1}$, the one-step return $TD(0)$, while for $\lambda = 1$ we obtain the complete return g_k, which is equivalent to Monte Carlo methods.

> Note that the subscripts $_{k:k+n}$ are meant to indicate that the return is truncated for the timestep k using the rewards up until $(k+n)$-steps backward into the trajectory.

An alternative called *n-step TD Prediction* [114] truncates the complete return in n-steps. Where n specifies over how many reward experiences the agent is bootstrapping into g_k for a timestep k, as visible in Figure 2.4. For the single-step $TD(0)$, the reward function g is then called "the target of the update" [114, p. 143] and calculated as follows:

$$g_{k:k+1} = r_{k+1} + \gamma V_k(s_{k+1}) \tag{2.10}$$

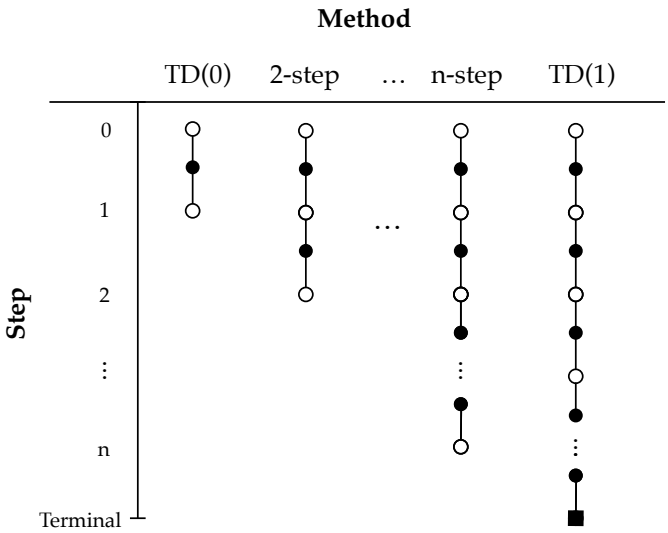


FIGURE 2.4: We can see how the n-step TD approach backups experiences. On the very left is the 1-step TD approach ranging up until the very right to the n-step TD approach, which backups the whole trajectory.

which is essentially the first reward value plus the discounted estimated value of the next state. For the $TD(1)$ (MC) approach, the complete return is calculated as:

$$g_{k:k+1} = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + ... + \gamma^{K-k-1} r_K. \tag{2.11}$$

With the arbitrary n-step update, this results for the calculation of the n-step return value into the following:

$$g_{k:k+n} = r_{k+1} + \gamma r_{k+2} + ... + \gamma^{n-1} r_{k+n} + \gamma^n V_{k+n-1}(s_{k+n}) \tag{2.12}$$

where $n \geq 1$ and $0 \leq k \leq K - n$. The n-step return is thus an approximation of the full return value, but truncated after n-steps and then corrected [114] over the missing terms by $\gamma^n V_{k+n-1}(s_{k+n})$.

2.1.10 Model-Free Learning

DP-based learning is model-based. In its pure form, a model of the environment is required. When it comes to the agent in a state s to greedily find the action which leads to best-valued successor state $s' \in \mathbb{S}$, model-based approaches must predict all possible next states with a model before the action is executed, as illustrated in Figure 2.5. This prediction is impracticable without knowing all possible next states and their probabilities. Q-Learning [125], an early success in reinforcement learning, has been developed as an off-policy TD control algorithm to solve this problem in model-free environments. Instead of learning a state value function $V(s)$, *Christopher Watkins* proposed [125] in his dissertation thesis to use an action-value function:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \Big[R(s,a,s') + \gamma \max_{a \in \mathbb{A}} Q(s',a) - Q(s,a) \Big], \tag{2.13}$$

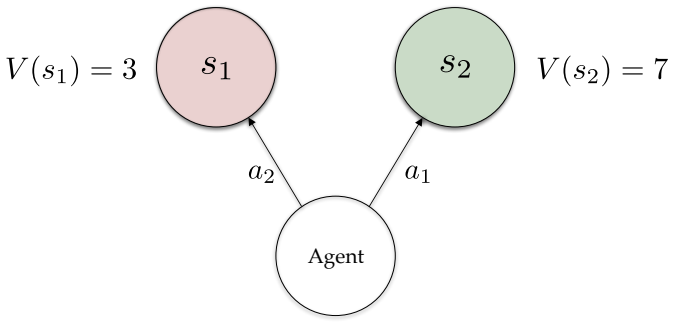


FIGURE 2.5: An agent in a state s_0 in a formal deterministic decision of selecting an action. In order to decide for the state with the best value, the agent must know all possible predecessor states when learning with a DP-based learning method.

which directly approximates the optimal Q-value by a given state-action tuple. The optimal action-value function represents the maximum expected possible return when following a greedy policy π. This statement complies with the Bellman equation under the assumption that the optimal value function Q^* in a trajectory of successor states is known for all possible actions A. Therefore, the optimal policy becomes the selection of an action $a' \in \mathbb{A}$, by maximizing the expected value of the target $R(s, a, s') + \gamma \max_{a \in \mathbb{A}} Q(s', a)$. Given that all state-action pairs are visited infinitely often, the learning rate α is reduced appropriately, and the state and action space are finite, the learning rate α is reduced appropriately, and the state and action space are finite it has been shown that Q-Learning converges to the optimal Q^* [94, 125, cf.]. Similar to other value-based approaches, the agent is responsible for making an experience and storing that reward experience successively according to Algorithm 1 [125, 114]. The inner loop from line 6 to line 12 represents the preceding of the agent through a trajectory until a terminal state has been reached. In each state, an action is selected and executed. After that, the occurring state transition rewards, as well as the successor state, are observed from the environment. The Q-function is updated online according to Equation 2.13 while the algorithm repeats this process for a defined number of episodes.

$$\pi^*(s) = \arg\max_{a \in \mathbb{A}} Q^*(s, a). \tag{2.14}$$

The agent will then find the optimal policy by greedily exploiting [125] the Q-table with Equation 2.14.

2.1.11 Off-Policy Learning

The terminology *off-* and *on-policy* categorizes the different RL methods into two sections [114]. The two approaches primarily differ in the way they generate data. All algorithms discussed up to this point are on-policy, which optimize and evaluate a *target policy* directly. Off-policy methods, in contrast, optimize a *behavior policy* that deviates from the current optimal choice of decisions to explore a more extensive state space "off" the current target policy. This behavior policy can be implemented by randomly behaving suboptimal or by making decisions based on "old" policies or data.

Algorithm 1 Q-Learning

1: **Parameters:** step size $\alpha \in (0, 1]$, small $\epsilon > 0$
2: $Q_0 \leftarrow -1$ ▷ initialize Q for all $s \in \mathbb{S}$ and $a \in \mathbb{A}$ arbitrary
3: $Q(terminal, \cdot) \leftarrow 0$ ▷ set terminal state values to 0
4: **for all** $e \in episodes$ **do**
5: initialize s
6: **repeat**
7: derive a from Q ▷ e.g. $\epsilon - greedy$
8: execute action a
9: observe r, s′
10:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \Big[r + \gamma \max_{a \in \mathbb{A}} Q(s',a) - Q(s,a) \Big]$$

11: $s \leftarrow s'$
12: **until** s is terminal state
13: **end for**

Q-learning is an off-policy algorithm. The reason behind this categorization comes from line 7 in Algorithm 1. We can see that an action is derived from the Q-function. We can derive an action from the Q-function by selecting an action under a *behavior policy*. The ϵ-greedy [114] policy is a common approach to describe such a behavior policy. It selects an action with a certain probability $p_{random} = \epsilon$ randomly and an action from the Q-function greedily with a probability of $p_Q = 1 - \epsilon$. This form of selection turns ϵ into a hyper-parameter, which balances policy exploitation and exploration accordingly. There are different strategies to optimize ϵ, which are beyond the scope of this work; however, the most common approaches can be found in [114].

2.2 Approximating in Value Space

In Section 2.1, we introduced basic traditional RL approaches. These methods use a tabular-based value function, which is a memory-structure contains a corresponding entry for each state or action-state value. While the usage of tabular value functions can work well on simple low-dimensional control problems, it is not feasible for many practical domains. In this section, we discuss *non-linear function approximation* as an alternative to tabular-based value functions. There are many types of non-linear approximations, such as e.g., the *Hilbert Space Approximation* [18] or the *Piecewise Polynomial Approximation* [18]. In this thesis, the term *non-linear approximation* refers to an approximation using a *neural network*. Hence we first introduce the artificial neural network and then infer a non-linear approximation of the value function. On top of that, we discuss the challenges and solutions of value-based deep reinforcement learning.

2.2.1 Artificial Neural Network

Inspired by the properties of the *biological neural network*, the artificial neural network (ANN) has become a widely used choice in modern machine learning. ANNs solve

a wide variety of problems in *supervised* and *unsupervised* machine learning such as *image recognition* [43] and *speech recognition* [34]. In the context of reinforcement learning, ANNs play a more predominant role [114] in the use of *non-linear function approximation*. Figure 2.6 shows the generic form of an ANN, consisting of multiple layers with multiple units, where each unit is connected strictly with each unit of its subsequent layer. We speak of a *feedforward* network with no communication loops. Each

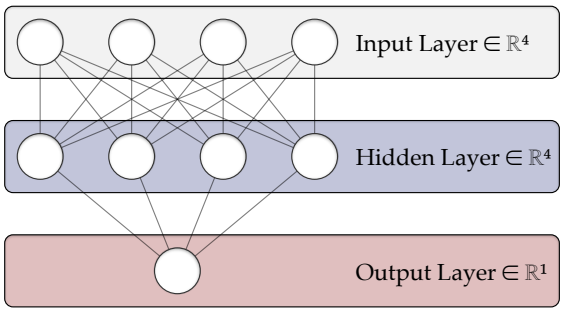


FIGURE 2.6: A generic feedforward neural network with four input units, one output unit, and a single hidden layer.

connection is associated with a real-valued *weight*, which is iteratively adjusted to optimize the output of the neural network. The *semi-linear* units calculate the weighted sum of their input signals and output the result using a non-linear function transformation, typically called the *activation function*. There are different activation functions, whereas in this thesis we consider two types of activation functions: *Tangens hyperbolicus* (tanh) $tanh(x) = 1 - (2/(e^{2x} + 1))$ and the *rectifier linear unit* (ReLU) [81] $relu(x) = x^+ = max(0, x)$. In order to train a neural network, we first *feedforward* the non-linear activation pattern over the input units and then *backpropagate* the influence of each weight w.r.t. an arbitrary defined objective function E [93]. This calculation, commonly known as *backpropagation*, is performed by repeated utilization [93] of the *chain rule* for each neuron of the network:

$$\frac{\partial E}{\partial \theta_{ij}} = \frac{\partial E}{\partial out_i} \frac{\partial out_i}{\partial net_i} \frac{\partial net_i}{\partial \theta_{ij}}, \tag{2.15}$$

where out_i is the output, net_i is the weighted sum of the inputs from unit i, and θ_{ij} is the weight between unit i and unit j. By concatenating all partial derivatives into a vector, we obtain an estimate of the true *gradient*: $\nabla_\theta E(\theta)$.

> Note that the gradient of a scalar vectored *mulivariate function* $f : \mathbb{R}^n \to \mathbb{R}$ of $x \in \mathbb{R}^n$ is defined as $\nabla_x f(x) := (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n})|_x \in \mathbb{R}^n$, where each element is proportional to the derivative of the function w.r.t. that element.

One may now minimize the objective function E by iteratively applying *batch gradient descent* [97]:

$$\theta = \theta - \alpha \nabla_\theta E(\theta), \tag{2.16}$$

where α is the step size or learning rate. Thus we optimize E to *first order*, following its negative gradient on curvature space w.r.t. the parameter vector θ. *Vanilla gradient descent* has been shown to be slow for a variety of domains and requires access to the

full data-set for a single update [97]. Most modern deep RL approaches [75, 105, 74, 68], therefore, leverage *stochastic mini-batch gradient descent* (SGD) [11, 97] instead, which performs a gradient update *online* for a set of mini-batches of n training samples:

$$\theta = \theta - \alpha \nabla_\theta E(\theta; x^{(i:i+n)}; y^{(i:i+n)}), \tag{2.17}$$

where $x^{(i:i+n)}$ corresponds to a mini-batch of *samples* tagged by an equal-sized mini-batch of *labels* $y^{(i:i+n)}$.

Generalization in the context of a neural network refers to the capability of handling *previously unseen* patterns [129]. Say, a neural network is fitted with a *training-set* $\mathcal{X} = \{x_1, x_2, ..., x_n\}$, then the learned non-linear relationship needs to be able to predict a suitable output y for any $\widetilde{x} \notin \mathcal{X}$. In supervised learning, the ultimate goal of the training of a neural network is to achieve a good *generalization capability*. Generalization is typically measured with a separate *test-set* $\mathcal{Z} = \{z_1, z_2, ..., z_n\}$, which is drawn independently from the same underlying *data generating distribution* [129] as $\mathcal{X}$. We then optimize the neural network with an arbitrarily defined error function E on the training-set $\mathcal{X}$, however, measure its *generalization error* [129] on the test-set $\mathcal{Z}$. The difference between the training error and the generalization error is known as *generalization gap* [129]. In order to achieve a good fit, optimal optimization must avoid two types of behaviors: *Overfitting* and *underfitting*. Overfitting is referred to as optimization behavior that is not able to reduce the generalization gap reasonably small [129], whereas underfitting is referred to as a fitting behavior that is not able to make the training-error adequately small [129].

Convolutional neural networks (CNN) [62] are a class of deep neural networks mainly applied to image analyzation. They imitate the effects of receptive fields of the early visual human cortex by using multiple hierarchical layers of tiled convolutional filters [75]. The hierarchical structure enables the network to recognize correlations and object categories from raw pixel data by successively producing more abstract representations from the data over multiple hidden layers. In previous RL approaches, the state of the environment was usually individually handcrafted on a limited amount of low-dimensional input features. CNNs allow the end-to-end representation of state with raw sensor data, such as pixels captured from a video or image. This ability contributes to the cross-domain usage of a single algorithm and allows a wider variety of problem domains.

2.2.2 What to Approximate?

In value-based reinforcement learning algorithms, the representation of a value function as a hash-table becomes very difficult when the amount of states is either very high or continuous. If the amount of states raises, the probability for an agent of revisiting an experienced state decreases [114], independent of the memory limitations. In other words: The agent cannot benefit from experience collected from similar states. A solution to this problem may be the discretization of the state-space. However, this can greatly limit the accuracy and performance of the learned policy. *Function approximation* [9] is a popular choice to overcome this issue. When using a non-linear function

approximation, such as a neural network, we can estimate the action-value function $Q(s,a)$ with a set of free parameters θ, which correspond to the weights and biases of the neural network, so that $Q(s,a) \approx Q(s,a;\theta)$ [75]. We can train the neural network by optimizing a changing sequence of loss functions $L_k(\theta_k)$ at every k iteration [76]:

$$L_k(\theta_k) = \mathbb{E}_{s,a\sim p(s,a)}\left[\left(y_k - Q(s,a;\theta_k)\right)^2\right]. \tag{2.18}$$

Here, $p(s,a)$ corresponds to a *behavior distribution*, from which states s and actions a are sampled from, while the target y_k is defined [76] as:

$$y_k = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'\in\mathbb{A}} Q(s',a';\theta_{k-1})|s,a\right], \tag{2.19}$$

in which we obtain the states we are averaging over, from the environment $s' \sim \mathcal{E}$.

> Note that the notation $\mathbb{E}_{s,a\sim p(s,a)}[\cdot]$ is meant to denote that we sample states s and actions a from the distribution $p(s,a)$ when calculating the expectation.

By differentiating the loss function w.r.t. θ, we attain the following gradient [76]:

$$\nabla_{\theta_k} L_k(\theta_k) = \mathbb{E}_{s,a\sim p(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'\in\mathbb{A}} Q(s',a';\theta_{k-1}) - Q(s,a;\theta_k)\right)\nabla_{\theta_k} Q(s,a;\theta_k)\right]. \tag{2.20}$$

Rather than calculating the full expectation, practitioners customarily optimize this loss function by stochastic gradient descent. Equally to the vanilla Q-learning algorithm, we speak of a model-free algorithm, as a model is not required when exploiting the approximated Q-function greedily in accordance with $a = \max_{a\in\mathbb{A}} Q(s,a;\theta)$.

2.2.3 Challenges

The combination of deep neural networks and reinforcement learning was previously regarded [33, 119] to be unstable; however, at the same time inevitable for many practical problems. Sutton et al. summarized this danger of instability as *the deadly triad* [115] of *function approximation*, *bootstrapping*, and *off-policy training*. If any two components of the deadly triad are present in the algorithm, but not all three, then the instability issues can be averted. If all three elements of the deadly triad are combined, learning can *diverge* [41] as estimates of the approximated value function become unbounded. However, all three elements are indispensable for scalability, generalization, data efficiency, and off-policy learning to achieve a well-performing policy [67, 114, cf.]. Interestingly, several recently introduced value-based deep RL algorithms [76, 40, 100] successfully combine all of those three properties, which shows that there is at least a partial gap in our understanding of the emergence of the deadly triad [41].

When looking into publications that successfully overcome the deadly triad issue, it emerges that the authors tackle the integration of sizeable non-linear function approximations instead from a different perspective, where they focus on differences to classical supervised learning approaches.

Generalization

A dominant issue appears to be the global representation mechanism of a neural network, which causes updates in a particular part of the state space to affect arbitrary other regions as well [92]. This effect, commonly known as *generalization*, also accelerates learning by allowing the exploitation of experience in similar state spaces. However, since even small updates in the value function can considerably change the policy [75], this may also work against the desired effect of learning.

Correlated Target Values

Further instabilities are caused by the strong correlation of the state values with their target values [75]. Consider, for example, Equation 2.18, which denotes the minimization of a *sequence* of loss functions, in which the target values y_k change in every optimization epoch in dependence of the action-value function $Q(s, a; \theta_k)$. Supervised learning methods, however, presume a fixed target distribution, which is independent of their fitted values.

Dynamic Underlying Distribution

In contrast to classical deep learning approaches, which usually learn from a large amount of hand-labeled training data, deep RL approaches learn from a *noisy*, *delayed*, or even *sparse* reward signal [76]. Recent research [95, 120] has indicated that deep learning can be incredibly robust against noisy labels. However, the underlying distribution of labels is highly correlated with the behavior of the policy, and data samples are, consequently, not independent.

2.2.4 Target Network and Experience Replay

Two predominantly successful approaches to overcome the challenges of value-based deep RL became popular with the publication of *Deep Q Networks* (DQN) [76, 75].

Experience Replay [70], which has been known in the RL community for quite a while, works by storing experience-tuples $e_t = (s_t, a_t, r_t, s_{t+1})$ at each timestep t in a queue $\mathcal{D}_t = \{e_1, e_2, ..., e_t\}$. The update of the neural network is then computed with samples drawn from this memory structure $(s_t, a_t, r_t, s_{t+1}) \sim U(\mathcal{D})$ with a uniform distribution U. This offers previous knowledge *explicitly* [92] on each update of the gradient and works against the undesired effects of generalization. The size of the replay memory turns out to be a crucial hyper-parameter, as it may delay critical agent transitions, which can considerably impact the final policy's performance [132].

The concept of a separate *Target Network* [75] was first introduced with DQN. The idea is to use an additional target network with a fixed set of weights θ^-. The *target network* is then used for calculating the target values $y_k = r + \gamma \max_{a' \in \mathbb{A}} Q(s', a'; \theta_k^-)$, while the *online network* $Q(s, a; \theta_k)$ is applied for the calculation of the Q-values, resulting into the following loss-function:

$$L_k(\theta_k) = \mathbb{E}_{s,a \sim p(\cdot); s' \sim \mathcal{E}} \left[\left(r + \gamma \max_{a' \in \mathbb{A}} Q(s', a'; \theta_k^-) - Q(s, a; \theta_k) \right)^2 \right]. \tag{2.21}$$

Every C-steps, one may now periodically update the frozen weights of the target network with the weights of the online network. This technique effectively reduces the correlation between the target values and the action-value function, by allowing the neural network optimizer to reduce the loss onto a temporarily fixed target.

Experience replay has been applied previously [92, 94, e.g.] to stabilize the training of neural networks in the reinforcement learning setting. However, the results of Mnih et al. show that the combination of both methods, experience replay with a target network, synergize exceptionally well with the *online*-setting of DQN. Previous methods have utilized relatively small multi-layer perceptrons, which were updated *offline* by using *batch gradient descent* after hundreds of iterations [92]. Deep neural networks are known to require a more substantial amount of training samples [63] and, therefore, raise the concern of sample efficiency. Hence, the combination of *stochastic mini-batch gradient descent* with experience replay and target networks appears to be the correct balancing act between *sample efficiency* and *stability* when using a deep neural network.

2.3 Literature Review

The timeline in Figure 2.7 shows the dependence of the different value-based deep RL proposals, which are compared regularly against novel methods. After the proposal of Deep Q Networks in 2013, a vast amount of different improvements and proposals arose from the RL research community. DQN leverages CNNs in order to approximate the Q-value space directly from raw pixel data; outperforming [75] *human-level* and prior *linear function approximation* on the *Atari 2600* games.

The idea of non-linear value-space approximation in reinforcement learning was studied [69, 33, 24, 92, 94] prior to the work of Mnih et al. Subsequent to a few proposals [69, 33], which contributed to a stable non-linear function approximation in the field of DP, the success of Tesauro's TD-Gammon [117] portrays one of the first successful outliers in the field of deep RL. However, in the following years, Tesauro's results remained fairly obscure and were hard to reproduce [89]. A first alternative framework was proposed as *Fitted Q Iteration* (FQI) [24], which was based on classical *tree-based* regression [5] instead. The FQI framework was later picked up by Riedmiller's *Neural Fitted Q Iteration* (NFQ) [92], where the type of regression was replaced with a multi-layer perceptron to learn agent policies from a selected low-dimensional feature space. Also, methods that learn agent policies directly from high-dimensional sensor space were known [60, 99] before the proposal of DQN, using an *Autoencoder* [63] bottleneck in order to feed a low-dimensional non-linear function approximation. Such type of *Batch Reinforcement Learning* [99] approaches work iteratively by (1) collecting a batch of multiple agent episodes under the current policy, and then by fitting (2) a small multi-layer perceptron with *batch gradient descent* on the collected experiences to improve the policy. The old experience batch is then either discarded or appended onto a *growing batch* [99].

Between the years 2015 and 2017, increasingly active research [40, 100, 6, 123, 26, 48] has been conducted in order to detect and prevent different bottlenecks of the DQN algorithm. While Figure 2.7 does not constitute the complete collection of proposed

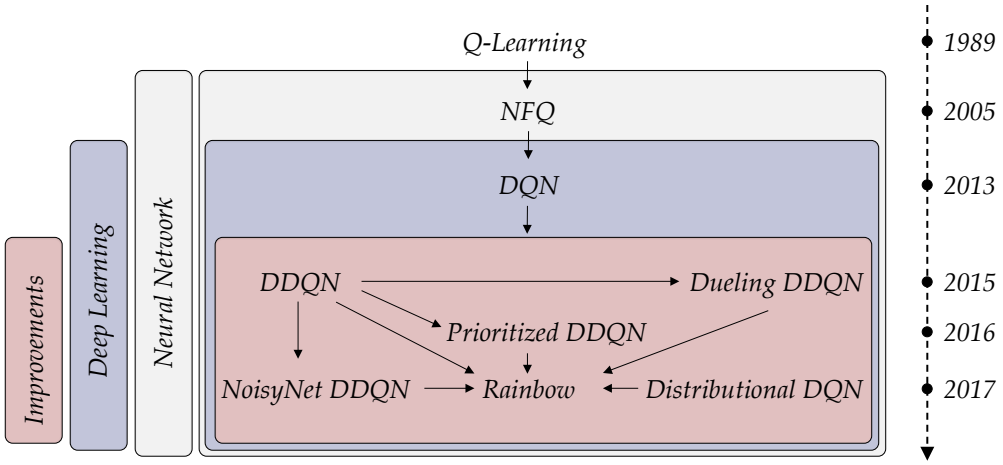


FIGURE 2.7: A timeline of the different value-based deep RL methods and their dependence. Notice that since 2013, the methods apply deep learning. From 2015 until 2017, different improvements of DQN were proposed, whereas *Rainbow* represents a combination of all.

enhancements, it does, however, summarize the most significant of its kind.

Hasselt, Guez, and Silver found that DQN suffers from consistent overestimation. *Double DQN* (DDQN)[40] addresses this issue by decomposing the maximization operation of the traditional DQN target: $y^{DQN} = r + \gamma \max_{a' \in \mathbb{A}} Q(s', a'; \theta_k^-)$ into action evaluation and action selection such that: $y^{DDQN} = r + \gamma Q(s', argmax_a Q(s', a; \theta_k); \theta_k^-)$. This simple modification uses the target network to estimate the greedy policy value, but evaluates it with the online network.

Dueling DDQN (DDDQN) [123] introduces a dueling architecture with two separate estimators; one for the state-dependent *advantage* function $A(s, a) = Q(s, a) - V(s)$ and one for the state value function $V(s)$, leading to further empirical improvements.

> Notice that we denote the advantage function with A, while the action space has been defined as $\mathbb{A}$. The advantage function will be discussed in more detail later in Section 3.1.5.

Both value functions are then merged by a special aggregator in accordance with:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + [A(s, a; \theta, \alpha) - \frac{1}{N} \sum A(s, a'; \theta, \alpha)], \tag{2.22}$$

where N stands for the action dimension and α and β are free parameters, prioritizing each value function accordingly.

Prioritized experience replay [100] improves the experience replay memory of DQN by threading collected experiences as not equally relevant. Instead of drawing collected experiences with a uniform distribution from the replay memory, Schaul et al. suggested using a stochastic sampling method that interpolates smoothly between uniform random sampling and a *greedy* prioritization based on the temporal difference error.

The probability of sampling the i-th experience of a replay memory is then defined by $P(i) = p_i^\alpha / (\sum_k p_k^\alpha)$, where $p_i > 0$ represents the priority of the i-th experience and α is the interpolation parameter between uniform sampling and priority sampling. New additions to the replay memory are added with maximum priority in order to ensure that all agent transitions are included in the value function estimation.

Noisy Networks [26] address the exploration issues of DQN, especially in regards to the difficulty of adjusting the exploration hyper-parameter ϵ, by using an additional noisy linear layer. Consider a linear neural network layer $y = wx + b$. We can then replace the slope and the intercept with two stochastically approximated identities $w = \mu^w + \sigma^w \odot \varepsilon^w$ and $b = \mu^b + \sigma^b \odot \varepsilon^b$, where for each term $\sigma^{(\cdot)}$ and $\mu^{(\cdot)}$ are approximated by the neural network and $\varepsilon^{(\cdot)}$ corresponds to a separately chosen noise variable. During the optimization, the network can learn to ignore the noise in different parts of the state space, allowing a *self-annealing* exploration.

