FH MÜNSTER
University of Applied Sciences

ETI FB Elektrotechnik und Informatik
Department of Electrical Engineering
and Computer Science

Evaluating Hyperparameter and Implementation Choices of Proximal Policy Optimization

On a Selection of ATARI 2600 Games

Master's Thesis

Daniel Lukats

submitted to obtain the degree of
Master of Science (M.Sc.) in Computer Science
at
FH Münster – University of Applied Sciences
Department of Electrical Engineering and Computer Science

July 22, 2020

Advisor Prof. Dr.-Ing. Jürgen te Vrugt
Co-Advisor Prof. Dr. Kathrin Ungru
Matriculation number *

Abstract

Reinforcement learning denotes a class of machine learning algorithms that train an agent through trial-and-error. Instead of correcting the agent's behavior, the agent is rewarded when it achieves a set out goal and punished when it fails to do so.

Recently, reinforcement learning was combined with deep learning to utilize neural networks. *Proximal Policy Optimization* is one of these deep reinforcement learning algorithms (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017). Although Proximal Policy Optimization enjoys widespread use and achieves competitive results on many benchmarking tasks, a number of non-disclosed optimizations can be found in implementations provided by the authors (OpenAI Inc., 2017a).

Therefore, this thesis introduces the mathematical foundations required to understand Proximal Policy Optimization and builds upon them to explain the algorithm itself. Afterwards, experiments are carried out to evaluate the aforementioned optimization choices as well as common configuration choices on five ATARI 2600 video games. ATARI games are a common benchmarking task for reinforcement learning algorithms.

The experiments reveal that most optimization choices have a strong effect on the performance of Proximal Policy Optimization. Furthermore, they support the authors' claims that the algorithm is robust to configuration choices. Notable outliers are apparent in approximately 35% of the experiments. As a consequence, reproducing the original results can pose a challenge.

Contents

1. Introduction **1**
1.1. Motivation . . . 1
1.2. Related Work . . . 2
1.3. Outline . . . 3
1.4. Notation . . . 3

2. Reinforcement Learning **5**
2.1. Agent and Environment . . . 5
2.2. Markov Decision Process . . . 6
2.2.1. Definition . . . 6
2.2.2. States, Actions and Rewards . . . 8
2.2.3. Dynamics Function and Policy . . . 9
2.2.4. Finite Horizon and Episodes . . . 11
2.3. Value Function and Advantage Function . . . 11
2.3.1. Return . . . 12
2.3.2. Value Function . . . 12
2.3.3. Advantage Function . . . 13
2.4. Function Approximation . . . 14
2.5. Policy Gradients . . . 15
2.5.1. Policy Parameterization . . . 15
2.5.2. Gradient Ascent . . . 16
2.5.3. Gradient Estimation . . . 18

3. Proximal Policy Optimization **19**
3.1. Motivation . . . 19
3.2. Advantage and Value Estimation . . . 20
3.2.1. Generalized Advantage Estimation . . . 21
3.2.2. λ-return . . . 22
3.3. Loss . . . 23
3.3.1. Background . . . 23
3.3.2. Clipped Surrogate Objective . . . 24
3.3.3. Value Function Loss . . . 28
3.3.4. Shared Parameterization Loss . . . 28
3.4. Algorithm . . . 29

4. Realization **33**
4.1. Environment Adaptation . . . 33
4.2. Model Architecture . . . 36
4.3. Advantage Normalization . . . 38
4.4. Initialization . . . 38
4.5. Value Function Loss Clipping . . . 39
4.6. Gradient Clipping . . . 39
4.7. Complete Algorithm . . . 40
4.8. Performance Optimization . . . 40

5. Evaluation **43**
5.1. Arcade Learning Environment . . . 43
5.2. Method . . . 44
5.3. Reproduction . . . 46
5.4. Experiments . . . 48
5.4.1. Reference Configuration . . . 48
5.4.2. Stability with Suboptimal Hyperparameter . . . 52
5.4.3. No Optimizations . . . 55
5.5. Discussion . . . 59
5.5.1. Non-disclosed Optimizations . . . 59
5.5.2. Outliers . . . 61
5.5.3. Robustness to Hyperparameter Choices . . . 62

6. Conclusion and Future Work **65**
6.1. Conclusion . . . 65
6.2. Future Work . . . 66

List of Mathematical Symbols and Definitions **67**

Glossary **69**

References **73**

A. List of Experiments **77**

1. Introduction

1.1. Motivation

In recent years reinforcement learning algorithms achieved several breakthroughs such as defeating a world champion in the game of Go (Silver et al., 2016) or defeating professional players in restricted versions of the video games StarCraft II (Vinyals et al., 2019) and Dota 2 (Berner et al., 2019). The latter was achieved using—amongst others—an algorithm called *Proximal Policy Optimization* (Schulman et al., 2017).