> Note that $\odot$ denotes the entry-wise Hadamard product which we discuss further in Appendix A.

Distributional RL [6] reformulates the Bellman equation with an approximate *value distribution* Z^π, which maps state-action pairs to distributions over returns, rather than taking an approximate expectation. Such *distributional Bellman equation* is then defined by $Z(s,a) = R(s,a,s') + \gamma Z(S', A')$, where the expected scalar quantity Q of the regular state-action Bellman equation [114]

$$Q^*(s,a) = \sum_{s',r} p_a(s,s')(r + \gamma max_{a' \in \mathbb{A}} Q^*(s',a')) \tag{2.23}$$

is replaced with the random *distribution of future rewards* $Z(S', A')$ [6]. Note that S' and A' are capitalized to emphasize the random nature of the next state-action pair since we consider here the expected outcome of the random transition $(s,a) \to (S', A')$ [6]. The value distribution is then modelled by a set of *atoms* [6], which represent the returns of the distribution

$$\left\{ z_i = V_{min} + (i-1)\frac{V_{max} - V_{min}}{N-1} : 0 \leq i < N \right\}, \tag{2.24}$$

where $N \in \mathbb{N}^+$ is the specified amount of atoms. At a timestep t, we can then define [48] the probability mass function $p_\theta^i(s_t, a_t)$ for an i-th atom in a distribution $d_t = (z, p_\theta(s_t, a_t))$. Our optimization problem becomes to adapt θ in such a way, that we match a *target distribution*

$$d_t^{target} \equiv R_{t+1} + \gamma_{t+1} z, p_{\theta'}(S_{t+1}, \arg\max_{a \in \mathbb{A}} z^T p_\theta(S_{t+1}, a)), \tag{2.25}$$

which is the optimal distribution under the optimal policy π^*, satisfying the *distributional Bellman equation* [48]. A neural network then represents the parameterized distribution, such that we can write down the loss function as the *cross-entropy* term of the Kullback-Leibler (KL) divergence[1] [59], between the parameterized policy and the target policy as $L_{s,a}(\theta) = D_{KL}(\Phi_z d_t^{target} || d_t)$.

[1] The KL divergence and the cross-entropy are covered in Appendix A.

Note that $\Phi_z d_t^{target}$ denotes an L^2-*projection* of the target distribution d_t^{target} onto a fixed discrete support z. We indicate how to solve such a projection in Appendix A but refer the reader to [6] or [12] for a more in-depth explanation.

Rainbow [48] implements a combination of the above mentioned improvements and additionally implements a form of n-step learning from Section 2.1.9, in which the truncated return forms an alternative [48] DQN objective:

$$\underset{\theta}{minimize} \sum_{t=1}^{N} || \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1} + \gamma_t^{(n)} \underset{a'}{max} Q(s_{t+n}, a'; \theta^-) - Q(s_t, a_t; \theta)||^2, \tag{2.26}$$

where the multi-step target is tuned n times throughout N total timesteps. Hessel et al. showed that the integration of different improvements could be effectively combined and that it improves the sample efficiency of value-based RL significantly, as to be seen in Figure 2.8. Rainbow has shown competitive performance in different publicly available RL challenges with discrete action space dimensions such as the OpenAi *Retro Contest* [82].

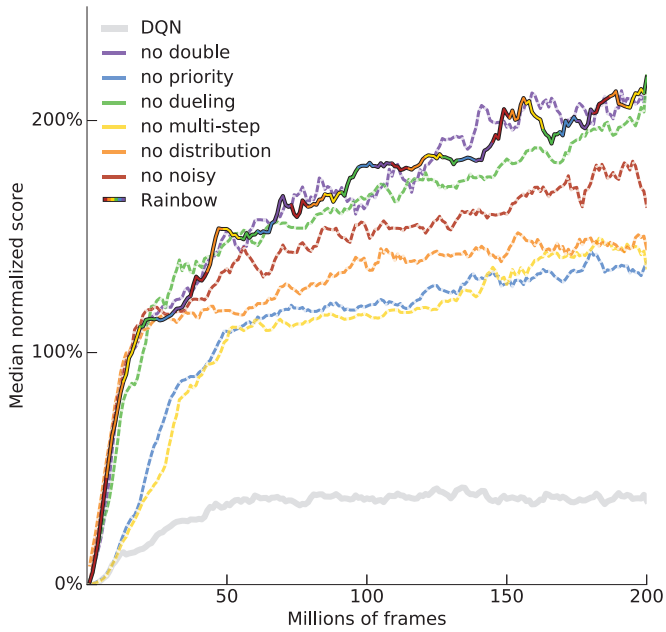


FIGURE 2.8: The average empirical results of Rainbow evaluated on 57 Atari games with the influence of different improvements, which we directly took from [48]. Hessel et al. normalized the score based on human performance. We can see a significant difference in sample-efficiency between the original DQN (bottom grey line) and combined improvements (rainbow-colored top line). The different branches in between results show the performance decrease of each DQN improvement being disabled.

2.4 Summary

In this chapter, we looked at value-based deep reinforcement learning algorithms. We first discussed classical tabular-based algorithms and then derived non-linear function approximation as a possibility to applicate value-based RL in a much more high-dimensional state dimension. Furthermore, we explored challenges that occur when using deep learning to approximate value space and considered the core contributions of DQN as possible salvations. Finally, we looked at a broader spectrum of DQN improvements that evolved in the last couple of years.

Compared to the following Chapter 3, which covers policy gradient-based deep RL approaches, the reader will notice a considerable difference in regards to how far the chapters dive into the theoretical properties. Despite the remarkable empirical success of DQN, its basic theoretical principles are less well-understood [128, 41, cf.]. Even though, parallel to this thesis, there have been attempts to understand the success of value-based DQN statistically [128] and visually [122], the majority [75, 74, 40, 100, 6, 123, 26, 48] of the value-based DQN proposals are largely based on empirical findings.

Furthermore, the answer to the question: "How and why does DQN overcome the *deadly triad issue*[2] so successfully?", is still under active research [41] and has not been fully processed up to this point in time. A first empirical study [41] indicates that the interaction of large neural networks with different improvements, such as the *prioritization of the replay memory* or the *type of bootstrapping method used*, may be more nuanced than initially hypothesized by the research community. Hence, further research may be required to comprehend the theoretical background and the findings of value-based deep reinforcement learning in its entirety.

[2] see Section 2.2.3

Chapter 3

Policy Gradient-Based Approaches

Deep reinforcement learning relies on function approximation. In Chapter 2, we discussed a standard approach, which first approximates a value function and then derives the optimal policy from it. In Chapter 3, we take a closer look at an increasingly popular alternative in deep RL that represents a *policy* directly by a function approximation and then optimizes it following the *gradient of the expected rewards*, w.r.t. the policy parameters. Policy gradient-based methods are less sample efficient compared to value-based approaches; however, they do not suffer from as many problems when marrying them with non-linear function approximations. Contrarily, policy gradient algorithms have their own challenges and are not to be taken as the salvation to all problems. In the last couple of years, the research community has leveraged novel findings of *optimization theory*, which made the use of deep neural networks with policy gradient algorithms much more attractive.

In this chapter, we identify *Policy Gradient* (PG) [126, 115] methods as a subcategory of policy search and understand their role in general reinforcement learning. PG-based algorithms require a significant amount of math to understand their background thoroughly. We, therefore, introduce definitions in a reasonable quantity; however, refer the reader to the mathematical encyclopedia in Appendix A. After introducing necessary foundations, we identify the theory of *trust region methods* [16] as an essential underlying motivation of many recent deep RL publications; which we understand by first recognizing the critical problems of previous policy gradient methods and then by extending expedient solutions step by step into a final practical algorithm of the *Proximal Policy Optimization* (PPO) [105] family. Lastly, we close this chapter with a literature review.

3.1 Foundations

In this section, we cover the foundations of gradient-based RL. We start with the basic terminology and categorization to continue with the optimization of the policy's gradient. Finally, we introduce different well-known improvements, which are essential to understand before continuing with more advanced gradient-based optimization theory.

3.1.1 Policy Search

Policy search-based reinforcement learning aims to improve a given policy directly by *searching* [29] in the space of possible policies $\mathbb{P}$. Typically, a local policy optimum π^* is found by choosing a parameterized policy π_θ that maximizes [4] the expected policy return

$$\underset{\theta}{maximize}\ \mathbb{E}[R|\pi_\theta] \tag{3.1}$$

in a subset of representable policies $\mathbb{P}_\theta \subset \mathbb{P}$, where θ is a vector of real-valued policy parameters and R refers to the total reward of the episode.

In contrast to the classical policy iteration scheme of dynamic programming, policy search solves the policy optimization problem without a value function by either using gradient-based [115, 53, 65] or gradient-free [116, 114, 79] methods. Also, there are existing combinations [42] of both worlds, which have shown remarkable performance in the past. While completely gradient-free methods can efficiently optimize policies with a low-dimensional parameter space, they have been significantly outperformed in higher-dimensional environments by gradient-based policy search, using deep neural networks [4]. Figure 3.1 illustrates the learning objectives of policy search (actor) methods compared to value-based (critic) methods. While *actor* methods attempt to learn in policy space $\mathbb{P}$, *critic* methods learn in value space $\mathbb{V}$, from which they derive the optimal policy.

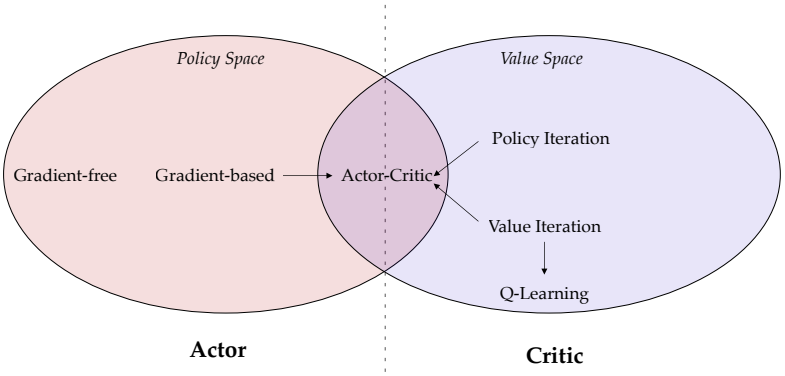


FIGURE 3.1: Different learning spaces of actor and critic methods and how they relate to each other. The left-hand side shows the policy space $\mathbb{P}$, while the right-hand side shows the value space $\mathbb{V}$. At the point where $\mathbb{P} \cap \mathbb{V}$, we speak of actor-critic methods, which are elaborated in Section 3.1.5.

PG methods solve the policy search optimization objective through repetitive estimation of the policy performance gradient. They assume a differentiable parameterized stochastic policy $\pi_\theta(\cdot)$, which they can differentiate w.r.t. the parameter vector θ, to optimize the policy's expected reward [56]. Derivative-free policy search optimizations perturb the policy parameters θ, measure the policy performance, and then move in the direction of good performance. Policy gradient methods do not require to perform any perturbation, as they can *estimate* the policy improvement w.r.t. θ directly. This form of

direct estimation enables them to optimize much larger policies and parameter spaces in comparison to derivative-free optimization [103].

The principal difference to pure value-based RL is the policy itself. In contrast to deterministic policies, stochastic policies diminish the need for exploration [56]. On the other hand, the policy may have very high variance, which can result in extensive, and, thus, destructive policy updates [105]. A critical necessity is, consequently, to collect the trajectories over a sufficiently large batch-size in order to apply the policy update with a reliable estimate of the expected discounted return of multiple trajectories. PG methods are handy when the action space is continuous or stochastic [114]. Critic-only methods have not shown to perform well when applied onto a very large action-space [76, 74, 75].

3.1.2 Preliminaries

Let π denote a stochastic policy $\pi : S \times A \rightarrow [0,1]$ in a 6-tuple MDP $(S, A, P, r, p_0, \gamma)$, in which S and A represent the finite sets of states and actions, and P relates to the transition probability distribution $P : S \times A \times S \rightarrow \mathbb{R}$. Let r be a reward function embedded into the transition probability distribution such that $P(s_{t+1}, r_t | s_t, a_t)$ specifies the probability of a reward r_t and successor state s_{t+1} for each state-action pair (s_t, a_t). Note that, in comparison to a deterministic reward formulation $R(s_t, a_t, s_{t+1})$, the reward can be simulated by embedding it into the transition probability distribution P. We will later see that this reformulation allows the stochastic optimization of the deterministic reward, by optimizing the transition probability distribution—in which the reward is lumped in. The initial distribution of the state s_0 is represented by $p_0 : S \rightarrow \mathbb{R}$ and $\gamma \in (0,1)$ is the discount factor. In each timestep t, the agent chooses an action a_t, which is sampled from a stochastic policy $\pi(a_t|s_t)$. After the agent has chosen the action, the environment generates a successor state and a reward with the distribution $P(s_{t+1}, r_t | s_t, a_t)$. Policy gradient methods usually sample over a batch of trajectories, which refer to as a *segment*. A trajectory is defined as the sequence of states, actions, and rewards the agent observes until a terminal state T is reached: $(s_0, a_0, r_0, s_1, a_1, r_1, ..., s_T)$. Thus, let τ denote the batch of trajectories the agent collects before the policy is updated. Similar to value-based RL, the goal of the agent is to find the policy which optimizes the expected total reward of τ:

$$\begin{aligned} &\underset{\pi}{maximize}\ \mathbb{E}_\tau[R|\pi] \\ &where\ R = r_0 + r_1 + ... + r_{length(\tau)-1}. \end{aligned} \tag{3.2}$$

Note that the expectation $\mathbb{E}_\tau[R|\pi]$ is meant to denote that we are averaging over a set of trajectories τ, while π represents the conditioning expression affecting R over τ.

Further, let $\eta(\pi)$ be the expected *discounted* reward in an infinite-horizon MDP, such that:

$$\begin{aligned} &\eta(\pi) = \mathbb{E}_{s_0,a_0,\ldots}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t)\right] \\ &\textit{where } s_{t+1} \sim P(s_{t+1}, r_t|s_t, a_t), a_t \sim \pi(a_t|s_t), s_0 \sim p_0(s_0). \end{aligned} \tag{3.3}$$

The discount parameter γ is usually introduced in value-based MDP formulations to weight later rewards indifferent from earlier rewards. In this context, γ is introduced as a technique [104] to reduce excessive variance for large horizon MDP formulations. This concept will be elaborated further in Section 3.1.4.

3.1.3 Policy Gradient

Consider a random variable $x \sim p(x|\theta)$, where p is a parametric distribution. Let $f(x)$ be a function, for which we need the gradient of its expected value $\nabla_\theta \mathbb{E}_x[f(x)]$. The direct computation of this gradient is infeasible, since the integral is typically unacquainted, and the derivative is taken w.r.t. θ of p. However, the *score function estimator*[1] allows to calculate an unbiased gradient estimator [115], such that:

$$\begin{aligned} \nabla_\theta \mathbb{E}_x[f(x)] &= \nabla_\theta \int dx\ p(x|\theta) f(x) \\ &= \int dx\ \nabla_\theta p(x|\theta) f(x) \\ &= \int dx\ p(x|\theta) \frac{\nabla_\theta p(x|\theta)}{p(x|\theta)} f(x) \\ &= \int dx\ p(x|\theta) \nabla_\theta \log p(x|\theta) f(x) \\ &= \mathbb{E}_x[f(x) \nabla_\theta \log p(x|\theta)]. \end{aligned} \tag{3.4}$$

Note that in Equation 3.4, we switched the integral and the derivative. It is not trivial to show the validity of this interchange in general. However, sufficient validity conditions are provided by related literature [64].

This equation can be used [115] to approximate the gradient $\hat{g}$, by averaging N sample values $x \sim p(x|\theta)$ in accordance with [103]:

$$\hat{g} = \frac{1}{N}\sum_{n=1}^{N} f(x) \nabla_\theta \log p(x|\theta). \tag{3.5}$$

The gradient estimator will get increasingly precise as $N \to \infty$ and its expected value will become the same as the true gradient. In order to use the gradient estimator in RL, consider a finite horizon MDP, let $R(\tau)$ be the total reward of a segment and $p(\tau|\theta)$

[1] The score function is defined further in Appendix A.

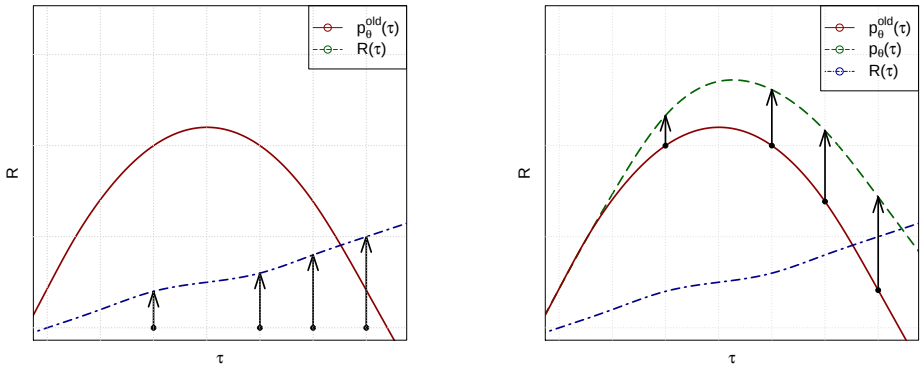


FIGURE 3.2: An allegorical illustration of the score function estimator. First, a set of values is sampled (left) from the current distribution $p^{old}(\tau|\theta)$ (to be seen on the τ axis). Then, the probability distribution is pushed up (right) in proportion to the function values $R(\tau)$ into a new probability distribution $p(\tau|\theta)$. Notice the difference in the arrow length, indicating the proportion of change. The distribution will slide to the right.

denote the parameterized probability of a segment τ with finite horizon T, thus:

$$\nabla_\theta \mathbb{E}_\tau[R(\tau)] = \mathbb{E}_\tau[R(\tau)\nabla_\theta \log p(\tau|\theta)]. \tag{3.6}$$

A schematic illustration of the score function estimator can be found in Figure 3.2. Even though this defines the policy gradient, we cannot use this formula in practice, as we do not know $p(\tau|\theta)$. By breaking up p with the chain rule of probabilities, we get [103]:

$$\begin{aligned} p(\tau|\theta) = \mu(s_0)\pi(a_0|s_0,\theta)P(s_1,r_0|s_0,a_0) \\ \pi(a_1|s_1,\theta)P(s_2,r_1|s_1,a_1) \ldots \\ \pi(a_{T-1}|s_{T-1},\theta)P(s_T,r_{T-1}|s_{T-1},a_{T-1}), \end{aligned} \tag{3.7}$$

where $\mu(s_0)$ is the initial state distribution and T the length of τ.

> Note that $\log(XY) = log(X) + log(Y)$, thus we can turn the product into a sum by taking the log.

Finally, we derive the term w.r.t. θ, which omits $\mu(s_0)$ and $P(s_t, r_{t-1}|s_{t-1}, a_{t-1})$, as they do not depend on θ, and we attain [103]:

$$\nabla_\theta \mathbb{E}_\tau[R(\tau)] = \mathbb{E}_\tau\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta)R(\tau)\right]. \tag{3.8}$$

To calculate the expectation for a single reward r_t, we can rewrite this formula as [103]:

$$\nabla_\theta \mathbb{E}_\tau[r_t] = \mathbb{E}_\tau\left[\sum_{t'=0}^{t} \nabla_\theta \log \pi(a_{t'}|s_{t'},\theta) r_t\right], \tag{3.9}$$

where $t' \leq t$ and r_t can be written in terms of actions $a_{t'}$, so that the final value of the summation becomes t. By summing this equation over the total time T of τ, we attain:

$$\begin{aligned}\nabla_\theta \mathbb{E}_\tau[R(\tau)] &= \mathbb{E}_\tau\left[\left(\sum_{t=0}^{T-1} r_t\right)\left(\sum_{t'=0}^{t} \nabla_\theta \log \pi(a_t|s_t,\theta)\right)\right] \\ &= \mathbb{E}_\tau\left[\left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta)\right)\left(\sum_{t'=t}^{T-1} r_{t'}\right)\right]\end{aligned} \tag{3.10}$$

and our final noisy policy gradient estimate becomes:

$$\hat{g} = \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta)\right)\left(\sum_{t'=t}^{T-1} r_{t'}\right). \tag{3.11}$$

Gradient Update

Policy gradient-based methods use a very similar approach to the vanilla *policy iteration*, in which the algorithm alternates between the process of collecting trajectories with a current policy π [56, 106] and adapting the policy afterward by modifying the policy parameters with a learning rate α such that:

$$\theta_k \leftarrow \theta_k + \alpha \hat{g} \tag{3.12}$$

In words: We move the policy parameters in the direction of the policy gradient estimate $\hat{g}$, using the step size α. In practice, the step size corresponds to the learning rate, and the parameter vector θ is equal to the flattened weights and biases of our neural network. We then adapt the weights iteratively by optimizing the policy gradient loss with, e.g., stochastic gradient descent.

3.1.4 Baseline

A well-known improvement of PG algorithms to address high variance remains to be the subtraction of some baseline function $b(s)$ from the empirical return, so that:

$$\hat{g} = \sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t,\theta) \underbrace{\left(\sum_{t'=t}^{T-1} r_{t'} - b(s_t)\right)}_{\psi_k}. \tag{3.13}$$

Note that we will elaborate ψ_k later in this section.

It has been shown [35] that the introduction of a baseline reduces the overall variance, while not adding any bias. There are many options to represent the baseline, such as

a *constant*, *moving-average* or *second order polynomial*. A better, *near-optimal* [35] choice remains to be the value function:

$$V(s) = \mathbb{E}_{s_t=s, a_{t:(T-1)} \sim \pi} \left[\sum_{t=0}^{(T-1)} r_t \right]. \tag{3.14}$$

When using the value function to approximate the baseline, one may interpret $b(s) \approx V(s)$ in Equation 3.13 as the *better estimate* of the empirical return. Therefore the term $r_{t'} - V(s_t)$ evaluates if a taken action a_t was *better than expected* [104]. This intuitively justifies the application of a baseline as a variance reduction method, since the probability of a taken action is only pushed up by the score function gradient estimator, when the *advantage* over the in average received reward is high.

For large trajectories, such that $\ll T$, Equation 3.13 will have profuse variance. A common approach to avoid this problem is to introduce a discount parameter γ, so that we attain a discounted form of the value function:

$$V^{\gamma}(s_t) := \mathbb{E}_{s_{t+1:\infty}, a_{t:\infty}} \left[\sum_{l=0}^{\infty} \gamma^l r_{t+l} \right]. \tag{3.15}$$

Directly from that follows the discounted policy gradient baseline, defined as:

$$\hat{g} = \sum_{t=0}^{T-1} \nabla_\theta \log \pi(a_t|s_t, \theta) \left(\sum_{t'=t}^{T-1} r_{t'} \gamma^{t'-t} - V^{\gamma}(s_t) \right), \tag{3.16}$$

which reduces variance "by down-weighting rewards corresponding to delayed effects" [104]. Even though PG objectives are typically optimized with *undiscounted* MDP formulations, γ, in this context, is treated as a further hyper-parameter to deduce variance at the cost of bias. This technique was analyzed further by [73, 54].

3.1.5 Actor-Critic

A vast majority of RL algorithms may be categorized [57] into two families.

Actor-only methods optimize parameterized policies. They estimate the gradient of performance w.r.t. the parameters and then optimize the policy locally. A typical drawback of such approaches remains to be excessive variance. On top of that, the gradient is estimated on the current batch of data, independent of previous experience. Experience from older batches may, therefore, not be leveraged.

Critic-only methods are not concerned with the policy gradient. They instead approximate a value function and then derive an optimal policy under the Bellman equation. Pure value-based approaches may not necessarily lead to the optimal policy, as the optimization is dependent on the *fit* of the value space approximation.

The *actor-critic* [57] approach aims to combine the best of both worlds by acting as a hybrid between the two of them. The idea is to have a value-based part, the critic, which measures the expected global long-term performance of a taken action [114] and

to have an actor, a policy-based part, which determines the best action locally. We can represent such actor-critic architecture with two separate non-linear function approximations [114], as illustrated in Figure 3.3. Such an algorithm would collect a sufficiently large batch of experience and then optimize the gradient of Equation 3.16. However, rather than calculating the full expectation of $V^{\gamma}(s)$, we would approximate the discounted sum of rewards with a second neural network under the following objective:

$$\underset{\theta}{minimize} \sum_{t=1}^{N} ||V_{\theta}(s_t) - V_t^{\gamma}||^2, \tag{3.17}$$

where $V_{\theta}(s_t)$ is the output vector of the neural network when using the parameter space θ, and V_t^{γ} the discounted sum of rewards $\sum_{l=0}^{T} \gamma^l r_{t+l}$ for the current segment of T samples.

There are several versions of the policy gradient using the actor-critic approach. Equa-

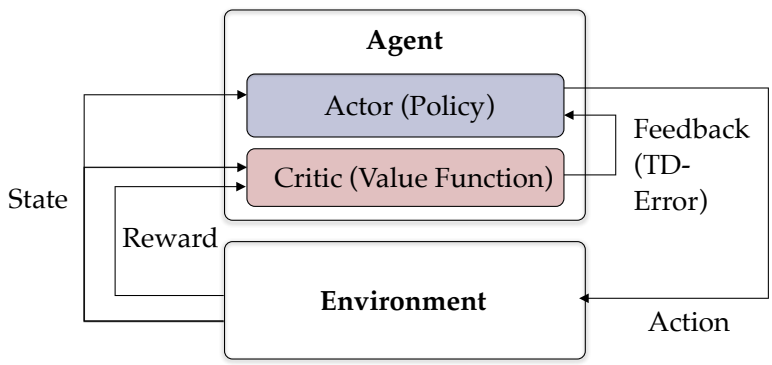


FIGURE 3.3: We see the communication diagram of an actor-critic approach. Notice how the action is purely chosen by the actor, while the critic does not interact with the environment directly; however, it provides feedback as a baseline.

tion 3.13 denotes a version, where ψ_k corresponds to the baselined empirical return of the reward-following action $\sum_{t'=t}^{T-1} r_{t'}$. Considering an infinite horizon MDP, in which $T \to \infty$, the empirical gradient can be replaced [104] with the discounted state-action function of the following equivalence:

$$Q^{\gamma}(s_t, a_t) := \mathbb{E}_{s_{t+1:\infty}, a_{t+1:\infty}} \left[\sum_{l=0}^{\infty} \gamma^l r_{t+l} \right]. \tag{3.18}$$

Note that the subscripts of the expectation denote that we are enumerating the variables which we are integrating over. The state s and the action a are sampled from the environment in accordance with some policy $\pi(a_t|s_t)$. The colon notation $x : y$ indicates an inclusive range between x and y.

One may now subtract the discounted baseline $V^\gamma(s)$ to arrive at the discounted advantage function:

$$A^\gamma(s,a) = Q^\gamma(s,a) - b(s) = Q^\gamma(s,a) - V^\gamma(s). \tag{3.19}$$

The advantage $A^\gamma(\cdot)$ measures if a taken action was better or worse than the policy's default behavior [104]. It follows [54] directly that the gradient of the discounted advantage becomes:

$$\hat{g} = \sum_{t=0}^{\infty} \nabla_\theta \log \pi(a_t|s_t,\theta) A^\gamma(s,a). \tag{3.20}$$

We can observe that, as long as $A^\gamma(\cdot) > 0$ (the action performs better than average), the gradient term $\nabla_\theta \log \pi(a_t|s_t,\theta) A^\gamma(s,a)$ will point in the direction of an improved policy.

Estimating Advantage

The advantage function A^γ consists of two value functions: $Q^\gamma(\cdot)$ and $V^\gamma(\cdot)$. In theory, this suggests one would need two neural networks for the approximation of the advantage on top of the policy network. Since this is impractical, recent publications work around this problem with different solutions. Some authors [124] use a single neural network with *two heads* to estimate the policy and the Q^γ-function at the same time. The value function V^γ is then derived by taking the expectation of Q^γ under the current policy $\pi(a_t|s_t,\theta)$ [124].

Other publications [106, 105, 74, 73] approximate A^γ by producing an estimate $\hat{A}(s_{0:\infty}, a_{0:\infty})$ of the discounted advantage function $A^\gamma(s,a)$ with the *TD residual* [114] $\delta_t^\gamma = r_t + \gamma V^\gamma(s_{t+1}) - V^\gamma(s_t)$. Say, we have a perfect estimate of the discounted value function $V^\gamma(s_t)$, then we can understand the expectation of the TD residual as an *unbiased estimator* of the advantage [104] $A^\gamma(s,a)$:

$$\begin{aligned} \mathbb{E}_{s_{t+1}}\left[\delta_t^\gamma\right] &= \mathbb{E}_{s_{t+1}}\left[r_t + \gamma V^\gamma(s_{t+1}) - V^\gamma(s_t)\right] \\ &= \mathbb{E}_{s_{t+1}}\left[Q^\gamma(s,a) - V^\gamma(s_t)\right] \\ &= A^\gamma(s,a). \end{aligned} \tag{3.21}$$

Note that this term is only unbiased for the perfect discounted value function V^γ.

Generalized Advantage Estimate

Using the undiscounted value function $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$, we can rewrite the advantage estimation as the sum of k of these $\hat{A}$ estimators as:

$$\hat{A}_t^k := \sum_{l=0}^{k-1} \gamma^l \delta_{t+l} = -V(s_t) + r_t + \gamma r_{t+1} + ... + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}), \tag{3.22}$$

in order to derive the λ-return *Generalized Advantage Estimator* (GAE) [104]:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\lambda\gamma)^l \delta_{t+l}. \tag{3.23}$$

Consider [104] of Schulman et al. for a proof and further details. Equation 3.23 is analogous to the well-known TD(λ) approach of Sutton and Barto and allows a smooth interpolation of the advantage estimator between low and high variance. For higher values of λ, the variance increases, but the bias becomes independent of the accuracy of V. For lower values of λ, we attain lower variance, but the term is dependent on the accuracy of V. The variation of $\lambda \rightarrow (0, 1)$ becomes a tradeoff hyper-parameter between variance and bias.

To conclude, Schulman et al. considered [104] the following replacements for the empirical baselined return $\psi_1 = \sum_{t'=t}^{T-1} r_{t'} - b(s_t)$ of Equation 3.13 as an estimator of $\hat{A}$:

- ψ_2 : The Q-function: $Q^\gamma(s, a)$
- ψ_3 : The advantage function: $A^\gamma(s, a)$
- ψ_4 : The TD-residual: $r_t + \gamma V^\gamma(s_{t+1}) - V^\gamma(s_t)$
- ψ_5 : The Generalised Advantage Estimator: $\sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$

Note that all of these terms are unbiased estimators. They will, however, have different variance and in practice, since neural networks approximate these quantities, they will have bias induced by the approximation error of the stochastic neural network [127]. Because of that, all of the latest introduced novel methods have different correction terms for bias and variance, appropriate to the utilized version of the advantage estimator.

The usage of a single discounted value function V^γ (ψ_4, ψ_5) can have several advantages [104] over an estimator containing the discounted Q-function estimator (ψ_2, ψ_3). First of all, a neural network approximating $Q^\gamma(s, a)$ is of higher dimension due to the action-vector. As a result, it is more difficult to approximate $\hat{A}$. Second, the learning process theoretically involves three non-linear approximations, one for $V^\gamma(\cdot)$, one for $Q^\gamma(\cdot)$, and a policy network. This overhead increases the complexity further and requires the adjustment of hyper-parameters w.r.t. the other neural networks. Third, the GAE allows a smooth and straightforward interpolation between high-bias estimation ($\lambda = 0$) and a low-bias estimation ($\lambda = 1$). With a Q^γ-function, similar lower-bias estimations can be achieved by replacing the 1-step distributional loss with a multi-step variant [114, 48]. The integration [124] of such is, however, not as trivial as the concise formula of the GAE and involves further correction terms [124].

3.2 Approximating in Policy Space

Gradient-based methods were designed to optimize stochastic policies directly with non-linear approximations. In comparison to value-based approaches, gradient-based

techniques come much closer to supervised machine learning approaches. Nevertheless, these methods come with a set of challenges, which had made it previously difficult to apply PG methods to very high-dimensional domains, like robotics. Recently, a variety of different PG-based ways have been proposed, capable of handling the challenges of extensive parameterized policies. In this section, we are concerned with problems and solutions to gradient-based optimization of stochastic policies, specifically in the context of large parameterized non-linear approximations. We start by discussing which components of a stochastic policy are approximated and address occurring challenges. We then see that the classical policy gradient equation can be rewritten as a *surrogate objective*, using *importance sampling* [96].

> The term *surrogate model* is usually employed in optimization theory when constructing an approximation model for an outcome of interest that cannot be easily measured.

On top of that, we discuss an alternative to *gradient descent*, which is the optimization of *a natural policy gradient* [53], by following the *steepest* descent direction of the underlying optimization surface. From there, we recognize that by estimating a *lower pessimistic bound* of the policy performance, we can achieve a *guaranteed positive or constant* policy performance improvement. Ultimately, we present two unconstrained proximal policy optimization objectives.

3.2.1 What to Approximate ?

In Chapter 2, we considered mostly deterministic policies $a = \pi(s)$ for episodic reinforcement learning problems. In Chapter 3, we focus on *stochastic policies*, which are conditional distributions $\pi(a|s)$. We can optimize such a conditional distribution with a neural network, by approximating a multivariate Gaussian distribution $\pi_\theta(a|s)$, so that, the parameter vector $\theta = \{W_i, b_i\}_{i=1}^{L}$ corresponds to the flattened weights W and biases b of our neural network [115, 106].

In *continuous* action spaces, one may then optimize the mean and the covariance of this multivariate normal distribution with an arbitrary loss function [37]. Alternatively, we can use the neural network only to represent the mean of this distribution and define a separate set of parameters that specify a log standard deviation of each input feature [106]. In such an approach, the policy is then established by:

$$\varphi(\mu = NeuralNetwork(s;\theta), \sigma^2 = exp(r)), \tag{3.24}$$

where r corresponds to a vector of standard deviations with the same dimension as the action space.

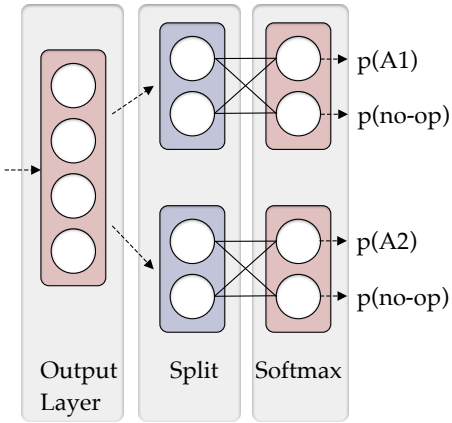


FIGURE 3.4: The neural network representation of a discrete two-dimensional action space. The output of the final, fully connected layer is first split, and then each tuple $(action, no-op)$ pushed through a final softmax layer that outputs the probability of each identity.

For *discrete* action spaces, we represent the probability of each action by a final *softmax* layer. Suppose we have two possible discrete actions, a_1 and a_2 that can either be triggered or not triggered, then the output layer of the neural network will have four dimensions, each corresponding to either the action or no operation, as illustrated in Figure 3.4.

3.2.2 Challenges

PG methods have been known for quite a long time [115]. The vanilla version of PG has a couple of practical limitations, which made it often the wrong choice for many problems [56, 103]. However, similar to the increasing popularity of value-based deep RL, many new policy gradient improvements have been introduced in the last couple of years, leveraging the success of deep learning. The latest proposed solutions agree on the following problems [127, 124, 68, 106, 105] of the vanilla policy gradient to be solved.

High Variance
The score function estimator returns noisy estimates, which have high variance. Since we estimate the policy gradient, current solutions focus on *variance reduction* to increase the precision of the sampled estimation. A lower accuracy causes the learning process to slow down, as wrong action choices impact the sampling space of the policy significantly.

Policy Change
Extensive policy updates can also be destructive for the learning process. As the gradient estimation relies on the current sampled state batch, a large policy update can lead to an entirely different sampled state space, which can cause the algorithm to converge to a near-deterministic policy prematurely.

Step Size
Subsequent to the previous problem, it is challenging to find the right learning

rate for the neural network that works over the entire course of learning. The success of the developed algorithm, consequently, depends on a single hyper-parameter, which is hard to adjust correctly, especially if the environment changes.

Low Sample Efficiency
PG methods learn offline. Meaning they need to sample the current policy for a while before they get to update the policy. On top of that, old data from previous policies cannot be reused, when using an on-policy setting, which makes the algorithm short-sighted and sample inefficient. This effect increases the cost of simulation and is even more impractical for the real-life applications of PG methods.

3.2.3 Surrogate Objective

Recalling a policy gradient loss with an advantage estimator from Section 3.1.5, we obtain the following empirical expectation:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi(a_t|s_t, \theta) \hat{A}_t \right], \tag{3.25}$$

where $\hat{\mathbb{E}}_t[...]$ denotes the empirical average over a finite batch of samples in an algorithm that shifts between sampling and optimization. With this gradient, one may now differentiate this gradient and update the policy network with the following policy gradient loss [105]:

$$L^{PG} = \hat{\mathbb{E}}_t \left[\log \pi(a_t|s_t, \theta) \hat{A}_t \right]. \tag{3.26}$$

Note that we can numerically evaluate the gradient by using *automatic differentiation* since we directly sample from the expectation. This type of evaluation if supported by most modern machine learning libraries (e.g., TensorFlow), which works by repeatedly applying the *chain rule* to compute the derivate of an arbitrary function.

In Section 3.1.5, we have seen that, by replacing the empirical estimate with an estimated advantage, we can considerably reduce the variance. However, independent of which ψ_k has been chosen for the policy gradient, another major problem remains *sample inefficiency*, as we are not able to reuse sampled data from an old policy. *Importance sampling* [96] allows the estimation of some new policy parameter vector θ w.r.t. an old parameter vector θ_{old}, calculated in the previous policy iteration. We can rewrite the

policy gradient with the chain rule at the parameter space $\theta = \theta_{old}$:

$$\begin{aligned}
\hat{g} &= \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi(a_t|s_t,\theta) \hat{A}_t \right] \\
&= \int dx \; \pi(a|s,\theta) \nabla_\theta \log \pi(a|s,\theta) \mid_{\theta_{old}} \hat{A} \\
&= \int dx \; \pi(a|s,\theta) \frac{\nabla_\theta \pi(a|s,\theta) \mid_{\theta_{old}}}{\pi(a|s,\theta_{old})} \hat{A} \\
&= \int dx \; \pi(a|s,\theta) \nabla_\theta \left(\frac{\pi(a|s,\theta)}{\pi(a|s,\theta_{old})} \right) \mid_{\theta_{old}} \hat{A} \\
&= \hat{\mathbb{E}}_t \left[\nabla_\theta \frac{\pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta_{old})} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[\nabla_\theta \; r_t(\theta) \hat{A}_t \right],
\end{aligned} \tag{3.27}$$

and use it as an importance sampling estimator. Notice that we replaced the probability ratio $(\pi(a_t|s_t,\theta))/(\pi(a_t|s_t,\theta_{old}))$ with the abbreviatory function $r_t(\theta)$. The importance sampling estimator turns the term into a *surrogate* objective function [105], which is to be optimized. What makes this rearrangement interesting is that, in comparison to the default policy gradient term notation, we attain a ratio between the old policy $\pi_{\theta_{old}}$ and the new estimated policy π_θ. We will later see how current deep PG methods use a probability distance measure to constraint the step size with this ratio.

3.2.4 Natural Policy Gradient

When learning with large non-linear neural networks through methods such as importance sampling, we aim to find an optimal policy π_θ in restricted policy space by following a gradient of the expected cumulative future reward. One could now follow this gradient with a first-order optimization, such as gradient descent. Unfortunately, with standard SGD, we have no guarantee that our next policy π_θ will be similar to our previous policy $\pi_{\theta_{old}}$, and we may end up in entirely different parameter space, from where the new policy will eventually never be able to recover. This outcome constitutes no problem for classical supervised learning methods, as their underlying distribution is typically fixed [87] [73]. However, it certainly does in the reinforcement learning setting, as our learning samples are in direct dependence of the policy. This relation has been identified [53] [106] [87] as one of the main challenges of policy gradient-based optimization when using large function approximations with SGD.

A first solution was proposed as the *natural policy gradient* [53]. Consider a negative gradient $-\nabla_\theta E(\theta)$ of an arbitrary defined parameterized loss function $E(\theta)$, which we want to follow in the *steepest* descent direction around the local neighborhood of our current parameter vector θ. We can than geometrically interpret [73] the change in θ with the standard Euclidean norm $||\cdot||$ as:

$$\frac{-\nabla_\theta E(\theta)}{||\nabla_\theta E(\theta)||} = \lim_{\epsilon \to 0} \frac{1}{\epsilon} \underset{v:||v|| \leq \epsilon}{\arg\min} E(\theta + v). \tag{3.28}$$

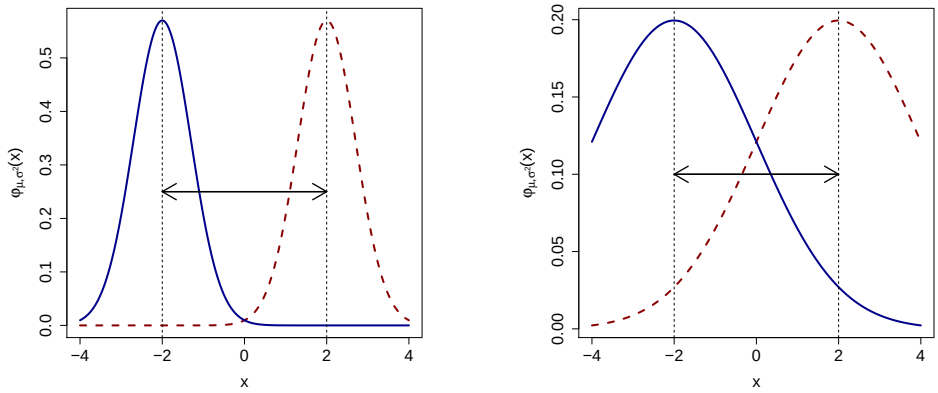


FIGURE 3.5: All distributions shown in this image are normal distributions. The left side corresponds to $\mu = \pm 2$ and $\sigma = 0.7$ while the right distributions correspond to $\mu = \pm 2$ and $\sigma = 2$. The black arrow in the middle is the Euclidean distance between the expectation of each pair.

A change of θ with some vector v must, therefore, be within the ϵ-neighborhood of θ in parameter space to follow the steepest direction. Note that this interpretation is bound to a strong dependence of the gradient on the Euclidean geometry [73]. Unfortunately, gradient descent is *non-covariant* and, therefore, *dimensionally inconsistent* [53], as the gradient differs for every parameterization θ of π_θ. Consider, for example, Figure 3.5. It can easily be verified that both pairs of distributions have different distances in convex probability space, even though their Euclidean distance of the expectations is the same. In order to find a covariant gradient, we must use a different metric than the Euclidean distance for probabilistic approximations. *Natural gradient descent* [53] solves this problem by using the Fisher [71] information matrix of the policy distribution $\pi(a|s,\theta)$, defined as:

$$\mathcal{F}_\theta = \mathbb{E}_{\pi(a|s,\theta)}\left[(\nabla_\theta log\pi(a|s,\theta))^T(\nabla_\theta log\pi(a|s,\theta))\right], \tag{3.29}$$

in order to define an *invariant* metric in the probability distribution parameter space [53]. The steepest covariant descent direction is then simply defined by $\mathcal{F}_\theta^{-1}\hat{g}$.

A metric is *invariant* when it defines the same distance between two points, independent of its coordinates. Here, we are interested in a metric in the space of the parameters of probability distributions. Therefore, the coordinates correspond to the parameterization θ.

3.2.5 Trust Region Policy Optimization

Let us consider policy gradient optimization from the perspective of the classical policy iteration scheme, in which the ultimate goal is to improve our policy in every policy iteration. For such policy improvement, Kakade and Langford showed [54] that the

advantage measured in the expected discounted reward η over a policy π of a successor policy π' of the next policy improvement can be expressed by the following equation:

$$\begin{aligned}\eta(\pi') &= \eta(\pi) + \mathbb{E}_{s_0,a_0,\ldots\sim\pi'}\left[\sum_{t=0}^{\infty}\gamma^t A(s_t,a_t)\right] \\ &= \eta(\pi) + \sum_s p_{\pi'}(s)\sum_a \pi'(a|s)A(s,a).\end{aligned} \tag{3.30}$$

This equation has been rewritten [106] in line 2 with the unnormalized discounted visitation frequency in terms of the transition probability distribution $P : S \times A \times S$, where

$$p_\pi(s) = P(s_0) + \gamma P(s_1) + \gamma^2 P(s_1) + \ldots . \tag{3.31}$$

With Equation 3.30, the policy improvement becomes guaranteed positive or constant, if there is at least one state-action pair with a non-negative advantage and a visitation frequency of $p(s) > 0$ [106]. However, if, due to approximation errors, some of the advantage values with a visitation frequency $p(s) > 0$ become smaller than zero, the policy improvement might become negative. In order to solve this issue, Kakade and Langford first derived a practical local approximation term for η using $p_\pi(s)$ instead of $p_{\pi'}(s)$:

$$L_\pi(\pi') = \eta(\pi) + \sum_s p_\pi(s)\sum_a \pi'(a|s)A(s,a). \tag{3.32}$$

Given a differentiable parameterized policy π_θ, Kakade and Langford then showed [54] that L_{π_θ} equals η to first order for any parameter value θ_0 such that: $L_{\pi_{\theta_0}}(\pi_{\theta_0}) = \eta(\pi_{\theta_0})$ and $\nabla_\theta L_{\pi_{\theta_0}}(\pi_{\theta_0})|_{\theta=\theta_0} = \nabla_\theta \eta(\pi_\theta)|_{\theta=\theta_0}$. Directly from that follows that a sufficiently small step in θ, improving the local approximation $L_{\pi_{\theta_0}}$, will also improve η. However, this does not hold any information about how small the step must be. As a result, Schulman et al. proved the following theoretical bound on the policy change, using the maximal KL divergence $D_{KL}^{max}(\pi,\pi') = \max_s D_{KL}(\pi(\cdot|s) \,||\, \pi'(\cdot|s))$ between the new and the old policy distribution:

$$\begin{aligned}&\eta(\pi') \geq L_\pi(\pi') - CD_{KL}^{max}(\pi,\pi'), \\ &where\ C = \frac{4\,\epsilon\gamma}{(1-\gamma)^2}, \\ &where\ \epsilon = \max_{s,a}|A(s,a)|.\end{aligned} \tag{3.33}$$

Considering a *parameterized* policy π_θ, one can maximize the right part of the equation: $\underset{\theta}{maximize}\big[L_\theta(\theta') - CD_{KL}^{max}(\theta,\theta')\big]$ in order to find the next set of parameters. This *Minorization-Maximization* (MM) [61] algorithm guarantees a non-decreasing true objective η for each policy iteration [106], as illustrated in Figure 3.6.

Note that we slightly abuse the policy notation π_θ by replacing it with its parameters θ such that e.g., $L_\theta(\theta') := L_{\pi_\theta}(\pi_{\theta'})$ and $\eta(\theta) := \eta(\pi_\theta)$, where θ corresponds to the previous policy parameters and θ' are the current parameters of the policy.

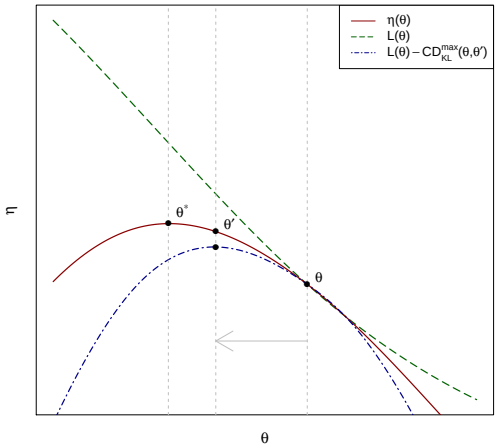


FIGURE 3.6: The MM approach of TRPO constraints the parameter change in θ for the approximation $L(\theta)$ at every point in the state space. Notice how the true objective $\eta(\theta)$ is increasing for step sizes smaller than $\theta^* - \theta$ and decreasing for step sizes above. By maximizing $\eta(\pi') \geq L_\theta(\theta') - CD_{KL}^{max}(\theta, \theta')$ instead of $L(\theta)$, the change in θ will always stay sufficiently small, as we attain a lower *pessimistic* bound on the policy performance.

While the theory suggests using a penalty coefficient C on the maximum KL divergence of the policies, the practical implementation, which is called Trust Region Policy Optimization (TRPO) [106], differs from that point of view. First, the coefficient C can lead to excessively small updates, and it is hard to choose the correct γ that performs well across different domains. Second, D_{KL}^{max} is difficult to optimize directly, as it is bound to every point in the state space. The practical implementation of TRPO, therefore, rather optimizes the estimate $L_\theta(\theta')$ subject to a constraint on a *heuristic approximation* of the mean KL divergence in accordance with:

$$\begin{aligned} &\underset{\theta'}{maximize}\ L_\theta(\theta') \\ &s.t.\ \mathbb{E}_{s\sim\pi_\theta}[D_{KL}(\pi(a|s,\theta)||\pi(a|s,\theta'))] \leq \delta, \end{aligned} \tag{3.34}$$

where δ is a hyper-parameter. Note that in Equation 3.34, the KL divergence becomes an empirical average between policies across states sampled with the old policy $s \sim \pi_\theta$. The optimization problem of Equation 3.34 may now be solved using the *conjugate gradient method* [77] by first calculating the search direction (1) with a linear approximation on the objective $L_\theta(\theta')$ and a quadratic approximation to the KL constraint; and then by performing (2) a *backtracking line search* (BLS) [3] in the obtained direction. In practice, we sample advantage values. For a practical algorithm, TRPO, hence, replaces the objective $L_\theta(\theta')$ with the approximated *surrogate objective* of Equation 3.27 using Monte

Carlo simulation. The final approximated TRPO objective [105] at a timestep t becomes:

$$\begin{aligned} &\underset{\theta'}{maximize}\ \hat{\mathbb{E}}_t\left[\frac{\pi(a_t|s_t,\theta')}{\pi(a_t|s_t,\theta)}\hat{A}_t\right] \\ &s.t.\ \hat{\mathbb{E}}_t[D_{KL}(\pi(a_t|s_t,\theta)||\pi(a_t|s_t,\theta'))] \leq \delta. \end{aligned} \tag{3.35}$$

TRPO approximates its search direction with a quadratic approximation to the KL divergence constraint: $\overline{D_{KL}}(\theta,\theta') \approx \frac{1}{2}(\theta'-\theta)^T\mathcal{F}(\theta'-\theta)$, where $\mathcal{F}$ is the Fisher[2] information matrix, which is computed with the second derivative (the Hessian) of the KL divergence between the new and the old policy $\mathcal{F}_{i,j} = \nabla_i\nabla_j\overline{D_{KL}}(\theta,\theta')$[3]. For deep neural networks, this is impractical due to the large number of parameters. TRPO approximates the constraint by using the conjugate gradient algorithm; however, calculating $\mathcal{F}$ in every iteration is still expensive and requires a large batch of rollouts to approximate the constraint correctly. The empirical results confirm this, as TRPO has not shown good performance on high-dimensional RL problems such as the Atari domain [105, 127, 124].

3.2.6 Proximal Policy Optimization

To avoid moving the new policy in the next policy iteration too far away from the old policy, TRPO uses a hard constraint to obtain a *pessimistic* lower bound on the true policy performance while maximizing the *surrogate* objective of the conservative policy iteration:

$$L_{\theta_{old}}(\theta) := \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right] = \hat{\mathbb{E}}_t\left[\frac{\pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta_{old})}\hat{A}_t\right], \tag{3.36}$$

Note that $r(\theta) = 1$ for $\theta = \theta_{old}$, which is the point in parameter space where $L_{\theta_{old}}(\theta)$ matches η to first order around θ_{old}, as illustrated in Figure 3.6. The theory of TRPO suggests a penalty for policy changes that move r away from $r = 1$.

> To ensure readability, we fluently switch between the parameter notations $(\theta_1,\theta_2) \equiv (\theta_{old},\theta) \cup (\theta,\theta')$, where θ_1 corresponds to the previous parameter vector and θ_2 to the current set of parameters, but remind the reader when necessary.

Proximal Policy Optimization reformulates the penalty as an unconstrained optimization problem, using a clipped objective instead:

$$L^{CLIP}(\theta) = \mathbb{E}_t\left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta),1-\epsilon,1+\epsilon)\hat{A}_t)\right], \tag{3.37}$$

where $\epsilon \in [0,1]$ is a hyper-parameter of choice. In order to avoid changes in the policy that move the ratio r outside of the interval $[1-\epsilon,1+\epsilon]$, Proximal Policy Optimization uses the expected minimum between the clipped and unclipped surrogate objective. This method includes the change in r when the objective gets worse and ignores it when

[2]The Fisher is defined in Appendix A.

[3]We refer to Appendix A for a more comprehensive definition of the relationship between KL divergence, Hessian, and Fisher. Furthermore details on this computation can be found in Appendix C of [106].