Proximal Policy Optimization is used in several research topics. On the one hand, researchers look to apply the algorithm to real-world tasks. For example, Proximal Policy Optimization is used to learn human walking behavior, which can aid in developing control schemes for prostheses (Anand, Zhao, Roth, & Seyfarth, 2019). Furthermore, the algorithm is utilized in spaceflight, e.g., to train agents for interplanetary transfers (Miller, Englander, & Linares, 2019) and autonomous planetary landings (Gaudet, Linares, & Furfaro, 2020). For a final example, researchers applied Proximal Policy Optimization to medical imaging in order to trace axons, which are microscopic neuronal structures (Dai et al., 2019).

On the other hand, researchers combine the algorithm with further concepts. For example, Proximal Policy Optimization can be used in meta reinforcement learning, which trains a reinforcement learning agent to train other reinforcement learning agents (Liu, Socher, & Xiong, 2019). Yet another concept that may be combined with the algorithm is curiosity (Pathak, Agrawal, Efros, & Darrell, 2017). Curiosity is a mechanism that incentivizes a methodical search for an optimal solution rather than a random search.

Despite its widespread use, Ilyas et al. (2018) as well as Engstrom et al. (2019) found that several implementation choices are undocumented in the original paper but have great effect on the performance of Proximal Policy Optimization. Consequently, the authors raise doubts on our mathematical understanding of the foundations of the algorithm.

The goal of this thesis is twofold. On the one hand, it provides the required fundamentals of reinforcement learning to understand policy gradient methods, so interested parties may be introduced to reinforcement learning. It then builds upon these fundamentals to explain Proximal Policy Optimization. In order to gain a thorough understanding of the algorithm, a dedicated implementation based on the benchmarking

framework *OpenAI Gym* and the deep learning framework *PyTorch* was written instead of relying on an open source implementation of Proximal Policy Optimization.[1]

On the other hand, this thesis examines not only the impact of the aforementioned implementation choices but also common hyperparameter choices on a selection of five ATARI 2600 games. These video games form a typical benchmarking environment for evaluating reinforcement learning algorithms. The importance of the implementation choices was already researched on robotics tasks (Ilyas et al., 2018; Engstrom et al., 2019), but the authors forewent an examination on ATARI games. Therefore, one may raise the question if these choices have the same importance for ATARI 2600 games as they do for robotics tasks.

1.2. Related Work

Proximal Policy Optimization (PPO) is a widely used reinforcement learning algorithm and therefore topic of various research questions. The publications by Ilyas et al. (2018) and Engstrom et al. (2019) are closest to the topic of this thesis, as the authors thoroughly research undocumented properties and implementation choices of the algorithm. However, their work focuses on robotics tasks and foregoes an examination of the modifications on ATARI video games. Furthermore, the authors assume prior knowledge of reinforcement learning and provide only a short background on policy gradient methods.

Multiple open source implementations of PPO are available (OpenAI Inc., 2017a; Jayasiri, n.d.; Kostrikov, 2018). As these publications mostly consist of source code, no evaluation on various tasks is available for comparison. The notable exception to this matter is the baselines repository (OpenAI Inc., 2017a), which was published alongside the original publication by Schulman et al. (2017). Among these, the work of Jayasiri (n.d.) stands out, as the author added extensive comments elaborating some concepts and implementation choices. The code created for this thesis differs, as it is suitable for learning ATARI tasks only, whereas OpenAI Inc. (2017a) and Kostrikov (2018) support robotics and control tasks as well. Jayasiri's (n.d.) code is intended for a single ATARI game only.

Lastly, PPO is topic of master's theses, for example by Chen (2018) and Güldenring (2019). The authors' theses differ from this one, as Chen (2018) researches the application of PPO to engineering tasks, whereas Güldenring (2019) examines PPO in the context of navigating a robot. Since both authors focus on application, they chose

[1] The code is available at `https://github.com/Aethiles/ppo-pytorch`.

to utilize open source implementations of PPO instead of implementing the algorithm themselves.

1.3. Outline

The fundamentals of reinforcement learning are given in chapter 2. These contain core terms and definitions required to discuss and construct reinforcement learning algorithms.

Issues with the naive learning approach outlined in chapter 2 are pointed out in chapter 3. This leads to the introduction of advanced estimation methods, which are used in *Proximal Policy Optimization*. With these estimation methods PPO is defined and ramifications of specific operations are explained. Chapter 3 closes with an outline of the complete reinforcement learning algorithm.