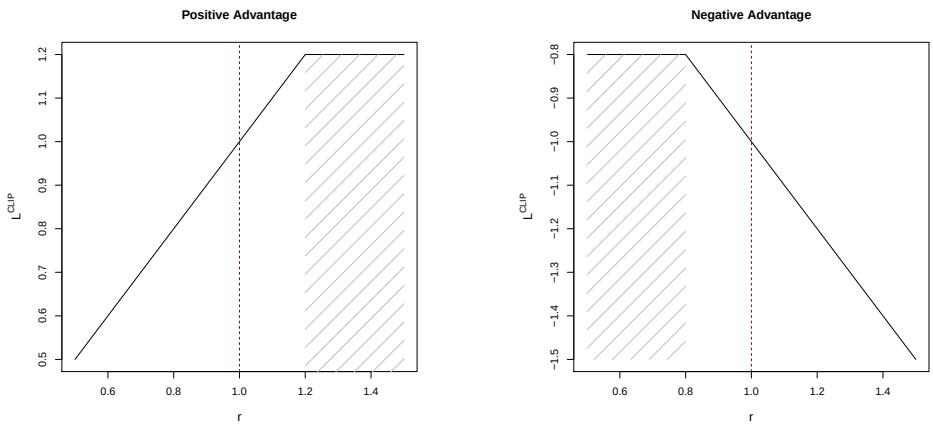


FIGURE 3.7: in accordance with [105], this plot shows L^{CLIP} as a function of r with $\epsilon = 0.2$ for a fixed advantage $A = 1$ on the left and $A = -1$ on the right. The dashed vertical line at $r = 1$ shows parameter space where L matches L^{CLIP} to first order. The grey area under the curve indicates where the negative or positive change of the probability ratio r is ignored.

the objective improves. Figure 3.7 illustrates the shift in r for positive and negative advantage values. Schulman et al. also proposed a second unconstrained optimization problem with a penalty instead, which reformulates the constraint TRPO objective as:

$$L^{KLPEN}(\theta) = \mathbb{E}_t \left[r_t \hat{A}_t - \beta \, D_{KL}(\pi(a_t|s_t, \theta_{old})||\pi(a_t|s_t, \theta)) \right]. \tag{3.38}$$

The right part of the equation, βD_{KL}, becomes the penalty, where the coefficient β is calculated at every policy iteration in order to move the KL divergence in the direction of the desired target KL divergence D_{KL}^T. β is then calculated according to the following algorithm:

$$\beta = \begin{cases} \frac{\beta}{\omega}, & \text{if } D_{KL} < \frac{D_{KL}^T}{\nu} \\ \omega\beta, & \text{if } D_{KL} > \nu D_{KL}^T, \end{cases} \tag{3.39}$$

and used for the next policy update, where ν and ω are hyper-parameters (say, $\omega = 2$ and $\nu = 1.5$). To derive a practical algorithm from these two approaches, one can now simply numerically evaluate L^{KLPEN} or L^{CLIP} with an automatic differentiation library and perform multiple steps of *mini-batch SGD* [97] with *Adam* [55] on it.

On top of the different loss functions, the PPO algorithm family can be enhanced by multiple improvements. First, when using a neural network architecture that uses a shared parameter model in between the actor and the critic, the objective can be further enhanced by adding the squared loss of the value network $L_t^{VF}(\theta) = ||V_\theta(s_t) - V_t^\gamma||^2$ to the policy objective. Second, it is further advised for some environments such as the Atari domain, to add a policy entropy[4] bonus H_{π_θ}, more precisely the entropy of the

[4]The entropy is discussed in Appendix A.

policy π_θ, in order to support exploration. Finally, we arrive at the following improved loss function:

$$L^{TOTAL}(\theta) = \hat{\mathbb{E}}_t \left[L^{CLIP} - x_1 L_t^{VF}(\theta) + x_2 H_{\pi_\theta} \right], \tag{3.40}$$

where the coefficients x_1 and x_2 become two hyper-parameters controlling the loss adjustments.

Going further, Lillicrap et al. proposed [68] the usage of *fixed-length trajectory segments*. This asynchronous approach uses a fixed timestep horizon T instead of a fixed amount of episodes for the batch-size of the policy network. The rollouts will be collected asynchronously from multiple agents N in parallel, whereas the policy loss will be calculated by NT timesteps of experiences [68]. Since a limitation of the actor batch size by a fixed timestep horizon will inevitably truncate trajectories before their terminal state, the advantage of the remaining timesteps is approximated with the value function V by a truncated form of the GAE [105]. Such an algorithm can then be developed according to the following pseudocode, which is partially adopted from [45], where M and B represent the number of optimizer episodes for each gradient:

Algorithm 2 Proximal Policy Optimization

1: **for all** $i \in N$ **do**
2: Run policy π_θ for T timesteps, collecting $\{s_t, a_t, r_t\}$
3: Estimate advantages $\hat{A}_t = \sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t)$
4: **for all** $j \in B$ **do**
5: $L^{CLIP}(\theta) = \mathbb{E}_t \left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right]$
6: Update θ w.r.t. L^{CLIP} by gradient method
7: **end for**
8: **for all** $j \in M$ **do**
9: $L^{VF}(\phi) = ||V_\phi(s_t) - V_t^\gamma||^2$
10: Update ϕ w.r.t. L_t^{VF} by gradient method
11: **end for**
12: **end for**

3.3 Literature Review

Figure 3.8 shows a timeline of publicly available baseline implementations, which are frequently compared against novel methods. Note that, in comparison to value-based deep RL, policy gradient-based deep RL comes with a better theoretical understanding and analysis of the learning process. On the other hand, we must not forget that the proposed methods are still very new. Hence new emerged hyper-parameters and the combination with other RL methods are not very well understood. Going into detail on each of these algorithms is beyond the scope of this thesis, however, a brief overview and categorization of Deterministic Policy Gradient (DPG) [109], Deep Deterministic Policy Gradient (DDPG) [68], Trust Region Policy Optimization [106], Sample

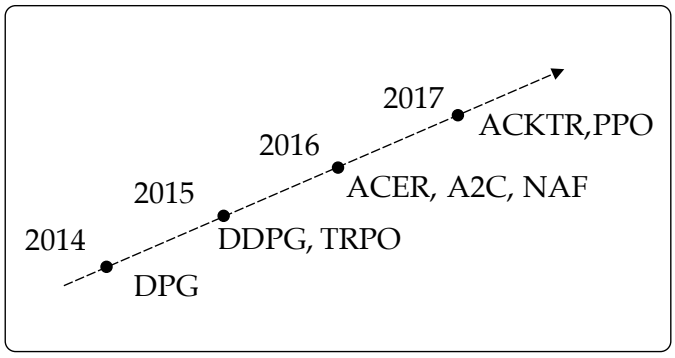


FIGURE 3.8: A timeline of recent model-free policy gradient algorithms.

Efficient Actor-Critic (ACER) [132], Advantage Actor-Critic (A2C) [74], Actor-Critic using Kronecker-Factored Trust Region (ACKTR) [127] and Proximal Policy Optimization [105] will be given. Figure 3.9 classifies the different off- and on-policy improvements. While all of them rely on the actor-critic model, we can examine two categories for policy gradient methods with deep function approximation.

3.3.1 Existing Knowledge Transfer

The first category builds upon *existing knowledge* gathered from value-based deep RL and actor-critic methods used with small neural networks. DPG, DDPG, A2C are very similar approaches to the Neural Fitted Q Iteration with Continuous Actions (NFQCA) [38], which uses batch learning to stabilize the training. NFQCA optimizes the policy by maximizing a Q-function based on Monte-Carlo rollouts. However, Lillicrap et al. showed [68] that batch learning turns out to be unstable with large neural networks and does, therefore, not scale sufficiently enough on more challenging problems. Hence, DPG was proposed, which uses the same update rule, but leverages mini-batch learning [97] with batch normalization. Inspired by the success of DQN, DDPG further improves the method by using a target network with "soft" target updates, rather than hard copies of the weights. A2C uses an on-policy method for the policy network and an off-policy value-based method. On-policy policy networks are sample inefficient, A2C was developed to bypass this problem by running multiple actor-learners in parallel [74] in order to explore a more versatile state space of the environment. Furthermore, the need for horizontal scalability was addressed by using large-scale cloud server architectures, which can be essential to extend deep RL on more complicated problems.

3.3.2 Trust Region Methods

The second category uses Trust Region Methods (TRM) [16]. TRMs are a well known mathematical optimization for non-linear problems. These methods work by solving a constraint optimization problem. By comparing Figure 3.8 and Figure 3.9, one can observe that, since the proposal of TRPO in 2015, the optimization of TRM on actor-critic methods was picked up by the research community. TRPO constructs a stochastic

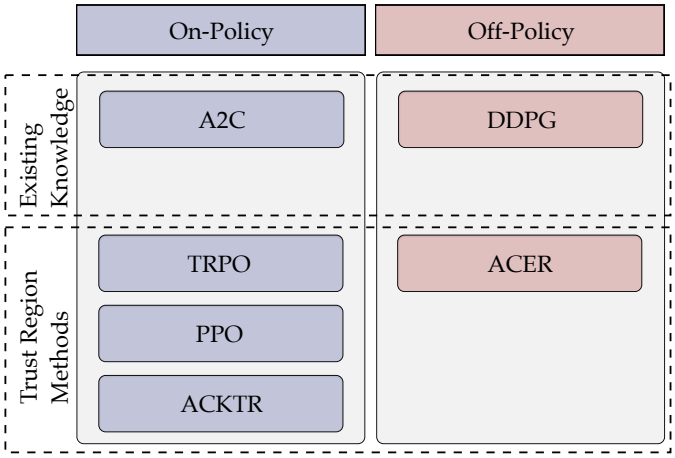


FIGURE 3.9: An overview and categorization of the latest state-of-the-art RL methods for continuous action spaces.

neural network policy directly, with a guaranteed monotonic improvement. The essential idea is to constrain the policy change on each policy iteration to avoid destructive updates. Even though TRPO tends to perform well on many continuous control problems, it is not competitively enough [124, 127, 105] in domains that involve a high-dimensional state (e.g., Atari). On top of that, it requires repeated computation of computationally expensive Fisher-vector products for each iteration of conjugate gradient descent [77].

A large number of parameters in deep neural networks make the exact calculation of the Fisher matrix and it's inverse intractable. Consequently, several novel techniques have been proposed to approximate the Fisher matrix efficiently. ACER can be interpreted as the off-policy counterpart of A2C, since it equally scales up asynchronously, but uses *Retrace* [80], an off-policy truncated *importance sampling* method, to obtain the policy gradient. Moreover, a combination of stochastic dueling networks, trust region policy optimization as well as a bias correction term [124], have been introduced. ACKTR avoids the repeated calculation of Fisher vector products by using tractable Fisher matrix approximations, a recently proposed technique called Kronecker-factored approximate curvature [127]. On top of that, an asynchronous computation of the second-order curvature statistic is collected during training [127] as a running average, which is required by the Kronecker approximation to reduce computation time further.

Instead of limiting the maximum divergence from the old policy with a hard KL constraint, PPO clips the TRPO objective with a *lower pessimistic bound* [105]. This clipping constraint avoids the approximation of the KL divergence in the first place and tends to be much more sample efficient, robust, and compatible with large state dimensions [105].

3.3.3 On- and Off-Policy Policy Gradient Algorithms

Besides the horizontal categorization into *existing knowledge* and *trust region* methods, Figure 3.9 reflects a vertical classification in off- and on-policy actor-critic algorithms. The vanilla actor-critic algorithm is known to be on policy, as the current policy is optimized based on samples generated by a current *target policy* π. An off-policy version of this algorithm would store experiences into memory and update the policy gradient w.r.t. experiences drawn from a different *behavior policy* μ. This approach is, however, not trivial, as small changes in the action choices can cause substantial changes in the policy, which makes the estimation of the policy gradient difficult [17]. A common approach [65, 17, 124] to obtain the off-policy gradient, nevertheless, is to employ a separate *behavior policy* and draw samples from it via *importance sampling*. We can then weight the the policy gradient estimates with an importance weight $p = (\pi(a|s))/(\mu(a|s))$, a ratio between the *target policy* π and the *behavior policy* μ, throughout all off-policy samples.

Although convergence and guaranteed policy improvement have been shown [17] for the tabular form, controlling the stability of such an algorithm is notoriously hard as the importance sampling procedure emits very high variance [124]. ACER uses *Retrace* [80], a truncated version of importance sampling, that has much lower variance and is proven to converge in tabular form to the target policy value function for any behavior policy [80, 124]. DDPG uses a *deterministic* target policy, generated by trajectories from a stochastic behavior policy. While actor-critic algorithms typically sample an action from a parametric probability distribution $\pi_\theta(a|s)$, a deterministic policy [109] $\mu_\theta^{det}(s) = a$ maps directly from state into action space. The deterministic policy avoids importance sampling in the first place for actor and critic, but still allows the algorithm to sample from a behavior distribution off-policy to estimate action-value function and policy gradient [109].

3.4 Summary

This chapter focused on gradient-based deep reinforcement learning. First, we categorized policy gradient-based optimization in the family of policy search algorithms. Following that, we talked about how the PG allows the direct optimization of the expected total reward and how this concept may be married with value-based reinforcement learning by using the value function as a baseline. Section 3.2 represents one of the theoretical core contributions of this work, where we discussed challenges and solutions when applying deep learning to PG-based optimization. We identified the first-order optimization: *gradient descent* as one of the main predicaments and considered the constraint optimization TRPO as a possible solution. TRPO instead uses a type of *conjugate gradient descent*, to approximate a guaranteed positive or constant objective. Finally, we introduced two practical unconstrained optimization problems, representing the family of PPO algorithms, which we will be using in Chapter 4 for follow-up experiments. We closed the chapter by offering the reader a comprehensive review of related research topics.

Chapter 4

Practical Experiments

In this chapter, we aim to support the theoretical results obtained in Chapter 2 and Chapter 3 with practical experiments and examples. Deep RL algorithms have become complicated, and the implementation of such is seldom straightforward. In order to maintain the rapid progress of deep reinforcement learning, it is essential to preserve a high degree of transparency, to enable reproducibility [46], and to strive for simplicity [1]. This section contributes to this matter with two central empirical examinations. First, in Section 4.5, we investigate the magnitude of the effect of hyper-parameters. We argue that the impact of hyper-parameter choices is often not appropriately emphasized, even though parameters vary considerably amongst publications. Second, Section 4.6 explores the origin of the success of deep RL: The size of the neural network. We will find that the architecture of the neural network is one of the most reluctantly discussed and concurrently also one of the most difficult hyper-parameter collections to get right. Neural network architectures differ considerably amongst deep RL publications. Related research [46] has shown over a range of different deep RL algorithms that even simple architecture changes can have a significant impact on the policy performance, although choices of shape, size, activation function, and step size are rarely sufficiently justified or explained amongst publications. As a result, we aim to investigate the influence of the neural network, and it's hyper-parameters on policy performance. Moreover, specifically on the example of policy gradient-based optimization, we demonstrate that a change in the size of the applied neural network has a direct influence on the policy change. Finally, we propose an initial recommendation on how to adapt the step size in order to contribute to a more scalable and dynamic foundation for the neural network design of future research.

4.1 Experimental Framework

For empirical evaluations, an actor-critic algorithm was chosen, specifically PPO. Figure 4.1 shows the evaluation approach. First, PPO was implemented based on the knowledge provided by the literature. Then, an *Evaluation Framework*, which executed different trials with different hyper-parameter configurations, has been developed. It processed the various log files and generated the evaluation plots. A second component, which we refer to as *Network Factory*, was responsible for creating different shapes and sizes for the neural networks of actor and critic. Experiences gathered from the algorithm implementation, as well as empirical results obtained from the evaluation, are presented in this chapter.

For experiments, an actor-critic approach was preferred over a pure critic-only ap-

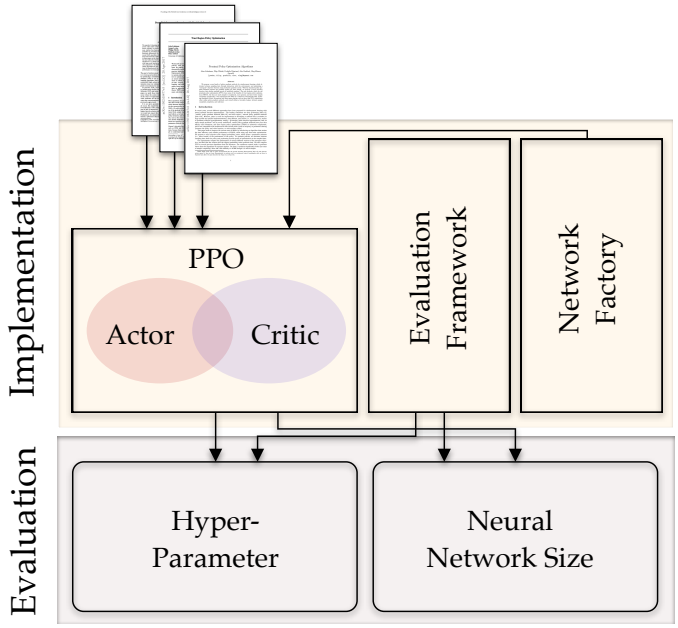


FIGURE 4.1: We see the research approach for the practical chapter of this thesis. The arrows indicate the dependence of the different components.

proach for multiple reasons. First of all, *classic control* environments of the *OpenAi Gym* can be considered not sophisticated enough to justify the use of deep neural networks. The Multi-Joint dynamics with Contact (MuJoCo) [118] environments provide a more challenging and more realistic framework with a high-dimensional and continuous state space, as typical in real-life domains like *robotics*. Furthermore, actor-critic based methods can be found to be less vulnerable to hyper-parameter changes. Off-policy value-based methods have become complicated, to an extent where research even started to question the complexity of these methods [1, 91]. Take, for example, the value-based improvement *prioritized experience replay* [100]. The authors hypothesize in their publication that deep neural networks interact entirely differently with their prioritized replay memory, hence allowing to reduce the network error much quicker. Other extensions, such as *NoisyNets* [26], apply some *regularization*, where they perturb the weight and biases of the neural network to encourage exploration. Examples like this show that we have not fully understood the different critic-only improvements and their interaction with deep neural networks yet. Gradient-based methods optimize stochastic policies directly, which makes RL much more like other domains where deep learning is used. On top of that, *simplicity* [105] has been a principal design goal of algorithms such as PPO, which we desperately need, given the level of intricacy of current deep reinforcement learning solutions.

4.2 Environments

Experiments were run on MuJoCo environments as well as and a single *Multi-Goal Robotics* [88] environment, provided by the *OpenAI Gym* [22, 13] library. This library

Environment	Type	State Dimensions	Action Dimensions
InvertedDoublePendulum-v2	MuJoCo	11	1
Hopper-v2	MuJoCo	11	3
Walker2d-v2	MuJoCo	17	6
FetchReach-v1	Robotics	16	4

TABLE 4.1: Different state and action spaces of the environments.

remains a vital contribution in the RL field, as it offers a pure *Python* library, providing complete environments, including the simulation of physics, observations, reward, and terminal states. In supervised learning, large datasets like *ImageNet* [58] provide a sophisticated benchmark. The RL community was lacking such a standardized baseline for a long time. The integrated MuJoCo environments are focused on accurate continuous control robotic simulation. In this thesis, experiments were conducted in four different environments, from which three had a two-dimensional state space, and one had a three-dimensional state space. Every environment comes with varying functions of reward and goals. These details are presented below, together with a table of the various state and action dimensions. Additionally, we display different snapshots, taken directly from the simulation.

InvertedDoublePendulum has the goal of balancing a double-joint pole on a cart. The reward is calculated [22] with an *alive bonus*, a penalty for each joint velocity (θ_1 and θ_2):

$$r(s,a) := 10 - 0.01x_{tip}^2 - (y_{tip} - 2)^2 - 10^{-3}(\theta_1)^2 - 5 \times 10^{-3}(\theta_2)^2, \tag{4.1}$$

where x_{tip} and y_{tip} are the tip coordinates of the pole. Although *InvertedDoublePendulum* describes a lower-dimensional action-space, compared to the other environments, it remains an arduous task for various continuous control algorithms.

Hopper is a *planar monopod robot with four-digit links* [22]. Its goal is to achieve a stable *hopping* behavior, moving the robot to the right. The reward function is given by:

$$r(s,a) := v_x - 0.005||a||_2^2 + 1, \tag{4.2}$$

where 1 is a bonus term for being *alive*, v_x is the velocity in x-direction, and a the acceleration. Hopper is especially interesting, as the environment has a difficult local optimum, which is a behavior that *dives* forward and then ends the trajectory by falling on the ground.

Walker is a *planar biped robot consisting of 7 links* [22] with two legs. The environment is similar to *Hopper*; however, it has a significantly larger degree of freedom and a more difficult behavior complexity as the agent has a higher tendency to fall. The reward is given by:

$$r(s,a) := v_x - 0.005||a||_2^2. \tag{4.3}$$

FetchReach represents a recently added 3-dimensional environment, which simulates a 7-DoF fetch robotics arm [88]. The goal is to reach a continuously changing target position (to be seen in red in Figure 4.2) with its *gripper*. Unlike any other task, the action space is four-dimensional, where three dimensions specify the desired arm position, and one dimension corresponds to the gripper movement. Even though this environment does not support *pick & place* tasks, the state and action spaces still contain the required picking dimensions. The reward, compared to other MuJoCo settings, is sparse, meaning the agent observes a reward of 0 if the gripper remains in the target position and -1 otherwise. The maximum trajectory length is limited to 50 samples. Even though the task is relatively easy to learn, the *sparse* reward objective makes it especially interesting to investigate.

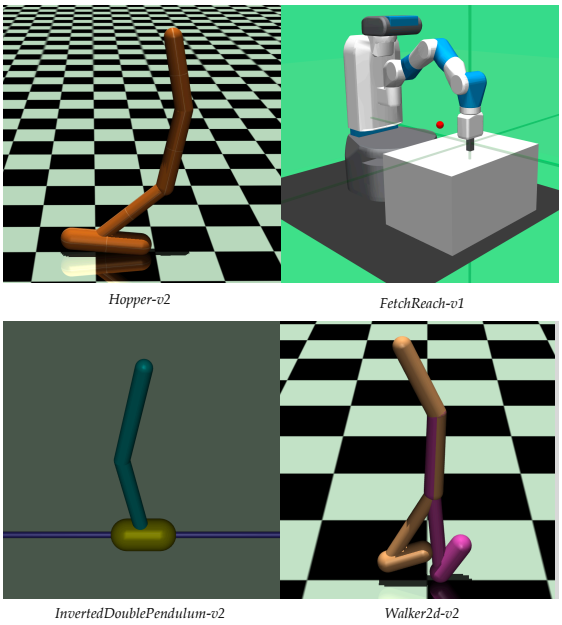


FIGURE 4.2: Different snapshots of the OpenAI environments. *Hopper-v2*, *Walker2d-v2* and *InvertedDoublePendulum-v2* are 2-dimensional tasks, whereas *FetchReach-v1* provides a 3-dimensional state and action space.

4.3 Experimental Evaluation

Given the recent concerns in reproducibility [46], experiments were run following the recommendations of Henderson et al. For each investigation, unless mentioned otherwise, six different seeded trials represent each evaluation. Unless otherwise specified, we used the default settings as disclosed in Appendix B. We then only varied the parameters of interest for each analysis. The modified parameters are elaborated further in the corresponding section or the caption below each experiment evaluation. For all graphs, each experiment line is represented by the aggregated *mean* across all trials for the y-axis observation and supported by a 95% *bootstrapped* confidence interval

[21] error band, calculated by 1000 bootstraps across different trials. *Bootstrapping* approximates the sampling error across the dataset by randomly drawing *samples* with replacements, such that many alternative permutations from the experimental data occur [21]. The variability across the generated datasets can then be used to compute the *bootstrap confidence interval* in order to attain a *significant* estimate of the sampling error.

Most hyper-parameter graphs represent different metrics (y-axis) for 1×10^6 timesteps (x-axis). Metrics were collected on each policy update $\pi \rightarrow \pi'$, smoothed with a symmetric *exponential moving average* (EMA) [50], and then resampled to an even grid represented by 512 data points. The original PPO implementation suggests using *fixed-length trajectory segments*, popularized [74] by Mnih et al., in order to generate a segment of equally (timestep) sized T trajectories in parallel, collected from multiple actors. An alternative implementation of the PPO family was chosen. In this alternative implementation, a single process (actor) receives a segment with numerous (different sized) trajectories S and then updates its gradient accordingly. Each process then represents a separate trial for each experiment, which was run in parallel. As a result, a policy gradient update represents a variable amount of timesteps, which would not allow a consistent confidence interval calculation. The EMA provides a reasonable estimate of the data trend by averaging over points near an observation. Such type of moving-average decreases randomness, offers a smooth trend line and allows resampling onto a consistent data grid for calculating the confidence intervals.

The graphs of the neural network experiments, usually represent different metrics averaged over all 1×10^6 timesteps by the neural network size. Contrary to the hyper-parameter tests, a resampling was not required, since we performed all experiments on a settled amount of 20 distinct sizes. Nevertheless, we computed the bootstrap confidence interval of all six trials. In some instances, the inscriptions of the neural network experiment charts refer to *final* metrics. In those cases, we measured the mean of the last ten policy updates.

In Section 3.2, we discussed various difficulties and solutions of policy gradient-based optimization. In order to underline the theory with practical experiments, an algorithm of the Proximal Policy Optimization family was implemented, supporting four different types of unconstrained policy losses: The vanilla PG objective L^{PG}, the adaptive KL penalized PPO objective L^{KLPEN} and the clipped KL objective L^{CLIP}. On top of that, the rewritten importance sampled version of the PG was added to show that both losses behave very similar, as they are mathematically equivalent. Note that all four implementations use an adaptive neural network learning rate, for which the Adam optimizer step size is adjusted based on the KL divergence of the new and old policy as proposed in [105]. Reminding the reader of the calculation of the adaptive penalty coefficient, for which $\omega = 2$ and $\nu = 1.5$ was used in all experiments:

$$\beta = \begin{cases} \frac{\beta}{\omega}, & \text{if } D_{KL} < \frac{D_{KL}^T}{\nu} \\ \omega\beta, & \text{if } D_{KL} > \nu D_{KL}^T, \end{cases} \tag{4.4}$$

we can then derive such an algorithm by using a learning rate coefficient $\varrho \in [0.1, 10]$ (initialized with $\varrho = 1$), which was multiplied with the initial Adam optimizer learning rate after the KL divergence was calculated between the new and the old policy.

Note that the parameters ω and ν are chosen similar to [105]. However, the algorithm turns out to be not very sensitive to them.

ϱ is then adjusted dynamically on every update of the penalty coefficient β, however, clipped in the range of $[0.1, 10]$, in accordance with:

$$\varrho = \begin{cases} clip(\frac{\varrho}{1.5}, 0.1, 10), & \text{if } \beta > 30 \\ clip(1.5\varrho, 0.1, 10), & \text{if } \beta < \frac{1}{30}. \end{cases} \tag{4.5}$$

Note that we use $clip(x, min, max)$ as a function which clips x inclusively within the interval $[min, max]$.

With this algorithm, we decrease the Adam step size with ϱ for $D_{KL} \gg D_{KL}^T$ (very high β) and increase it for $D_{KL} \ll D_{KL}^T$ (very low β). The interval of ϱ was additionally clipped, as unconstrained versions would sometimes lead to notably high KL divergence spikes, causing the run to converge to a non-optimal policy prematurely. We show in Section 4.6 that this method can significantly increase the stability and performance of PPO, even though indications to such an algorithm are only mentioned in footnotes and appendixes of prestigious PPO publications [105, 45].

Both, actor and critic network had three hidden layers and were optimized with different sets of parameters. The network width was dynamically adjusted based on a coefficient n_{size} following Table 4.2, such that a *pattern layer*, a *summation layer*, and a *decision layer* was attained in accordance with [72]. While small values for nn_{size} yielded a lower empirical performance, higher values have led to inconsistent results, prematurely converging to a local optimum. For more information, we refer to Section 4.6, in which the different architectures are invested accordingly. The activation function was *tanh* on all layers, except for the linear output layer.

Network	Hidden Layer	Size
Actor	1	$n_{obs} \times nn_{size}$
Actor	2	$\sqrt{n_{obs} n_{act}}$
Actor	3	$n_{act} \times nn_{size}$
Critic	1	$n_{obs} \times nn_{size}$
Critic	2	$\sqrt{n_{obs} \times nn_{size}}$
Critics	3	$max(\lfloor \frac{n_{obs}}{2} \rfloor, 5)$

TABLE 4.2: The size of the neural networks was adapted based on dimensions of the observation vector n_{obs} and the action vector a_{act} of each environment. For policy and value network, the size of the first hidden layer was set to $n_{obs} \times nn_{size}$, where nn_{size} is an empirically determined coefficient. The last hidden layer was dependent on the action space: $n_{act} \times nn_{size}$ (policy) and n_{obs}: $max(\lfloor \frac{n_{obs}}{2} \rfloor, 5)$. The second hidden layer was calculated based on the geometrical mean between the first and last layer.

4.4 Implementation

The selected results in Figure 4.3 and the supplementary material, show us that the implemented clipped version of PPO significantly outperforms the sample efficiency and final performance of conventional policy gradient algorithms in all environments. The algorithm matches the final performance of Schulman et al. for *InvertedDoublePendulum-v2* and *Hopper-v2* when compared in timesteps. However, sample efficiency and final performance for *Walker2d-v2* are lower compared to the original implementation. We will elaborate on the deviations further in Section 4.5 by investigating key differences and compromises, which had to be taken in hyper-parameters and algorithm details. Surprisingly, the implemented adaptive KL version of PPO performs relatively well compared to the clipped KL version on the selected environments. Schulman et al. showed that over a wider variety of environments, clipped PPO appears to perform

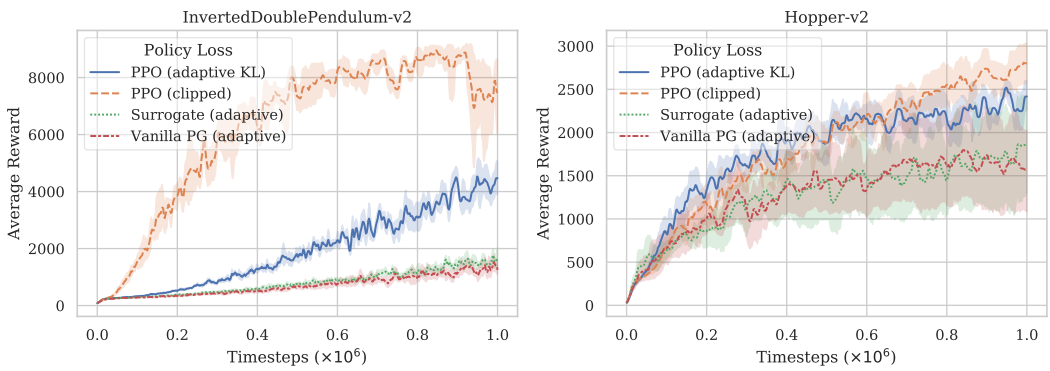


FIGURE 4.3: Selected benchmark results of several policy gradient losses.

generally better. However, different implementations show [45] well-performing solutions on very high dimensional environments for the adaptive KL PPO version as well.

4.5 Importance of Hyper-Parameters

In this section, we empirically investigate the magnitude of the effect different hyper-parameters have on the results of the implemented PPO algorithm. While hyper-parameter tuning is a well-known procedure amongst RL publications, we argue that the importance of different hyper-parameters is often not sufficiently emphasized, and the choice of optimal hyper-parameter configuration not consistent amongst literature. Parallel to this work, Henderson et al. showed [46] with the original repositories of different authors that small implementation details, which are often not reflected in the publications, can have a dramatic impact on the learning performance. This work contributes in that manner by first showing in-depth accentuated hyper-parameters (1) that have little effect on the performance of the algorithm; and then by underlining (2) poorly highlighted algorithm details, which can dramatically impact empirical results. Henderson et al. investigated parameters with a *top-down* approach by breaking down the existing repositories in a reverse engineering fashion. The investigation of this thesis represents a *bottom-up* approach, in which a PPO implementation was first pieced together based on publicly available baselines [102, 19] and publications [106, 104, 103, 105, 45] and then assessed hyper-parameters with minor and major influence.

4.5.1 Entropy Coefficient

Recent policy gradient optimization improvements started to emphasize adding the *entropy* H of the policy to the objective: $objective + x_2 H(\pi(\cdot|s))$, where H is the entropy and x_2 a coefficient. The idea is to support exploration by penalizing over-optimization on a small portion of the environment, to discourage premature convergence to sub-optimal deterministic policies [74, 105]. A policy entropy would, therefore, negatively reward actions that dominate too quickly and encourage diversity. In the experiments, the coefficient was set to 0.01, as commonly seen in the literature. No significant advantage in any of the considered environments could be observed. Conversely, the procedure did instead worsen the performance, as to bee seen in Figure 4.5. Also, experiments with significantly higher values for the entropy coefficient were run on the *Walker-v2* environment, which did cause the algorithm to collapse, as to be seen in Figure 4.4.

When looking into default parameters of publicly available baselines, one can observe that the coefficient x_2 seems only to be set $x_2 > 0$ for the Atari domain, whereas for other areas, such as the *MuJoCo* environments, the parameter seems to be set to 0, ignoring the policy entropy completely. Recent work [2] argues that entropy regularization is in direct correlation to the curvature of the objective and, therefore, requires adaption of the learning rate. On top of that, Ahmed et al. observed that entropy regularization does not seem to have any effect on specific environments such as *HalfCheetah*. Publications must elaborate on the usage of entropy regularization more clearly, specifically

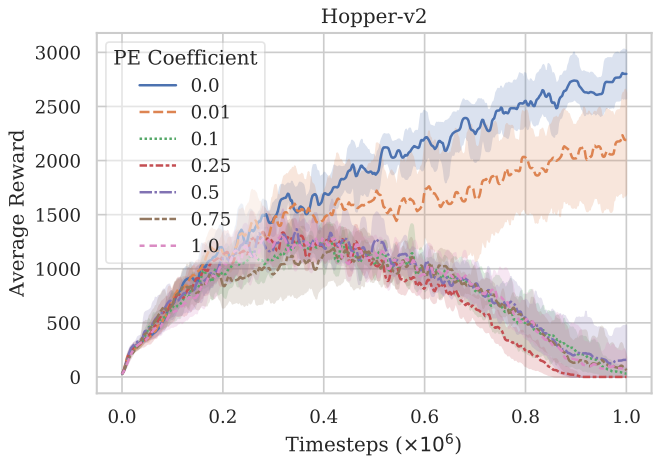


FIGURE 4.4: Different policy entropy coefficients evaluated on the *Hopper-v2* environment. Higher values for the entropy coefficient, without further adaption of other hyper-parameters, caused the algorithm to diverge to a reward of 0.

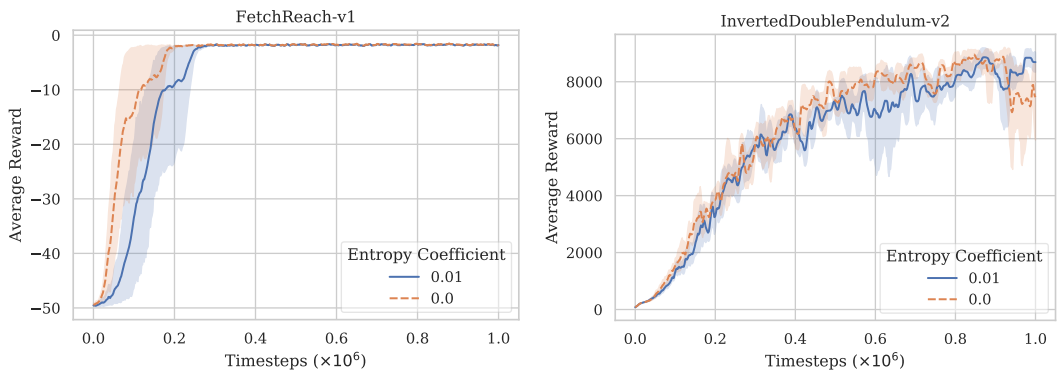


FIGURE 4.5: We only see little empirical evidence justifying the usage of entropy regularization for exploration purposes on the MuJoCo environments. An entropy coefficient of 0.0 equals no entropy regularization.

if the yielded performance of the hyper-parameter may be correlated with the learning rate or specific environment types.

4.5.2 Activation Function

In recent years, ReLU became popular amongst supervised deep learning approaches. ReLU has been shown [32] to outperform previous activation functions on deeper network structures. In Table 4.3, activation functions of different deep RL publications are concluded for the final fully connected layers. Since Rainbow, A2C, and ACER do not specify the utilized activation in their publications, publicly available baseline repositories were examined instead. Notice that algorithms focused on the *Atari* domain such as DQN, Rainbow, and A2C tend to use ReLU for fully connected layers, while tanh still

Algorithm	Activation Function
DQN [75]	ReLU
Rainbow [48]	ReLU
A2C [74]	ReLU + tanh
DDPG [68]	ReLU + tanh
TRPO [106]	tanh
PPO [105]	tanh
ACER [124]	tanh
ACKTR [127]	tanh + ELU [14]

TABLE 4.3: Different activation functions denoted in publications for fully connected layers.

plays a dominant role in more complex domains such as the MuJoCo environments, focused by DDPG, TRPO, PPO, ACER, and ACKTR. Henderson et al. investigated across algorithms and environments and found that activation functions can have a significant impact on the performance of the algorithm. In addition, they figured that a change in the architecture or activation function might also require tweaking in other hyper-parameters such as the trust region clipping or learning rate [46]. Experiments were performed on all four environments accordingly for ReLU and tanh activation functions. The weights of each tanh layer were initialized with the initialization technique [31] of Glorot and Bengio. The weights for ReLU activations were adjusted correspondingly to the initialization [44] of He et al. We will elaborate those method further in Section 4.5.3. Observing the selected results in Figure 4.6, we can see significant performance differences. Additionally, the average KL divergence between the new and old policy is plotted for the *Walker2d-v2* environment in Figure 4.6. We can see that the KL for tanh activations has less variance and tends to reduce, whereas the KL divergence of ReLU activations has high variance and steadily increases. For the *FetchReach* environment, no ReLU trial was able to learn a satisfactory policy[1]. A change of the activation function might, therefore, need more consideration with PPO, adapting different hyper-parameters of the algorithm to attain a sufficiently stable performance. Results of Henderson et al. confirm the stability issues of ReLU in combination with PPO by reporting a significantly lower mean reward when using ReLU instead of tanh.

4.5.3 Weight Initialization

Before 2006, no successful examples of deep multi-layer neural networks have been trained [31]. Weight initialization has been identified [31] as one of the key obstacles for the previously poor performance of standard gradient descent on deeper neural networks. In order to prevent the saturation of activation values in deeper hidden layers Glorot and Bengio proposed a *normalized initialization* of the weights W_{ij} with the

[1]Evaluations for all environments are provided in the supplementary material.

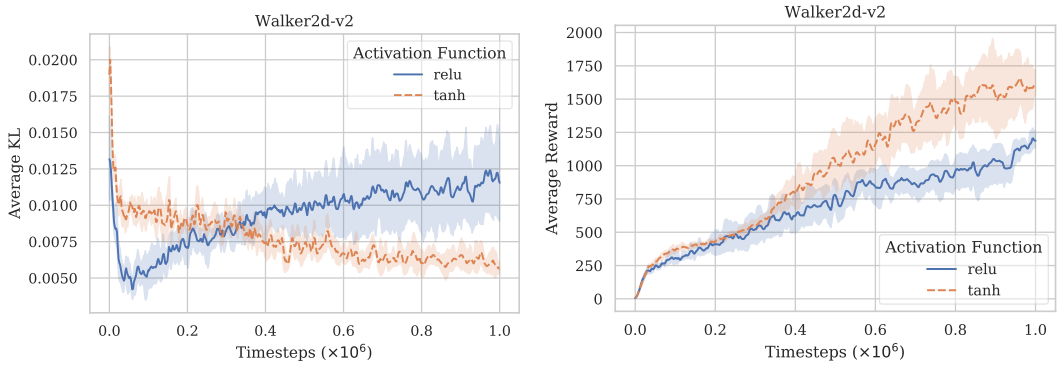


FIGURE 4.6: A comparison of ReLU and tanh activation functions on *Walker2d-v2*, which shows the mean KL divergence between the new and old policy on the left against the policy performance on the right. Notice how the performances of the ReLU experiment drops as soon as its mean KL divergence rises significantly around the 0.3×10^6 mark on the x-axis.

following normal distribution:

$$W_{ij} \sim \varphi\left(\mu = 0, \sigma^2 = \frac{2}{n_{in} + n_{out}}\right), \tag{4.6}$$

where n_{in} represents the number of input units, and n_{out} is the number of output units. Later, a similar method was proposed for *rectified activation functions*, where weights are initialized according to the following distribution:

$$W_{ij} \sim \varphi\left(\mu = 0, \sigma^2 = \frac{2}{n_{in}}\right), \tag{4.7}$$

To that end, we performed experiments to show that the correct weight initialization of deep neural networks can be crucial to learn competitive policies. Figure 4.7 shows considerable performance differences between a *standard* initialization and the technique proposed by Glorot and Bengio for the environments *Hopper* and *Walker2d*. Both experiments were optimized with tanh activations. For the *standard* initialization, initial weights W_{ij} for all layers were drawn from the following "commonly used heuristic" [31]:

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n_{in}}}, \frac{1}{\sqrt{n_{in}}}\right], \tag{4.8}$$

where n_{in} corresponds to the size of the previous layer und $U[-a, a]$ is a uniform distribution in the interval $(-a, a)$. Note that two neural networks with only three hidden layers were evaluated. Hence, even higher importance of correct initialization can be expected for deeper neural networks, when, e.g., using CNNs to represent state directly from raw sensor input.

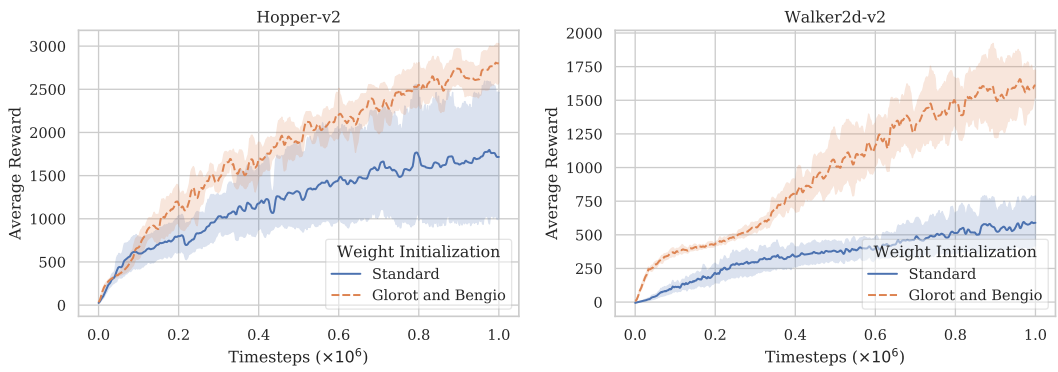


FIGURE 4.7: A comparison between a *standard* weight initialization and the weight initialization. Activations of deeper layers saturate [31], which slows down learning significantly.

4.5.4 Segment Size

In Section 3.2.6 of the theoretical analysis, we were concerned with *fixed-length trajectory segments* [74, 105], which use a truncated version of the advantage estimator. We identifiy the *segment size* of the algorithm as one of the key differences to the original algorithm of Schulman et al. and thus empirically investigate the horizon T and the segment size S of the family of *Proximal Policy Optimization* algorithms. Two different segment sizes were implemented:

- A dynamic segment size (1) based on the parameterized amount of collected trajectories S, so that in each policy update the gradient is approximated by a total amount of $T_{total} = \sum_{s=1}^{S} T_{\tau_s}$ timesteps, where $T_{\tau_s} = |\tau_s|$ is the dynamic length of the s-th trajectory $\tau_s \in \{\tau_1, \tau_2, ..., \tau_S\}$. Each trajectory ends with a terminal state, such that we estimate advantage with the classical GAE:

$$\begin{aligned} \hat{A}_t^{GAE(\gamma,\lambda)} &= \sum_{l=0}^{T} (\lambda\gamma)^l \delta_{t+l}, \\ \textit{where } \delta_t &= r_t + \gamma V(s_{t+1}) - V(s_t). \end{aligned} \tag{4.9}$$

- A fixed segment size (2), equal to the one defined by Schulman et al., where $T_{total} = H$, so that each trajectory either ends with a terminal state or the trajectory is truncated and the expected remaining reward estimated with the value function in accordance with a truncated version of the Generalized Advantage Estimation [105]:

$$\hat{A}_t = \delta_t + (\lambda\gamma)\delta_{t+1} + ... + (\lambda\gamma)^{T-t+1}\delta_{T-1}. \tag{4.10}$$

Figure 4.8 compares the segment size of 2000 policy updates for two dynamic and one fixed segment size solution. T_{total} has been found to differ considerably depending on the environment when using a segment size limited by trajectories. This discrepancy occurs due to the difference in maximum trajectory length. One must, therefore, be more careful in adjusting S when using dynamic-length trajectory sizes. Consider, for

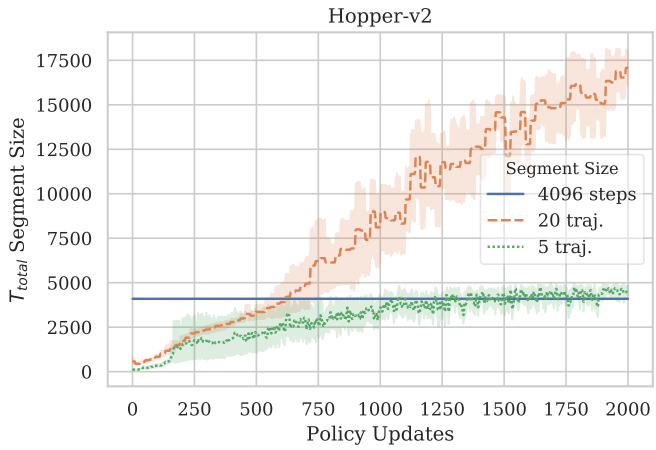


FIGURE 4.8: Different segment size limitations evaluated on the *Hopper-v2* environment for 2000 policy updates. The y-axis shows the total amount of timesteps T_{total} of each segment. T_{total} represents the total amount of states in a segment collected before each gradient update of the policy network.

example, the environment *FetchReach-v1*, which has a fixed trajectory length of 50. For $S = 5$, this yields into $5 \times 50 = 250$ states per segment, which turned out to be insufficient, as visible in Figure 4.9. Further experiments of the supplementary material show that lower dynamic segment sizes are more *sample efficient*, while larger dynamic segment sizes tend to have lower *variance* and are more consistent in between trials.

On top of different dynamic sized segment sizes, we also conducted experiments for different fixed-size segment sizes. Figure 4.10 shows in several cases that in the here presented algorithm, lower values for H resulted in convergence to a premature sub-optimal solution, while a higher H turned out to be less sample efficient; however, trials learned more stably.

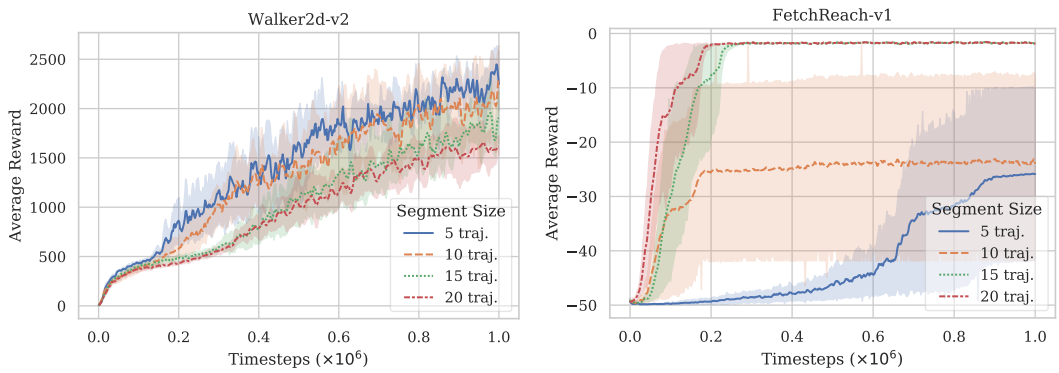


FIGURE 4.9: Selected results of the dynamic horizon solution, where $S \in \{5, 10, 15, 20\}$. We can see considerable differences in learning speed and final performance.

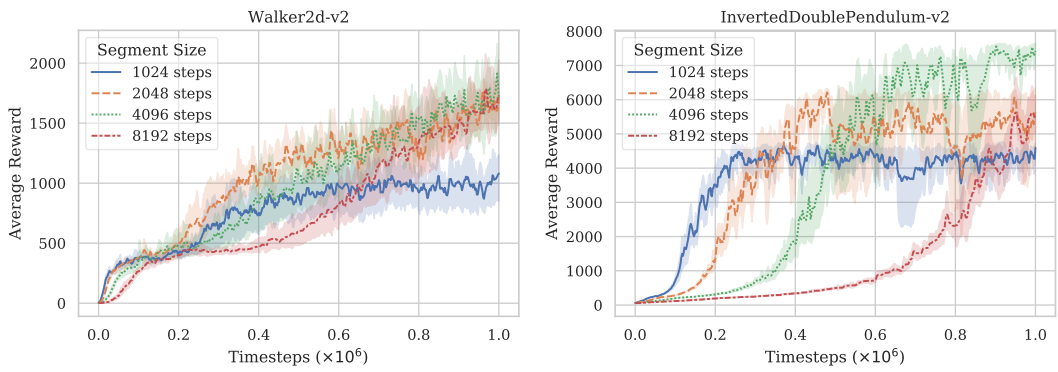


FIGURE 4.10: Selected results of the fixed horizon solution, where $H \in \{1024, 2048, 4096, 8192\}$. We can see that, depending on the environment, lower fixed segment sizes prematurely converge to a local optimum.

When comparing fixed with dynamic segment sizes, notable differences in final policy performance of the two solutions can be found. One may interpret the value function approximation to be the cause, as the prediction of the remaining advantage of truncated trajectories is, other than the batch-size, the only notable difference between the two implementations.

Measuring the performance of a non-linear approximation is not as trivial as in, e.g., linear function approximation. In *linear modeling*, one may use the well-known *coefficient of determination* R^2 [101] for a reasonable first assertion on the model performance. However, R^2 requires the *regressors* (the input variables of the regression) to be independent and is calculated based on the underlying squared error of a presumed *linear* functional relationship. In the present case, our value function represents a non-linear approximation, and we also may not assume independence in all dimensions of the state-space. Hence, the use of the "proportion of variance explained" (VE) [101] can be considered instead:

$$VE = 1 - \frac{Var(V^{\gamma} - V_{\theta}(s))}{Var(V^{\gamma})}, \tag{4.11}$$

to quantify, the respective sample variance, measured around their respective sample means [101]. The VE intuitively indicates how well the value function model $V_{\theta}(s)$ predicts the baseline V^{γ} as a function.

> Similar to R^2 one may interpret values for $VE \to 1$ as good predictions. Keep in mind that the VE may not be construed as a measure for good *generalization* [129], which is commonly validated in *supervised learning* with a separate *test error* on an independent test data set.

Figure 4.11 shows VE calculated before fitting and the *mean squared error* $\mathbb{E}_t[(V^{\gamma} - V_{\theta}(s))^2]$ calculated after fitting. We can observe considerable better baseline predictions for the dynamic step size solutions, causing a performance difference in comparison to the fixed step size version. A more detailed performance comparison of best performing fixed and adaptive solutions of all environments is contained in the supplementary material.

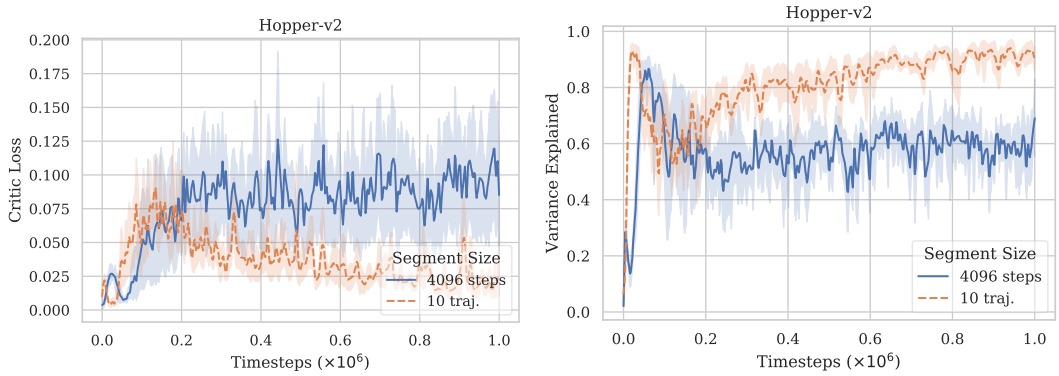


FIGURE 4.11: A comparison of the *mean squared error* and the *proportion of variance explained* of the critic network, which shows the best performing fixed segment solution (4096 steps) against the best performing dynamic segment solution (10 trajectories). Notice how the fixed solution consistently suffers from inaccurate value function predictions.

In this section, we investigated dynamic-size horizons. We find that many publications [105, 106, 74], which use fixed-length horizons, perform the majority of their theoretical research on the assumption of complete trajectories, then, however, move on to a truncated trajectory horizon in their practical implementation. With this investigation, we saw that there are notable differences between the two approaches, which we find, are often insufficiently emphasized in the literature. The experiments demonstrate that fixed-horizon solutions have a firm reliance on the value function prediction quality. A change from a dynamic-sized horizon to a truncated fixed-sized horizon may, consequently, require further adaptions in the architecture of the value network or the GAE parameterization. Furthermore, it has been shown that a dynamic-sized horizon has a strong dependence on the maximum trajectory length of the environment. Hence, the selection of the hyper-parameter S may require additional care, in particular, if the algorithm is evaluated on multiple environments with the same hyper-parameter configuration.

4.6 Neural Network Size

This section contributes to current RL research by investigating the neural network size with the implemented PPO algorithm. First, in Section 4.6.1, we review the neural network architectures of related literature, where we focus on difficulties in interpretability and reproducibility. Second, Section 4.6.2 describes the experimental setup and elaborates on the research approach. Then, we experiment with the interaction of network size and policy change in Section 4.6.3 and the interaction between network size and performance in Section 4.6.4.

4.6.1 Architectures of the Literature

The term hyper-parameter extends by definition from simple tunable parameters as, e.g., the *horizon*, up to much more influential parameters like the architecture of the neural network. While RL was previously limited to manageable small MLPs [94, 92], recent advances in supervised deep learning and RL made it possible to extend the RL domain from toy problems to much more complicated challenges such as robotics and learning from raw sensor-state. It can be said that even though most recent successes in RL apply deep neural networks, the size and shape of the neural network architecture have not received enough research attention in the context of RL. Architectures of recent deep RL publications can differ considerably, as concluded in Table 4.4, and require careful fine-tuning [46] w.r.t. other hyper-parameters such as, e.g., the activation function. Despite the substantial influence of the architecture, it can be said that many publications do not justify their chosen architecture details sufficiently.

Algorithm	Architecture	
	Atari	MuJoCo
DQN [75]	CNN + (256)	-
Rainbow [48]	CNN + (512)	-
A2C [74]	CNN + (256) + LSTM [49]	1 x (256) + LSTM
DDPG [68]	CNN + (200x200)	1 x (400x300)
TRPO [106]	CNN + (20)	2 x (30/50)
PPO [105]	CNN + (256) + LSTM	2 x (64x64)
ACER [124]	CNN + (512)	1 x (512)
ACKTR [127]	CNN + (256)	2 x (64x64)

TABLE 4.4: Different neural network architectures of influential deep RL contributions interpreted from their original publications. For MuJoCo environments, it is additionally indicated, if two separate networks have been used for the value and policy representation or a *double-headed* single network was applied.

First, we argue that it is often not clearly stated, which network architectures are used for value and policy networks under which circumstances. In many of the reviewed publications, the utilized neural network architecture was only quoted instead of consistently reported. A noticeable example is the asynchronous architecture [74] of Mnih et al., which is often referenced, however, then, utilized with a different network size or activation function without further reasoning. These avoidable inconsistencies impede a conclusive interpretation of results in deep reinforcement learning research.
Second, benchmarks against other publications are often compared with a modified neural network structure that differs considerably from its original design. Such types of comparisons make the interpretability and comparability of benchmark results less transparent.
Third, the reviewed publications often only report architectures for specific environment types but then modify them for different kinds of environments. While this seems

reasonable for particularly disparate environments like Atari and MuJoCo, it is also conspicuously often applied for high-dimensional tasks such as the *Humanoid* [22, 13] environments, without further reasoning.

4.6.2 Experiment Setup

Neural network architecture optimization strives with a more substantial degree of freedom in contrast to other hyper-parameters. The optimization complexity can be assessed as a composition of four adjustable factors: *network depth*, *network width*, the *activation function*, as well as the *learning rate*. Prior research has suggested that there may be more influential factors, such as the SGD mini-batch size [112]. However, these factors have not been included in the research scope of this work. Instead, we focus on experiments that investigate the *network width* and *learning rate*. To that end, this section describes an attempt to design a flexible neural network architecture that scales with respect to both of these dimensions.

Dynamic Step Size

In order to investigate the dependence of learning rate and neural network size, we applied an *adaptive* Adam step size approach. In supervised learning, similar adaptive concepts exist. *RPROB* [93] adapts the learning rate locally based on information about the local gradient. *Adam* [55] estimates *lower-order moments* to adjust learning rate scales for each layer independently. However, none of these methods were designed to leverage the internal representations of the RL algorithm, such as metrics we have about the change in policy. One approach to use these metrics has been intimated in [105] for the vanilla policy gradient. In this method, the Adam step size is adjusted based on the KL divergence between the old and the new policy with a similar procedure to Equation 3.39 of Section 3.2.6. We will observe that this algorithm—which is only mentioned in a footnote—can considerably increase algorithm stability. Concurrently we will show that this algorithm performs independently of the learning rate on an extensive range of different model sizes without requiring further adjustment of the step size. Consequently, this will simplify previously required expensive fine-tuning of both dimensions and, thus, allow a much more flexible adjustment of the network size to the problem complexity.

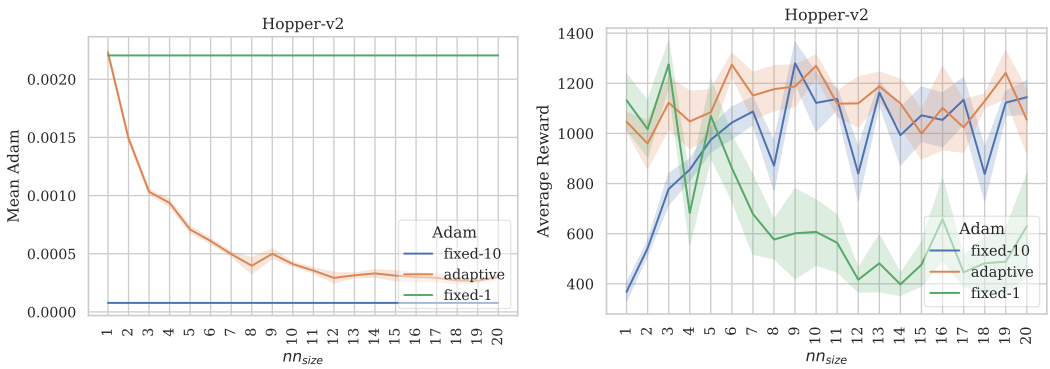
Environment

The *Hopper-v2* environment was selected for all experiments. *Hopper-v2* contains, in contrast to, e.g., *InvertedDouble-Pendulum-v2* or *FetchReach-v1*, an unusual local minimum around the reward of 1000. Even though this environment property increased the variance of the results, it also allowed the exploration of stability and, consequently, led to much more meaningful results.

Evaluations

Due to the results of Section 4.5.2, the *tanh* activation function was selected for the neural network experiments. Additionally, the depth of the neural networks was limited to three hidden layers, suggesting a field of subsequent research. In order to examine

the step size as a factor of network width, different values for $nn_{size} \in \{1, ..., 20\}$ were evaluated. Three types of evaluations were considered for neural network experiments.



(A) Mean Adam (B) Mean Reward

FIGURE 4.12: The Adam step size (left) of the policy network and the reward (right) averaged over all policy updates. The optimal (adaptive) learning rate was on average ×15 higher for small networks in comparison to large networks.

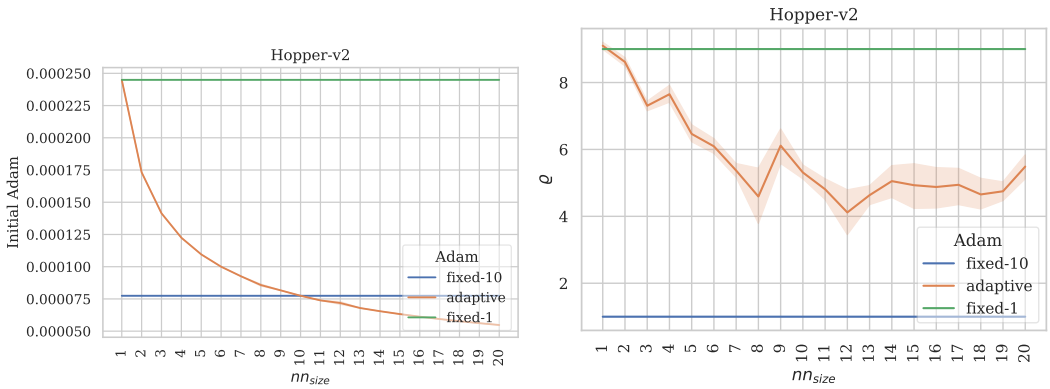
In an **adaptive** evaluation, the already discussed dynamic adjustment of the step size was used. This would allow the algorithm to adjust the learning rate dynamically within the interval $lr \in [lr_{initial} \times 0.1, lr_{initial} \times 10]$ with the learning rate coefficient $\varrho \in [0.1, 10]$, based on the observed difference between the new and the old policy. On top of that, the initial learning rate for value and policy network was biased by:

$$lr_{initial}^{policy} = \frac{9 \times 10^{-4}}{1.5\sqrt{h_2^{policy}}}, \; lr_{initial}^{value} = \frac{0.0108}{1.5\sqrt{h_2^{value}}}, \tag{4.12}$$

where h_2 is the size of each second hidden layer for each network capacity. The constants of Equation 4.12 were found heuristically. This bias will intuitively increase the initial learning rate for smaller network sizes and decrease it for larger network sizes; however, it still allows the algorithm to adjust its learning rate when applied with the adaptive approach. We show the adjustments of the initial step size and the step size coefficient in Figure 4.12.

In a fixed evaluation, which we refer to as **fixed-10**, the initial step size of the adaptive solution for $nn_{size} = 10$ was used for all network sizes. The step size coefficient was set to 1, and the learning rates were not adjusted dynamically and instead kept frozen during the course of all evaluations.

The third evaluation, which we refer to as **fixed-1**, was fixed as well. However, in this case, the initial learning rate of $nn_{size} = 1$ was used with a fixed step size coefficient of $\varrho = 9$, which turned out to be the optimal (adaptive) learning rate for $nn_{size} = 1$, as we can read from Figure 4.12. Also, here, the step size coefficient and the learning rate was unchanged during the course of all evaluations.



(A) Initial Adam

(B) Step size coefficient

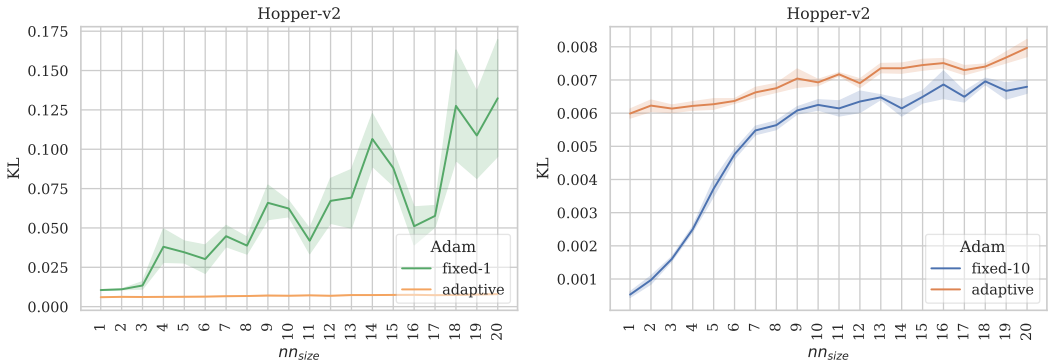
FIGURE 4.13: The adaptive Adam optimizer step size was calculated by multiplying the initial learning rate (left) with the step size coefficient (right) $\varrho \times lr_{initial}$. The result is plotted in Figure 4.12, which we then directly used for the optimization. The step size coefficient may be seen as the correction factor of the algorithm for our initial guess of the correlation between step size and model size.

4.6.3 Neural Network Size and Step Size

Recent policy gradient advances such as PPO concentrate on limiting the step size in the direction of the steepest direction of the gradient. Based on the empirical results here presented, we argue that the size of the neural network has a direct influence on the parameter optimization landscape of the optimized objective function. Consequently, the model size also influences the change in policy on every optimization step of the gradient.

Figure 4.12 shows us that the algorithm employed up to $\times 15$ larger step sizes for small networks, in contrast to larger ones. This effect was surprising, as the learning rate was already biased with higher initial values for smaller network sizes. The algorithm additionally increased this bias with a step size coefficient of $\varrho \sim 9$ for $nn_{size} = 1$ and then continuously decreased it for more extensive models up until a factor of $\varrho \sim 5$. We can observe this effect in Figure 4.13. The learning rate was, hence, initially set too low; however, the algorithm then corrected the factor by itself.

To understand the correlation between the optimal learning rate and the model size more in-depth, we compare both fixed solutions with the adaptive solution. Consider the multivariate space mean KL divergence between the new and old policy $D_{KL}(\pi(a|s,\theta)||\pi(a|s,\theta'))$ over the whole learning process in Figure 4.14. By interpreting this figure, we can confirm the increase in ϱ for small network sizes, as the average change in policy seems to become negligibly little, with a decrease in model size. Inversely, by increasing the model capacity without adjusting the learning rate, we can observe a substantial increase in the policy change. A shift in the model size, consequently, directly affects the transformation of the policy and, thus, requires the recalibration of the learning rate. These results confirm the suspicions in [46] of Henderson



(A) KL Divergence (B) KL Divergence

FIGURE 4.14: The KL divergence averaged over all policy updates of the fixed solutions *fixed-1* (left) and *fixed-10* (right) against the adaptive solution. An inappropriately low or high KL divergence led to suboptimal learning behavior, as visible in Figure 4.12. At $nn_{size} = 20$, the mean KL divergence of *fixed-1* was around $\times 163$ higher than optimal (adaptive).

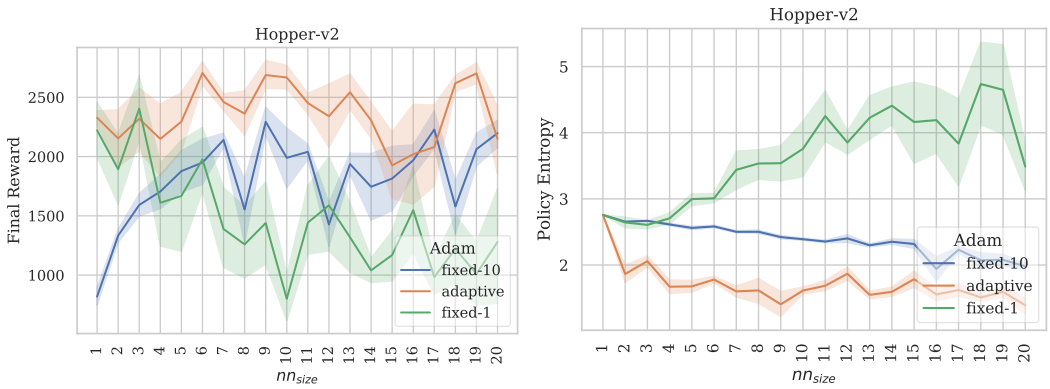
et al., which also reported substantial differences in the KL divergence for larger neural networks. Based on the results of this work, we argue that a change of the network width also requires a shift in the learning rate, when using a KL constraint policy-based optimization algorithm. One may advocate this position, as the results show a considerable influence of the chosen step size on the change in policy.

Going further, we draw the initial suggestion to use higher learning rates with smaller networks and to apply lower learning rates with more extensive networks when optimizing stochastic policies with an algorithm such as PPO. It can additionally be indicated that a change in the network architecture may also require some further adjustments in the KL constraint. While not elaborating this further, we assess that an accurate statement in this regard may require further effort, studying different shapes, types, and depths of architectures as well as a wider variety of policy gradient losses.

4.6.4 Neural Network Size and Policy Performance

To understand the influence of neural network size on reward, we measured the average return (learning speed) and the final return (learning performance) in Figures 4.12 and 4.15 of all three types of evaluations. The final performance was measured by taking the average of the last 10 policy updates. Notice that the results show different variance for each neural network size, which comes down to the fact that unstable solutions prematurely converge to the local optima of the environment. One can, therefore, interpret the error-bars of Figure 4.15 as a measure of algorithm stability.

Overall, we can see how crucial the correct selection of the learning rate is. For an insufficiently low selected step size, we observe a significant decrease in learning speed and final performance but find excellent algorithm stability. For an inappropriately high chosen step size, we perceive unequivocal stability issues and similarly, declining



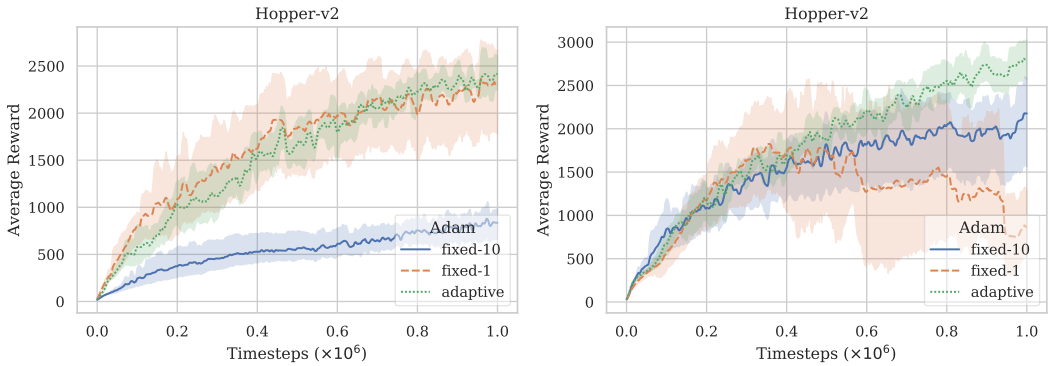
(A) Final Mean Reward (B) Policy Entropy

FIGURE 4.15: The final mean reward (left) against the final policy entropy (right). Both graphs represent the average taken over the last ten policy updates. An insufficiently large step size does not affect the entropy of the policy, which we can read for *fixed-10* in the interval $n_{sizes} \in \{1, ..., 7\}$.

speed and performance. The comparison in Figure 4.16 confirms these stability issues. Additionally, the final policy entropy of all solutions is shown in Figure 4.15. We find no effect on the entropy for learning rates chosen too little; however, a large impact when using a step size that is too large. One may speculate that this effect indicates a supervised learning problem of underfitting for insufficient step sizes and a KL constraint violation for overly extensive learning rates. It is obscure why this kind of KL outburst occurs, given that the PPO objective should clip excessive policy updates.

Considering the assumption that the adaptive solution corresponds to the optimal learning rate, it is remarkable how little the difference in final performance between a small and an extensive neural network is. For the three-layered neural networks, we observe the best stability and final return for around $nn_{size} \in \{6, ..., 11\}$. Further experiments may be required with more complex environments, investigating a broader range of different model shapes and sizes may be necessary to understand this effect more thoroughly.

It has been challenging to comprehend the complex correlation between network width and performance fully. Take, for example, $nn_{size} = 8$ in comparison to $nn_{size} = 9$ of the fixed solutions. Even though the network size is very similar, the measured reward would perform consistently worse for $nn_{size} = 8$, no matter how often both experiments were repeated. Related research [10, 30, 67, 98] has suggested that a complexity measure of a deep learning model may be more involved than a simple count of model parameters. While various new measurements have been proposed [10, 30, 67], one interpretation is that the smoothness of the high-dimensional objective landscape is likewise critical. The idea is that a smoother parameter optimization landscape allows more predictive and stable behavior of the gradients, and, thus, accelerates the training by enabling "a larger range of learning rates" [98]. Given this assumption, we



(A) $nn_{size} = 1$ (B) $nn_{size} = 10$

FIGURE 4.16: A comparison between a small neural network (left) and a large neural network (right) for the adaptive and fixed solution over all timesteps. Notice the shift in performance of *fixed-10* and stability for *fixed-1*. Small neural networks were able to learn a surprisingly good policy on the *Hopper-v2* task when using the correct (adaptive) learning rate.

can interpret the change of $8 \rightarrow 9$ in nn_{size} as more versatile than a simple change in the number of parameters. A network of the size $nn_{size} = 9$ might, consequently, be more receptive to a broader range of learning rates, causing the differences in performance. Certainly, more research will be required to understand this effect more thoroughly.

A reader should keep in mind that these RL polices were trained and evaluated in the same environment. Consequently, the measured policy performance does not testify any amount of *generalization* ability. To this end, we refer the reader to related research [52, 91, 82, 15, 86].

4.7 Summary

This practically orientated chapter contributes to current RL research with two types of empirical evaluations. It investigated the magnitude of the effect of hyper-parameters and the influence of neural network size and step size on policy performance. The chapter first described the *bottom-up* approach and how a representable algorithm of the PPO family has been implemented. Then, an indication of the experimental framework, the utilized environments, and the form of evaluation and implementation was given. Subsequently, we looked at the results of hyper-parameter experiments in Section 4.5 as well as the results of neural network size experiments in Section 4.6. In this final summary, we are concerned with the practical contributions of this work, where we briefly discuss the conclusions of experiments and derive some general recommendations.

What recommendations can be drawn from hyper-parameter experiments?

The hyper-parameter experiments show that little changes in often negligibly reported

hyper-parameters can have significant differences in the final performance of the algorithm. For future deep RL research, it is, consequently, advisable to explain or reference all utilized techniques sufficiently, such as, e.g., the entropy coefficient and to justify their use on specific environment types. Specifically, the architecture of the neural network, which includes the utilized activation function, optimizer, or weight initialization, must be more consistently reported. When comparing the performance of different types of algorithms, the originally quoted neural network architecture should be used, since the algorithm performance can differ considerably with only little discrepancies in the neural network architecture [46, cf.]. The use of the original architecture will avoid ambiguous and misleading results and provide better transparency for future research. If hyper-parameters, algorithm details, or the experimental setup will continue to negligibly reported, the future development speed of deep RL algorithms will suffer from the arising *technical depth*, caused by the missing reproducibility and transparency of the developed algorithms.

What recommendations can be drawn from neural network experiments?

The experiments, performed on different neural network sizes, indicate that the size of the neural network directly influences the parameter optimization landscape of the objective function. A change in the network architecture may, therefore, also require a difference in the learning rate. The performance of different fixed learning rates and an adaptive learning rate was compared against different neural network sizes. The results indicate that smaller neural networks require higher learning rates, whereas more extensive neural networks require lower learning rates to achieve optimal performance. Subsequently, if the learning rate is chosen too little, the change in policy diminishes, and the performance increases are impractical. Inversely, step sizes selected too high cause the policy performance—independently of the KL constraint—to collapse significantly. Given an optimal step size, we can only attribute little differences in performance and stability to the model size. A more in-depth exploration of a broader range of model sizes and more complex environments may be required.

Chapter 5

CONCLUSION

This thesis antagonized the arising obscureness of recently published deep reinforcement learning research. While the exploitation of deep learning in the field of reinforcement learning introduces new challenges, it also opens up new frontiers in a wider variety of new domains. This work showed that the solution to these challenges is not always straightforward and introduces further complexity.

In order to handle this emerging overhead, we have seen suitable approaches in the field of value function- and policy gradient-based reinforcement learning. In the value function-based domain, we looked at the off-policy algorithm DQN and discussed various improvements in Chapter 2. In the policy gradient-based field, we focused on on-policy algorithms in Chapter 3, which optimize their policy with a modified version of gradient descent. Interestingly, both chapters introduced fairly different approaches to tackle the challenges of deep neural networks. On the one hand, this may be attributed to the difference in the form of the optimization objective. On the other hand, algorithms like PPO are purely on-policy based and, consequently, do not employ off-policy techniques, such as experience replay. Parallel to this thesis, there is ongoing research [28, 37], concerned with the integration of different value function-based improvements in an off-policy actor-critic setting.

Chapter 4 was dedicated to the practical research objectives of this work. First, we were concerned with the effect and importance of algorithm details and hyper-parameters. Based on an investigation of the literature, we showed how crucial the exact documentation and disclosure of little algorithm details are. We investigated hyper-parameter contradictions and made an attempt to explore them empirically. This investigation put forth important details of modern algorithms to promote better clarity and transparency for future reinforcement learning research. The second part of Chapter 4 makes the point that the deep neural network yet remains a poorly understood collection of hyper-parameters in reinforcement learning. It reviewed the disparities of recent publications and determined that we need further research efforts regarding the architecture of deep neural networks in the domain of reinforcement learning. To that end, an initial empirical effort has been made to investigate the influence of network size and step size on policy performance. It was shown that a shift in the size of the neural network directly influences the change in policy and therefore requires an adjustment in the step size.

OUTLOOK

This thesis suggested several opportunities to conduct future research. It especially showed how complicated the latest novel deep reinforcement learning methods have become. One route of future research concerning this complexity remains the worksite of value-based deep reinforcement learning. Without a proper theoretical understanding of the newly proposed improvements in this domain, the integration of such into an off-policy gradient algorithm—which is what research is currently working [28, 37] on—introduces new, not understood intricacies. A focus of research, which is less oriented on empirical success and returns to more critical theoretical properties [128, 41], will avoid [47] a similar hype-cycle we had [89] with the proposal of TD-Gammon. Furthermore, we require a shift in focus on simplicity [1] and better software architecture, so we keep the technical depth of deep reinforcement learning to a minimum and develop algorithms less sensitive to hyper-parameters [37, 22, 46] and hidden details.

In regards to the issues of reproducibility and hyper-parameter awareness, one possible line of future research is to develop a reliable hyper-parameter agnostic approach [46]. Such a framework would allow the fair comparison between novel algorithms on a standard benchmark and ensure that there are no external inconsistencies introduced. One achievement in the matter remains to be the OpenAI baseline repository [19]. In this publicly available repository, the algorithms are reviewed in an open-source process, integrated into a standard-framework, and evaluated with fair benchmarking [22]. The intricacy of new emerged RL algorithms crucially requires the release of the full codebase and the learning curves to ensure fairness and reproducibility.

Appendix A

MATHEMATICAL APPENDIX

This appendix serves the reader as a mathematical encyclopedia. It covers several principles of *Information Theory* [108], such as the *entropy*, the *cross-entropy*, and the KL divergence. Furthermore, this appendix discusses *multivariate probability* theories, such as the *likelihood function*, the *score*, and the *Fisher information* with its relation to the *square matrix of second-order partial derivatives* (the Hessian). Finally, this appendix defines the *Hadamard product* as well as an indication of how the L^2 *projection* is performed.

Definition 1. (Entropy). The *entropy* [108], typically denoted as $H(p)$, measures the minimum average required lossless encoding size of a data distribution. Given a discrete random variable x and a probability mass function $p(x)$, the entropy is defined [108] as:

$$H(p) = -\sum_{i=1}^{n} p(x_i) log p(x_i), \tag{A.1}$$

whereas, for continuous variables, the term can be rewritten as an integral such that

$$H(p) = -\int p(x) log p(x) dx, \tag{A.2}$$

which can be equally used with the expectation form:

$$H(p) = \mathbb{E}_{x \sim p}[-log p(x)]. \tag{A.3}$$

Note that $x \sim p$ means that we are sampling x values from $p(x)$ to calculate the expectation.

The entropy is usually interpreted as uncertainty or unpredictability. If the entropy is high, $p(x)$ is more or less evenly distributed with small probabilities. Hence we are not certain about the encoding sizes of the individual values. For many practical problems, we may not know the true probability distribution. However, the true distribution can be approximated using an estimating probability distribution q. The quality of q can then be measured with the following definition.

Definition 2. (Cross-entropy). Given a true probability distribution p, which is estimated by an estimating probability distribution q, the *cross-entropy* $H(p, q)$ [71] defines the expected encoding size for each drawn sample, if the encoding size is calculated

with q but the expectation taken with p.

$$\begin{aligned} &H(p,q) = \mathbb{E}_{x\sim p}[-logq(x)] \\ &where\ H(p) \leq H(p,q), \end{aligned} \tag{A.4}$$

This measure essentially cross-checks the theoretical minimum encoding size with the actual true encoding. Note that $H(p,q) \neq H(q,p)$, except that $q = p$, so that $H(p,q) = H(p,p) = H(p)$, which would mean that q already represents a perfect estimate of p.

Definition 3. (Kullback-Leibler divergence). The *Kullback-Leibler (KL) divergence* [51, 59] measures the difference between two probability distributions. We can, therefore, express the difference between $H(p,q)$ and $H(p)$ with the KL divergence like:

$$\begin{aligned} D_{KL}(p \mid\mid q) &= H(p,q) - H(p) \\ &= \mathbb{E}_{x\sim p}[-logq(x)] - \mathbb{E}_{x\sim p}[-logp(x)] \\ &= \mathbb{E}_{x\sim p}[-logq(x) - (-logp(x))] \\ &= \mathbb{E}_{x\sim p}[logp(x) - logq(x)] \\ &= \mathbb{E}_{x\sim p}\left[log\frac{p(x)}{q(x)}\right]. \end{aligned} \tag{A.5}$$

With this equation, we can express the difference between two stochastic policies, as they are equivalently defined as conditional probability distributions. Let $\pi(a|s)$ be a stochastic policy and $\widetilde{\pi}(a|s)$ be a different stochastic policy. Then, we can express the KL divergence between those two policies as $D_{KL}(\pi \mid\mid \widetilde{\pi})$. In order to calculate the KL divergence for multivariate normal distributions with a mean μ, a covariance matrix C and equal dimensions n, we can replace the distributions $p(x)$ and $q(x)$ with their multivariate density function [23] and obtain:

$$\begin{aligned} D_{KL}(p \mid\mid q) &= \mathbb{E}_{x\sim p}[logp(x) - logq(x)] \\ &= \frac{1}{2}\left(log\frac{detC_q}{detC_p} - n + tr(C_q^{-1}C_p) + (\mu_q - \mu_p)^T C_q^{-1}(\mu_q - \mu_p)\right). \end{aligned} \tag{A.6}$$

Note that $tr(\cdot)$ defines the trace of a $n \times n$ matrix M, such that $tr(M) = \sum_{i=1}^{n} m_{ii} = m_{11} + m_{22} + ... + m_{nn}$, which is the sum of the complex eigenvalues of M.

Definition 4. (Likelihood Function). Let a random vector $X^n = (X_1, X_2, ..., X_n)$ have a joint density $p(x^n|\theta) = p(x_1, x_2, ..., x_n|\theta)$, where $\theta \in \Theta$. Then, we can define the *likelihood function* $L(\theta) : \Theta \to [0, \infty)$ [71] as

$$\mathcal{L}(\theta) \equiv \mathcal{L}(\theta|x^n) = p(x^n|\theta), \tag{A.7}$$

so that x^n is fixed and θ varies within Θ. The likelihood, therefore, expresses the plausibility of different values of θ for a given sample x^n. This expression is usually extended into the *log-likelihood function*

$$\ell(\theta) = log\mathcal{L}(\theta), \tag{A.8}$$

by taking the natural logarithm of the likelihood.

Definition 5. (Score). Given a random vector $X^n = (X_1, X_2, ..., X_n)$ with a joint density $p(x^n|\theta) = p(x_1, x_2, ..., x_n|\theta)$, where $\theta \in \Theta$ corresponds to a parameter vector the *score* [71] defines the gradient of the log-likelihood function w.r.t. the parameter vector θ:

$$\mathcal{S}(\theta) \equiv \nabla_\theta \ell(\theta) = \nabla_\theta \log p(x^n|\theta), \tag{A.9}$$

which is the *sensitivity* of the model p in regards to changes in the parameter vector θ.

Definition 6. (Fisher information). Given a parameterized model $p(x|\theta)$, where θ corresponds to a single parameter and X to a discrete random variable, the *Fisher information* [71] $\mathcal{F}_\theta$ is defined as

$$\begin{aligned} &\mathcal{F}_\theta = \sum_{x \in X} (\nabla_\theta \log f(x|\theta))^2 p(x|\theta), \\ &\textit{where } 0 \leq \mathcal{F}_\theta < \infty. \end{aligned} \tag{A.10}$$

For a continuous X, the sum is replaced by an integral. W.r.t. the chance $p(x|\theta) = f(x|\theta)$, the Fisher information describes the overall sensitivity of the functional relationship f to any change of θ by weighting the sensitivity at each potential outcome x [71]. As a result, we can measure the amount of information that X owns about θ, upon which the functional relationship $f(x|\theta)$ depends on. For a random vector X^n about θ the Fisher information can be defined by replacing $f(x|\theta)$ with $f(x^n|\theta)$ and $p(x|\theta)$ with $p(x^n|\theta)$. Given a vector of parameters, such that $\theta = (\theta_1, ..., \theta_n)$, the Fisher information becomes a positive semidefinite symmetric matrix of size $k \times k$ [71] defined as [87]:

$$\mathcal{F}_\theta = \mathbb{E}_{x \sim p} \left[(\nabla_\theta logp(x|\theta))^T (\nabla_\theta logp(x|\theta)) \right]. \tag{A.11}$$

Note that $\mathcal{H}$ denotes the Hessian, while H is reserved for the entropy and cross-entropy.

It has been shown [51] that $\mathcal{F}$ becomes the negative expected square matrix of second-order partial derivatives, commonly known as *Hessian* $\mathcal{H}$, of the log likelihood $logp(x|\theta)$, such that

$$\mathcal{F}_\theta = -\mathbb{E}_{x \sim p} \left[\mathcal{H}_{logp(x|\theta))} \right], \tag{A.12}$$

which identifies $\mathcal{F}_\theta$ as a curvature matrix in the distribution space, where the KL divergence appears to be the metric. Given two parameterized distributions $p(x|\theta)$ and $p(x|\theta')$, the Fisher information matrix $\mathcal{F}_\theta$ defines a quadratic approximation of the KL divergence between them when evaluated with $\theta' = \theta$ [73]. Such proof requires a second-order Taylor series expansion of the KL divergence [73] and relies on the fact that, given $\theta' = \theta$, the KL divergence $D_{KL}(p(x|\theta) \mid\mid p(x|\theta + d))$ becomes approximately symmetric in its local neighborhood as the distance d between the new and the old parameter goes to zero [3] .

Definition 7. (Hadamard product). Let X and Y be $m \times n$ equal-size matrices. The *Hadamard product* is then defined by

$$[X \odot Y]_{ij} = [X]_{ij} Y]_{ij}, \tag{A.13}$$

for all $1 \leq i \leq m$, $1 \leq j \leq n$. The Hadamard product is, hence, the simple entry-wise multiplication of two matrices, which is *associative*, *distributive*, and unlike the normal matrix product, also *commutative*.

Definition 8. (L^2 projection). Let L^2 be the so-called *Hilbert Space* [39], which is an arbitrary defined abstract vector space that allows the transformation of a partial differential equation into an inner product of two functions [12, 39]. A projection of an arbitrary function $f \in L^2(\Omega)$ into a finite element space $V_h \subset L_2(\Omega)$ where $\Omega \subset \mathbb{R}^d$ we then call L^2-projection. Such projection may be solved as an optimization problem by minimizing [12] the following objective:

$$minimize \; \frac{1}{2}(||f_h - f||)^2_{L^2(\Omega)}. \tag{A.14}$$

Appendix B

Appendix of Experiments

This appendix includes supplemental material, such as the full results of different experiment evaluations based on four environments of the OpenAI Gym [13]. We additionally include the default settings, which have been used for all experiments, unless otherwise noted.

B.1 Default Parameters

We disclose the default settings for all experiments in Table B.1. The network architecture had a "compressing" structure as described in Section 4.4. The Adam step size of the critic network was dynamically adjusted based on the KL divergence, between the old and the new policy.

Hyper-parameter	Value
Segment size (S)	20 trajectories
Horizon (T)	∞
Algorithm	clipping
Clipping ϵ	0.2
Adaptive KL target d_{targ}	0.01
Activation function	tanh
Neural network factor nn_{size}	10
Mini-batch size	256
Discount (γ)	0.995
GAE parameter (λ)	0.98
Num. epochs actor	20
Num. epochs critic	10
Adam step size actor	adaptive
Adam step size critic	0.00147
Network architecture	compressing

TABLE B.1: The experiment default settings.

B.2 Importance of Hyper-Parameters

This section includes a detailed analysis of the performance comparison of the hyper-parameter experiments. First, different empirical returns for various policy losses are shown in Figure B.1. Then, we continue investigations with the clipped PPO version and display different variations for the entropy coefficient in Figure B.2, the activation function in Figure B.3, and the weight initialization in Figure B.4. On top of that, we include experiments for dynamic segment sizes in Figure B.5, fixed segment sizes in Figure B.6, and a comparison between the best performing fixed against the best performing dynamic segment size solutions in Figure B.7.

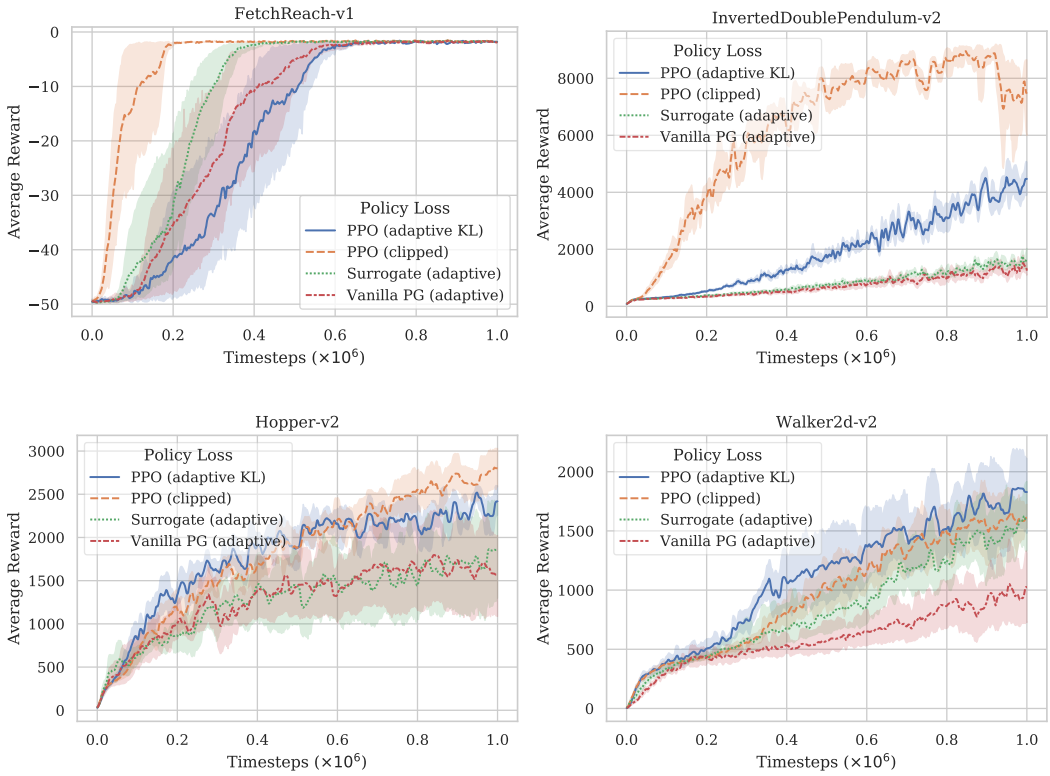


FIGURE B.1: Benchmark results of several policy gradient losses.

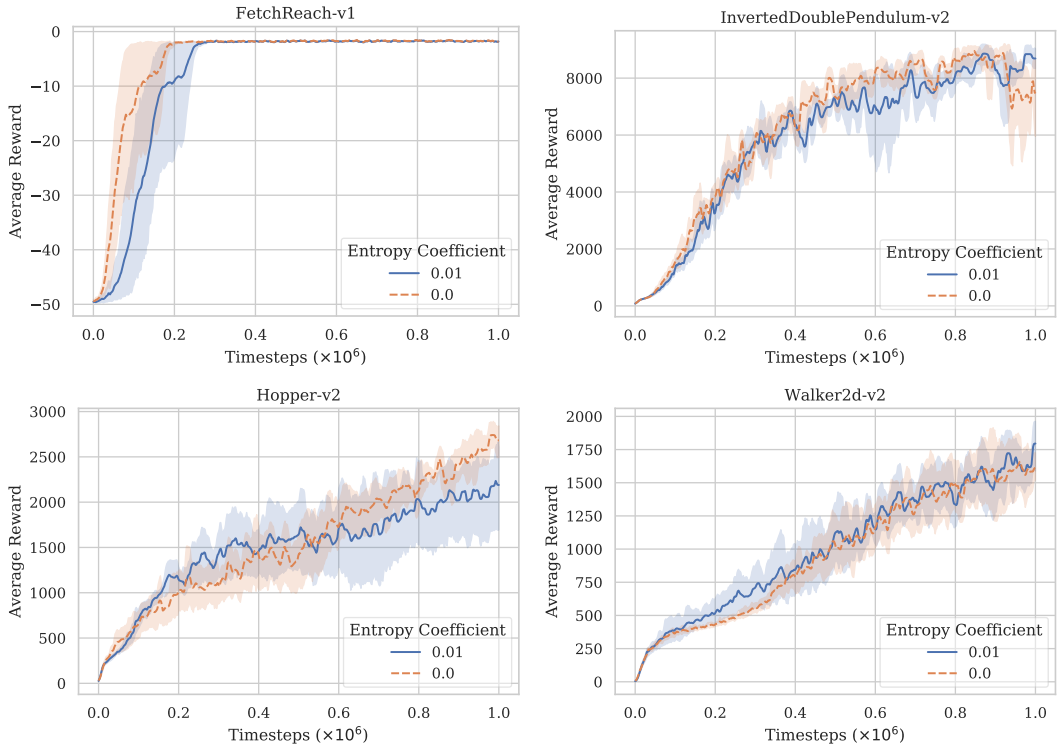


FIGURE B.2: Entropy coefficient experiments.

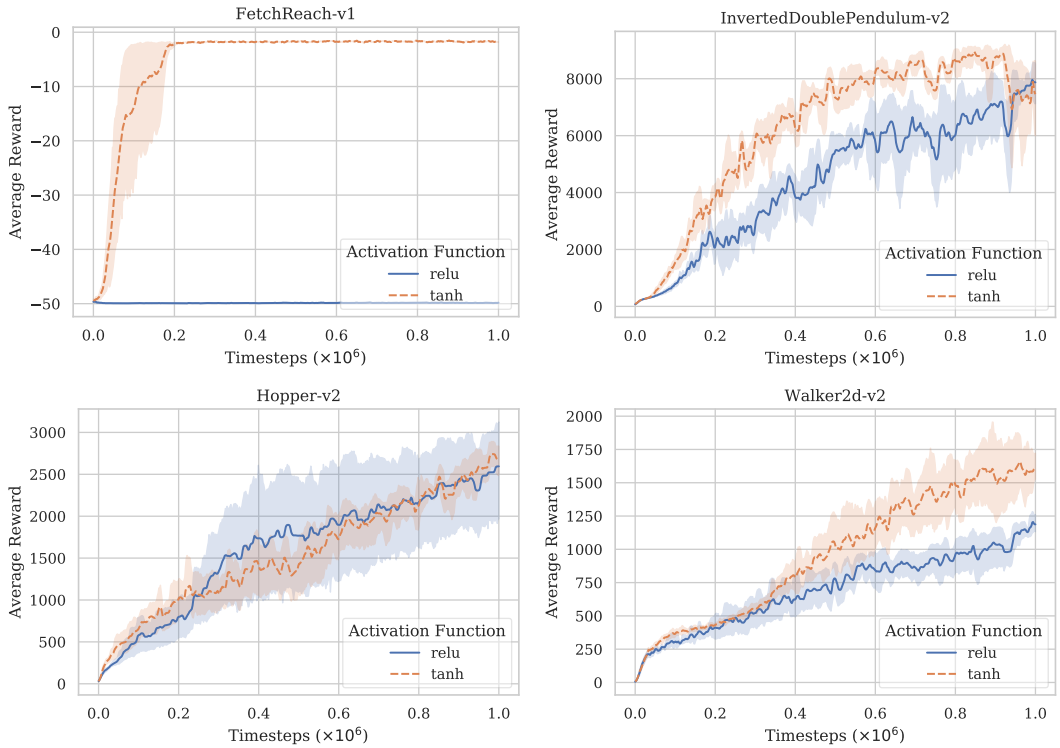


FIGURE B.3: Activation function experiments.

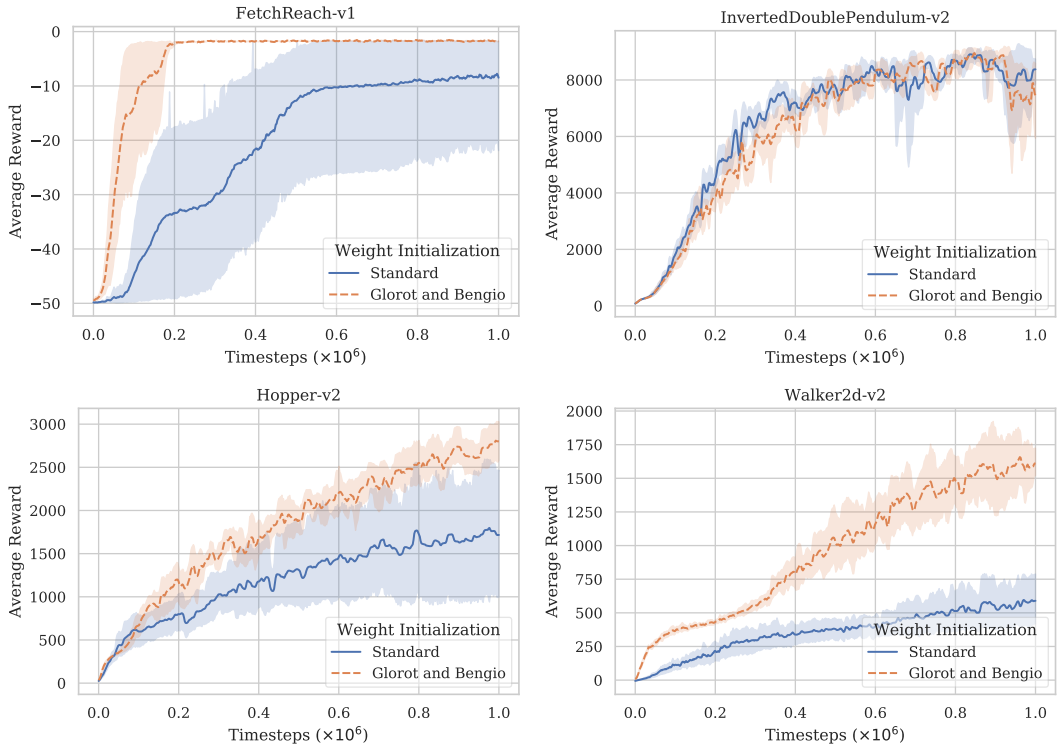


FIGURE B.4: Weight initialization experiments.

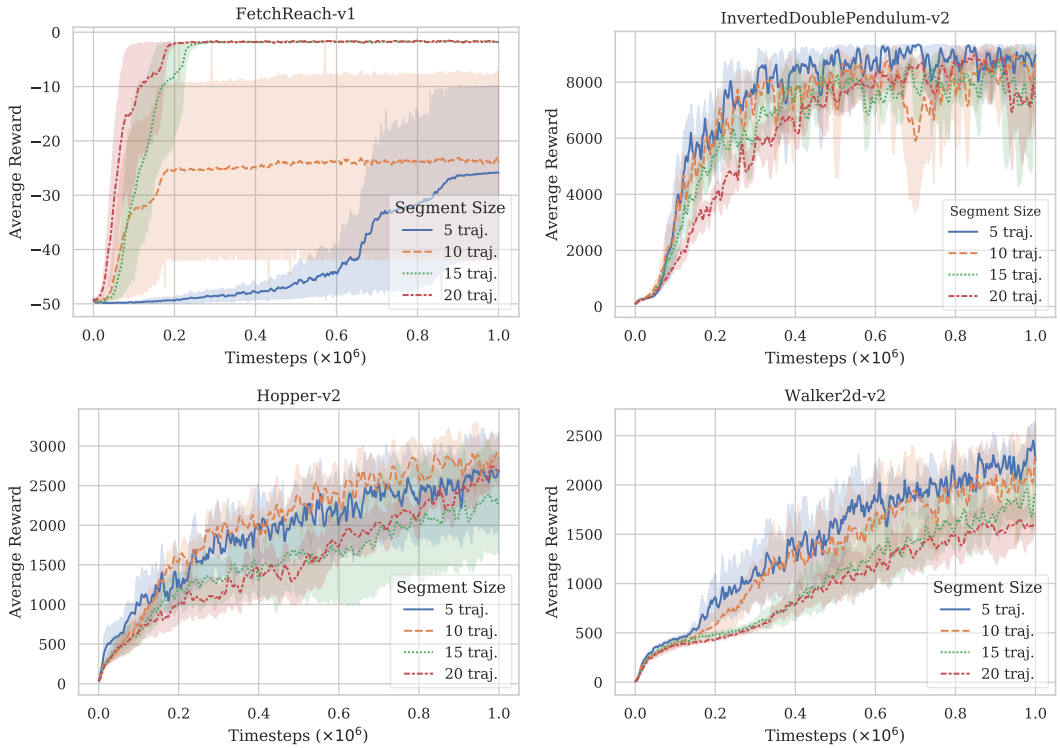


FIGURE B.5: Dynamic horizon experiments.

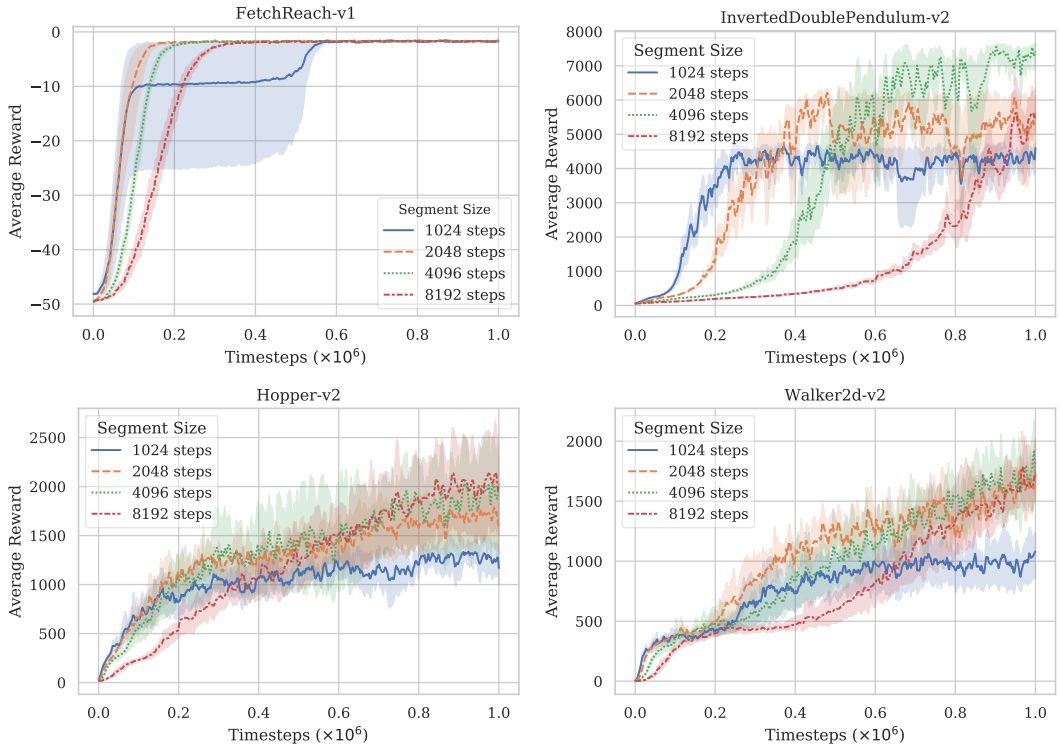


FIGURE B.6: Fixed horizon experiments.

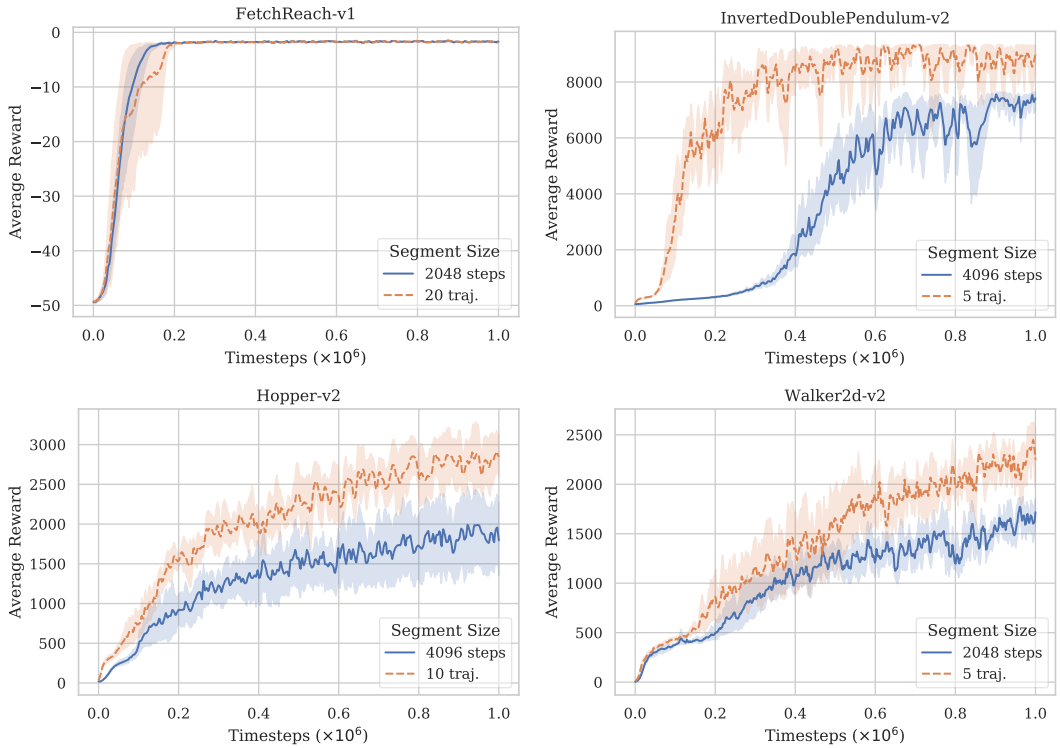


FIGURE B.7: Best dynamic horizon vs. best fixed horizon experiments.

BIBLIOGRAPHY

[1] Rishabh Agarwal, Dale Schuurmans, and Mohammad Norouzi. *Striving for Simplicity in Off-policy Deep Reinforcement Learning*. 2019. arXiv: 1907.04543. URL: http://arxiv.org/abs/1907.04543.

[2] Zafarali Ahmed et al. "Understanding the impact of entropy on policy optimization". In: *The Thirty-Second AAAI Conference on Artificial Intelligence* (2018). arXiv: 1811.11214. URL: http://arxiv.org/abs/1811.11214.

[3] Larry Armijo. "Minimization of functions having Lipschitz continuous first partial derivatives". In: *Pacific Journal of Mathematics* 16.1 (1966), pp. 1–3. ISSN: 0030-8730. DOI: 10.2140/pjm.1966.16.1. URL: https://projecteuclid.org/euclid.pjm/1102995080.

[4] Kai Arulkumaran et al. *Deep reinforcement learning: A brief survey*. 2017. DOI: 10.1109/MSP.2017.2743240. arXiv: 1708.05866v2. URL: https://arxiv.org/pdf/1708.05866.pdf.

[5] Joyce N Barlin et al. "Classification and regression tree (CART) analysis of endometrial carcinoma: Seeing the forest for the trees." In: *Gynecologic oncology* 130.3 (2013), pp. 452–6. ISSN: 1095-6859. DOI: 10.1016/j.ygyno.2013.06.009.

[6] Marc G Bellemare, Will Dabney, and Rémi Munos. *A Distributional Perspective on Reinforcement Learning*. Tech. rep. 2017. arXiv: 1707.06887v1. URL: https://arxiv.org/pdf/1707.06887.pdf.

[7] Richard Bellman. "Dynamic Programming". In: *Science* 153.3731 (1966), pp. 34–37. DOI: 10.1201/b10384.

[8] Richard Bellman. "The Theory of Dynamic Programming". In: *Bulletin of the American Mathematical Society* 60.6 (1954), pp. 503–515. ISSN: 02730979. DOI: 10.1090/S0002-9904-1954-09848-8. URL: http://www.pnas.org/cgi/doi/10.1073/pnas.38.8.716.

[9] Dimitri P. Bertsekas. *Neuro-Dynamic Programming*. Boston, MA: Springer US, 1996. DOI: 10.1007/978-0-387-74759-0_440. URL: http://www.springerlink.com/index/10.1007/978-0-387-74759-0{_}440.

[10] Léonard Blier and Yann Ollivier. "The Description Length of Deep Learning Models". In: *Advances in Neural Information Processing Systems 31* (2018), pp. 2216–2226. arXiv: 1802.07044. URL: http://arxiv.org/abs/1802.07044.

[11] Léon Bottou. "Large-Scale Machine Learning with Stochastic Gradient Descent". In: *Proceedings of COMPSTAT'2010*. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186. DOI: 10.1007/978-3-7908-2604-3_16.

[12] James H Bramble, Joseph E Pasciak, and Olaf Steinbach. *ON THE STABILITY OF THE L 2 PROJECTION IN H 1 (Ω)*. Tech. rep. 2001.

[13] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540.

[14] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: *CoRR* abs/1511.0 (2015). arXiv: 1511.07289. URL: http://arxiv.org/abs/1511.07289.

[15] Karl Cobbe et al. "Quantifying Generalization in Reinforcement Learning". In: *ArXiv* abs/1812.0 (2018). arXiv: 1812.02341. URL: http://arxiv.org/abs/1812.02341.

[16] Andrew R. Conn, Nicholas I. M. Gould, and Philippe L. Toint. *Trust Region Methods*. Society for Industrial and Applied Mathematics, 2000. ISBN: 978-0-89871-460-9. DOI: 10.1137/1.9780898719857. URL: http://epubs.siam.org/doi/book/10.1137/1.9780898719857.

[17] Thomas Degris, Martha White, and Richard S Sutton. *Off-Policy Actor-Critic*. Tech. rep. 2012. arXiv: 1205.4839v5. URL: https://arxiv.org/pdf/1205.4839.pdf.

[18] Ronald A. Devore. "Nonlinear approximation". In: *Acta Numerica* 7 (1998), pp. 51–150. ISSN: 14740508. DOI: 10.1017/S0962492900002816.

[19] Peter Dhariwal, Prafulla and Hesse, Christopher and Klimov, Oleg and Nichol, Alex and Plappert, Matthias and Radford, Alec and Schulman, John and Sidor, Szymon and Wu, Yuhuai and Zhokhov. *OpenAI Baselines*. 2017. URL: https://github.com/openai/baselines (visited on 10/10/2019).

[20] Kenji Doya. "Reinforcement learning: Computational theory and biological mechanisms." In: *HFSP journal* 1.1 (2007), pp. 30–40. ISSN: 1955-2068. DOI: 10.2976/1.2732246/10.2976/1. URL: http://www.ncbi.nlm.nih.gov/pubmed/19404458.

[21] Pierre Dragicevic. "Fair Statistical Communication in HCI". In: *Modern Statistical Methods for HCI*. 2016, pp. 291–330. DOI: 10.1007/978-3-319-26633-6_13. URL: http://link.springer.com/10.1007/978-3-319-26633-6{_}13.

[22] Yan Duan et al. "Benchmarking Deep Reinforcement Learning for Continuous Control". In: *CoRR* (2016). arXiv: 1604.06778. URL: http://arxiv.org/abs/1604.06778.

[23] John Duchi. *Derivations for Linear Algebra and Optimization*. Tech. rep. 2007, pp. 1–13. URL: http://web.stanford.edu/{~}jduchi/projects/general{_}notes.pdf.

[24] Damien Ernst, Pierre Geurts, and Louis Wehenkel. "Tree-based batch mode reinforcement learning". In: *Journal of Machine Learning Research* 6 (2005). ISSN: 15337928.

[25] Chelsea Finn and Sergey Levine. "Deep Visual Foresight for Planning Robot Motion". In: *CoRR* (2016). arXiv: 1610.00696. URL: http://arxiv.org/abs/1610.00696.

[26] Meire Fortunato et al. "Noisy Networks for Exploration". In: *CoRR* abs/1706.1 (2017). arXiv: 1706.10295. URL: http://arxiv.org/abs/1706.10295.

[27] Martin Fowler. *TechnicalDebt*. 2003. URL: https://www.martinfowler.com/bliki/TechnicalDebt.html (visited on 10/10/2019).

[28] Scott Fujimoto, Herke van Hoof, and David Meger. "Addressing Function Approximation Error in Actor-Critic Methods". In: *CoRR* abs/1802.0 (2018). arXiv: 1802.09477. URL: http://arxiv.org/abs/1802.09477.

[29] Thomas Gabel. "Multi-Agent Reinforcement Learning Approaches for Distributed Job-Shop Scheduling Problems". PhD thesis. University of Osnabruck, 2009.

[30] Tianxiang Gao and Vladimir Jojic. "Degrees of Freedom in Deep Neural Networks". In: *CoRR* abs/1603.0 (2016). arXiv: 1603.09260. URL: http://arxiv.org/abs/1603.09260.

[31] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Journal of Machine Learning Research*. Vol. 9. 2010, pp. 249–256.

[32] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Journal of Machine Learning Research*. Vol. 15. 2011, pp. 315–323. URL: http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf.

[33] Geoffrey J. Gordon. "Stable Function Approximation in Dynamic Programming". In: *Machine Learning Proceedings 1995* (1995), pp. 261–268. DOI: 10.1016/B978-1-55860-377-6.50040-2. URL: https://www.sciencedirect.com/science/article/pii/B9781558603776500402.

[34] Alex Graves and Navdeep Jaitly. "Towards end-to-end speech recognition with recurrent neural networks". In: *31st International Conference on Machine Learning, ICML 2014* 5 (2014), pp. 1764–1772.

[35] Evan Greensmith et al. *Variance Reduction Techniques for Gradient Estimates in Reinforcement Learning*. Tech. rep. 2004, pp. 1471–1530. URL: http://jmlr.csail.mit.edu/papers/volume5/greensmith04a/greensmith04a.pdf.

[36] Shixiang Gu et al. "Continuous deep Q-learning with model-based acceleration". In: *33rd International Conference on Machine Learning, ICML 2016*. Vol. 6. 2016, pp. 4135–4148. ISBN: 9781510829008. arXiv: 1603.00748. URL: https://arxiv.org/pdf/1603.00748.pdf.

[37] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *CoRR* abs/1801.0 (2018). arXiv: 1801.01290. URL: http://arxiv.org/abs/1801.01290.

[38] Roland Hafner and Martin Riedmiller. "Reinforcement learning in feedback control". In: *Machine Learning* 84.1-2 (2011), pp. 137–169. ISSN: 0885-6125. DOI: 10.1007/s10994-011-5235-x. URL: http://link.springer.com/10.1007/s10994-011-5235-x.

[39] Paul R. Halmos. *Introduction to Hilbert Space: And the Theory of Spectral Multiplicity*. 1998.

[40] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.0 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[41] Hado van Hasselt et al. "Deep Reinforcement Learning and the Deadly Triad". In: *DEEP RL WORKSHOP NEURIPS* (2018). arXiv: 1812.02648. URL: http://arxiv.org/abs/1812.02648.

[42] Matthew Hausknecht et al. "A Neuroevolution Approach to General Atari Game Playing". In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.4 (2014), pp. 355–366. ISSN: 1943068X. DOI: 10.1109/TCIAIG.2013.2294713.

[43] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[44] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE International Conference on Computer Vision*. Vol. 2015 Inter. 2015, pp. 1026–1034. ISBN: 9781467383912. DOI: 10.1109/ICCV.2015.123. arXiv: 1502.01852. URL: http://arxiv.org/abs/1502.01852.

[45] Nicolas Heess et al. "Emergence of Locomotion Behaviours in Rich Environments". In: *CoRR* abs/1707.0 (2017). arXiv: 1707.02286. URL: http://arxiv.org/abs/1707.02286.

[46] Peter Henderson et al. "Deep reinforcement learning that matters". In: *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*. 2018, pp. 3207–3214. ISBN: 9781577358008. arXiv: 1709.06560v3. URL: www.aaai.org.

[47] James Hendler. "Avoiding Another AI Winter". In: *IEEE Intelligent Systems* 23.2 (2008), pp. 2–4. ISSN: 1541-1672. DOI: 10.1109/MIS.2008.20. URL: http://ieeexplore.ieee.org/document/4475849/.

[48] Matteo Hessel et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)* (2017). arXiv: 1710.02298. URL: http://arxiv.org/abs/1710.02298.

[49] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735.

[50] Rob J. Hyndman. "Moving Averages". In: *International Encyclopedia of Statistical Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 866–869. DOI: 10.1007/978-3-642-04898-2_380. URL: http://link.springer.com/10.1007/978-3-642-04898-2{_}380.

[51] H. JEFFREYS. "An invariant form for the prior probability in estimation problems." In: *Proceedings of the Royal Society of London. Series A, Mathematical and physical sciences* 186.1007 (1946), pp. 453–461. ISSN: 00804630. DOI: 10.1098/rspa.1946.0056. URL: http://www.royalsocietypublishing.org/doi/10.1098/rspa.1946.0056.

[52] Niels Justesen et al. *Illuminating Generalization in Deep Reinforcement Learning through Procedural Level Generation*. 2018. arXiv: 1806.10729. URL: http://arxiv.org/abs/1806.10729.

[53] Sham Kakade. "A Natural Policy Gradient". In: *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic* 14 (2001), pp. 1531–1538.

[54] Sham Kakade and John Langford. "Approximately Optimal Approximate Reinforcement Learning". In: *Proceedings of the 19th International Conference on Machine Learning* (2002), pp. 267–274.

[55] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference for Learning Representations, San Diego, 2015* (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[56] Jens Kober and Jan Peters. "Reinforcement Learning in Robotics: A Survey". In: Springer, Cham, 2014, pp. 9–67. DOI: 10.1007/978-3-319-03194-1_2.

[57] Vijaymohan Konda. "Actor-critic Algorithms". PhD thesis. Massachusetts Institute of Technology, 2002. DOI: Konda:2002:AA:936987. URL: https://dl.acm.org/citation.cfm?id=936987.

[58] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems*. Vol. 2. 2012, pp. 1097–1105. ISBN: 9781627480031. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.299.205.

[59] Solomon; Kullback. *Information theory and statistics*. Courier Corporation, 1997. ISBN: 0486696847.

[60] Sascha Lange and Martin Riedmiller. "Deep auto-encoder neural networks in reinforcement learning". In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2010, pp. 1–8. ISBN: 978-1-4244-6916-1. DOI: 10.1109/IJCNN.2010.5596468. URL: http://ieeexplore.ieee.org/document/5596468/.

[61] Kenneth Langed, Avid R Hunter, and Ilsoon Yang. "Optimization Transfer Using Surrogate Objective Functions". In: *Journal of Computational and Graphical Statistics* 9.1 (2000), pp. 1–20. ISSN: 15372715. DOI: 10.1080/10618600.2000.10474858. URL: http://www.yaroslavvb.com/papers/lange-optimization.pdf.

[62] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. ISSN: 00189219. DOI: 10.1109/5.726791. URL: http://ieeexplore.ieee.org/document/726791/.

[63] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444. ISSN: 0028-0836. DOI: 10.1038/nature14539. URL: http://www.nature.com/articles/nature14539.

[64] Pierre L'Ecuyer. "On the Interchange of Derivative and Expectation for Likelihood Ratio Derivative Estimators". In: *Management Science* 41 (1995), pp. 738–748. DOI: 10.2307/2632893. URL: https://www.jstor.org/stable/2632893.

[65] Sergey Levine, Nolan Wagener, and Pieter Abbeel. "Guided Policy Search". In: *Proceedings - IEEE International Conference on Robotics and Automation* 2015-June.June (2015), pp. 156–163. ISSN: 10504729. DOI: 10.1109/ICRA.2015.7138994. arXiv: 1501.05611.

[66] Chunyuan Li et al. "Measuring the Intrinsic Dimension of Objective Landscapes". In: *CoRR* abs/1804.0 (2018). arXiv: 1804.08838. URL: http://arxiv.org/abs/1804.08838.

[67] Yuxi Li. "Deep Reinforcement Learning: An Overview". In: *CoRR* abs/1710.0 (2017). arXiv: 1701.07274. URL: http://arxiv.org/abs/1701.07274.

[68] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *ICLR 2015* (2015). arXiv: 1509.02971. URL: http://arxiv.org/abs/1509.02971.

[69] Long-Ji Lin. "Reinforcement learning for robots using neural networks". PhD thesis. Carnegie Mellon University, 1993. URL: http://www.incompleteideas.net/lin-92.pdf.

[70] Long-Ji Lin. *Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching*. Tech. rep. 1992, pp. 293–321. URL: http://www.incompleteideas.net/lin-92.pdf.

[71] Alexander Ly et al. "A Tutorial on Fisher information". In: *Journal of Mathematical Psychology* 80 (2017), pp. 40–55. ISSN: 10960880. DOI: 10.1016/j.jmp.2017.05.006. arXiv: 1705.01064v2. URL: https://arxiv.org/pdf/1705.01064.pdf.

[72] K. Z. Mao, K. C. Tan, and W. Ser. "Probabilistic neural-network structure determination for pattern classification". In: *IEEE Transactions on Neural Networks* 11.4 (2000), pp. 1009–1016. ISSN: 10459227. DOI: 10.1109/72.857781. URL: http://ieeexplore.ieee.org/document/857781/.

[73] James Martens. "New insights and perspectives on the natural gradient method". In: (2014). arXiv: 1412.1193. URL: https://arxiv.org/pdf/1412.1193.pdf.

[74] Volodymyr Mnih et al. *Asynchronous Methods for Deep Reinforcement Learning*. Tech. rep. 2016. arXiv: 1602.01783v2. URL: https://arxiv.org/pdf/1602.01783.pdf.

[75] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533. ISSN: 0028-0836. DOI: 10.1038/nature14236. URL: http://www.nature.com/articles/nature14236.

[76] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Tech. rep. 2013. arXiv: 1312.5602v1. URL: https://arxiv.org/pdf/1312.5602v1.pdf.

[77] Martin Fodslette Møller. "A scaled conjugate gradient algorithm for fast supervised learning". In: *Neural Networks* 6.4 (1993), pp. 525–533. ISSN: 08936080. DOI: 10.1016/S0893-6080(05)80056-5. URL: https://www.sciencedirect.com/science/article/pii/S0893608005800565.

[78] J. E Moody, J. E., Hanson, S. J., & Moody. "An Analysis of Generalization and Regularization in Nonlinear Learning Systems". In: *Neural Information Processing Systems (NIPS)* 1 (1992), pp. 847–854.

[79] David E. Moriarty, Alan C. Schultz, and John J. Grefenstette. "Evolutionary Algorithms for Reinforcement Learning". In: *Journal of Artificial Intelligence Research* 11 (1999), pp. 241–276. ISSN: 10769757. DOI: 10.1613/jair.613. URL: https://jair.org/index.php/jair/article/view/10240.

[80] Rémi Munos et al. "Safe and Efficient Off-Policy Reinforcement Learning". In: *CoRR* (2016). arXiv: 1606.02647. URL: http://arxiv.org/abs/1606.02647.

[81] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines Vinod Nair". In: *Proceedings of ICML*. Vol. 27. 2010, pp. 807–814.

[82] Alex Nichol et al. "Gotta Learn Fast: A New Benchmark for Generalization in RL". In: *CoRR* (2018). arXiv: 1804.03720. URL: http://arxiv.org/abs/1804.03720.

[83] OpenAI. *OpenAI Five*. URL: https://blog.openai.com/openai-five/ (visited on 10/10/2019).

[84] OpenAI et al. *Solving Rubik's Cube with a Robot Hand*. 2019. arXiv: 1910.07113. URL: http://arxiv.org/abs/1910.07113.

[85] Leslie Pack Kaelbling et al. *Reinforcement Learning: A Survey*. Tech. rep. 1996, pp. 237–285. URL: https://arxiv.org/pdf/cs/9605103.pdf.

[86] Charles Packer et al. "Assessing Generalization in Deep Reinforcement Learning". In: *CoRR* (2018). arXiv: 1810.12282. URL: http://arxiv.org/abs/1810.12282.

[87] Razvan Pascanu and Yoshua Bengio. "Revisiting Natural Gradient for Deep Networks". In: *CoRR* (2013). arXiv: 1301.3584. URL: https://arxiv.org/pdf/1301.3584.pdf.

[88] Matthias Plappert et al. "Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research". In: *CoRR* (2018). arXiv: 1802.09464. URL: http://arxiv.org/abs/1802.09464.

[89] Jordan B. Pollack, Jordan B. Pollack, and Alan D. Blair. "Why did TD-Gammon Work?" In: *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 9* 9 (1997), pp. 10–16. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.2313.

[90] Martin L. Puterman and Wiley InterScience (Online service). *Markov decision processes : discrete stochastic dynamic programming*. Wiley, 1994, p. 649. ISBN: 9780470316887.

[91] Aravind Rajeswaran et al. "Towards Generalization and Simplicity in Continuous Control". In: *CoRR* (2017). URL: https://sites.google.com/view/simple-pol.

[92] Martin Riedmiller. "Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method". In: Springer, Berlin, Heidelberg, 2005, pp. 317–328. DOI: 10.1007/11564096_32. URL: http://link.springer.com/10.1007/11564096{_}32.

[93] Martin Riedmiller and Heinrich Braun. "Direct adaptive method for faster backpropagation learning: The RPROP algorithm". In: *1993 IEEE International Conference on Neural Networks*. IEEE, 1993, pp. 586–591. ISBN: 0780312007. DOI: 10.1109/icnn.1993.298623. URL: http://ieeexplore.ieee.org/document/298623/.

[94] Martin Riedmiller et al. "Reinforcement learning for robot soccer". In: *Autonomous Robots* 27.1 (2009), pp. 55–73. ISSN: 0929-5593. DOI: 10.1007/s10514-009-9120-4. URL: http://link.springer.com/10.1007/s10514-009-9120-4.

[95] David Rolnick et al. *Deep Learning is Robust to Massive Label Noise*. 2017. arXiv: 1705.10694. URL: http://arxiv.org/abs/1705.10694.

[96] Reuven Y. Rubinstein and Dirk P. Kroese. *Simulation and the Monte Carlo Method: Third Edition*. Wiley Series in Probability and Statistics. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2016, pp. 1–414. ISBN: 9781118631980. DOI: 10.1002/9781118631980. URL: http://doi.wiley.com/10.1002/9781118631980.

[97] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *ArXiv* (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[98] Shibani Santurkar et al. "How does batch normalization help optimization?" In: *Advances in Neural Information Processing Systems*. Vol. 2018-Decem. 2018, pp. 2483–2493. arXiv: 1805.11604. URL: http://arxiv.org/abs/1805.11604.

[99] Sascha Lange, Thomas Gabel and Martin Riedmiller. "Batch Reinforcement Learning". In: *Reinforcement Learning. Adaptation, Learning, and Optimization*. Vol. 4. 7540. Springer, Berlin, Heidelberg, 2012, pp. 45–73. ISBN: 978-1-4799-0356-6. DOI: 10.1038/nature14236. arXiv: 1502.04623.

[100] Tom Schaul et al. "Prioritized Experience Replay". In: *CoRR* (2015). arXiv: 1511.05952. URL: http://arxiv.org/abs/1511.05952.

[101] Oliver Schoppe et al. "Measuring the performance of neural models". In: *Frontiers in Computational Neuroscience* 10.FEB (2015), p. 10. ISSN: 16625188. DOI: 10.3389/fncom.2016.00010.

[102] John Schulman. *modular_rl*. 2017. URL: https://github.com/joschu/modular{_}rl (visited on 10/10/2019).

[103] John Schulman. "Optimizing Expectations: From Deep Reinforcement Learning to Stochastic Computation Graphs". PhD thesis. EECS Department, University of California, Berkeley, 2016.

[104] John Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *CoRR* (2015). arXiv: 1506.02438. URL: http://arxiv.org/abs/1506.02438.

[105] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *ArXiv* (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[106] John Schulman et al. "Trust Region Policy Optimization". In: *CoRR* (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[107] W Schultz, P Dayan, and P R Montague. "A neural substrate of prediction and reward." In: *Science (New York, N.Y.)* 275.5306 (1997), pp. 1593–9. ISSN: 0036-8075. URL: http://www.ncbi.nlm.nih.gov/pubmed/9054347.

[108] C E Shannon. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27.3 (1948), pp. 379–423. ISSN: 15387305. DOI: 10.1002/j.1538-7305.1948.tb01338.x. URL: http://math.harvard.edu/{~}ctm/home/text/others/shannon/entropy/entropy.pdf.

[109] David Silver et al. "Deterministic Policy Gradient Algorithms". In: *31st International Conference on Machine Learning, ICML 2014*. Vol. 1. 2014, pp. 605–619. ISBN: 9781634393973. URL: http://proceedings.mlr.press/v32/silver14.pdf.

[110] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. Tech. rep. 2017. arXiv: 1712.01815v1. URL: https://arxiv.org/pdf/1712.01815.pdf.

[111] David Silver et al. *Mastering the game of Go with deep neural networks and tree search*. 2016. URL: https://ai.google/research/pubs/pub44806.

[112] Leslie N. Smith. "Cyclical learning rates for training neural networks". In: *Proceedings - 2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017*. 2017, pp. 464–472. ISBN: 9781509048229. DOI: 10.1109/WACV.2017.58. arXiv: 1506.01186. URL: http://arxiv.org/abs/1506.01186.

[113] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning : an introduction*. MIT Press, 1998, p. 322. ISBN: 0262193981. URL: https://dl.acm.org/citation.cfm?id=551283.

[114] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. Second edi. The MIT Press Cambridge, Massachusetts, 2017. URL: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

[115] Richard S. Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *IN ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 12* 12 (2000), pp. 1057–1063. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.79.5189.

[116] István Szita and András Lorincz. "Learning Tetris Using the Noisy Cross-Entropy Method". In: *Neural Computation* 18.12 (2006), pp. 2936–2941. ISSN: 08997667. DOI: 10.1162/neco.2006.18.12.2936.

[117] Gerald Tesauro. "Temporal Difference Learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995), pp. 58–68. ISSN: 15577317. DOI: 10.1145/203330.203343. URL: http://portal.acm.org/citation.cfm?doid=203330.203343.

[118] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control". In: *IEEE International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033. ISBN: 9781467317375. DOI: 10.1109/IROS.2012.6386109. URL: http://ieeexplore.ieee.org/document/6386109/.

[119] J.N. Tsitsiklis and B. Van Roy. "An analysis of temporal-difference learning with function approximation". In: *IEEE Transactions on Automatic Control* 42.5 (1997), pp. 674–690. ISSN: 00189286. DOI: 10.1109/9.580874. URL: http://ieeexplore.ieee.org/document/580874/.

[120] Arash Vahdat. "Toward Robustness against Label Noise in Training Deep Discriminative Neural Networks". In: *CoRR* (2017). arXiv: 1706.00038. URL: http://arxiv.org/abs/1706.00038.

[121] Oriol Vinyals et al. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. 2019. URL: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/ (visited on 10/10/2019).

[122] Junpeng Wang et al. "DQNViz: A Visual Analytics Approach to Understand Deep Q-Networks". In: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), pp. 288–298. ISSN: 1077-2626. DOI: 10.1109/TVCG.2018.2864504. URL: https://ieeexplore.ieee.org/document/8454905/.

[123] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: *CoRR* (2015). arXiv: 1511.06581. URL: https://arxiv.org/abs/1511.06581.

[124] Ziyu Wang et al. "Sample Efficient Actor-Critic with Experience Replay". In: *CoRRR* (2016). arXiv: 1611.01224. URL: http://arxiv.org/abs/1611.01224.

[125] Christopher John Cornish Hellaby Watkins. "Learning from Delayed Rewards". PhD thesis. Cambridge, UK: King's College, 1989. URL: http://www.cs.rhul.ac.uk/{~}chrisw/new{_}thesis.pdf.

[126] Ronald J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.3-4 (1992), pp. 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696. URL: http://link.springer.com/10.1007/BF00992696.

[127] Yuhuai Wu et al. "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation". In: *Advances in Neural Information Processing Systems*. Vol. 2017-Decem. 2017, pp. 5280–5289. arXiv: 1708.05144. URL: https://arxiv.org/abs/1708.05144.

[128] Zhuora Yang, Yuchen Xie, and Zhaoran Wang. "A Theoretical Analysis of Deep Q-Learning". In: *CoRR* (2019). arXiv: 1901.00137. URL: http://arxiv.org/abs/1901.00137.

[129] Alessio Zappone, Marco Di Renzo, and Mérouane Debbah. "Wireless Networks Design in the Era of Deep Learning: Model-Based, AI-Based, or Both?" In: *CoRR* (2019). arXiv: 1902.02647. URL: http://arxiv.org/abs/1902.02647.

[130] Chiyuan Zhang et al. "A Study on Overfitting in Deep Reinforcement Learning". In: *CoRR* (2018). arXiv: 1804.06893. URL: http://arxiv.org/abs/1804.06893.

[131] Chiyuan Zhang et al. "Understanding deep learning requires rethinking generalization". In: *CoRR* (2016). arXiv: 1611.03530. URL: http://arxiv.org/abs/1611.03530.

[132] Shangtong Zhang and Richard S. Sutton. "A Deeper Look at Experience Replay". In: *CoRR* (2017). arXiv: 1712.01275. URL: http://arxiv.org/abs/1712.01275.