Undocumented design and implementation choices are elaborated and—if possible—explained in chapter 4. Before these choices are examined on ATARI 2600 games, the benchmarking framework and the evaluation method are introduced in chapter 5. A discussion of the results completes the chapter.

Finally, we summarize the results of this thesis in chapter 6 and discuss possible future work that builds upon this thesis.

1.4. Notation

For ease of reading, an overview of all definitions including the equation and page number can be found on page 67. The mathematical notation used in publications regarding reinforcement learning can differ greatly. In this thesis, we adhere to the notation introduced by Sutton and Barto (2018) and adapt definitions and proves from other sources accordingly. Consequently, interested readers may notice differences between the notation used in chapter 3 and the publications of Schulman, Moritz, Levine, Jordan, and Abbeel (2016) and Schulman et al. (2017):

- The advantage estimator is denoted δ instead of $\hat{A}$, as both A and a are heavily overloaded already.
- The likelihood ratio used in chapter 3.3 is denoted ρ_t instead of r_t to avoid confusions with rewards R_t or r. Furthermore, using ρ is consistent with Sutton and Barto's (2018) definition of *importance sampling*.

Besides that, it should be noted that an active *we* is used regularly in this thesis. This *we* is meant to include the author and all readers.

2. Reinforcement Learning

In this chapter, the fundamentals of reinforcement learning are introduced. In order to describe the components present in a reinforcement learning algorithm, we first discuss the core terms *agent* and *environment*. Then, we define a Markov decision process that contains key components we require to learn a proper policy—the function that guides an agent's behavior. Afterwards we examine two functions, the value function and the advantage function. The former allows us to set a formal goal that we strive for, whereas the latter enables us to assess the quality of an agent's choices. Finally, we reduce complexity by approximating and parameterizing the value function and the policy.

2.1. Agent and Environment

In reinforcement learning, an *agent*—which is the acting and learning entity—is embedded in and interacts with an *environment* (Sutton & Barto, 2018, chapter 3.1). The environment describes the world surrounding the agent and is beyond the agent's immediate control. However, agents can affect the environment through actions, which they choose based on the environment's observed state. After taking an action, agents observe the environment again. The interplay of agent and environment is shown in figure 1. In this thesis, we use a set of ATARI 2600 games as environments (cf. chapter 5.1).

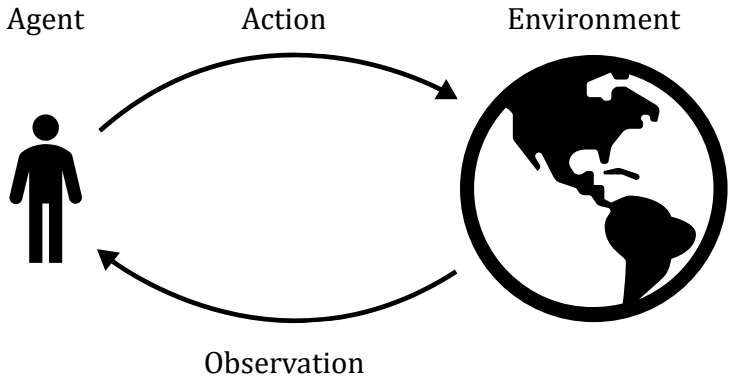

Figure 1: Agent and environment interact. The agent perceives the environment through observations and chooses an action based on the environment's state. Afterwards, the agent observes the environment again, prompting another action.

In addition to a new state, observations are associated with a reward. Agents seek to maximize the rewards they observe through trial-and-error. Hence, we can teach an

agent to achieve a certain goal provided we design rewards properly: "[...] the reward signal is your way of communicating to the [agent] *what* you want it to achieve, not *how* you want it achieved" (Sutton & Barto, 2018, p. 54). As a consequence, only the action achieving the goal must yield a positive reward. However, a sequence of actions instead of a single one may be essential to achieving the goal, raising the issue that rewards for vital actions may be delayed. Delayed reward and the trial-and-error approach are considered to be the "most important distinguishing features of reinforcement learning" by Sutton and Barto (2018, p. 2).

In order to describe agent and environment, we require a mathematical construct that encompasses the environment's states and rewards as well as the actions agents may take. These requirements are met by a Markov decision process, which we introduce in chapter 2.2. Subsequently, we define a goal the agent shall achieve utilizing the *value function*.

A key issue of reinforcement learning is the balance of *exploration* and *exploitation* (Sutton & Barto, 2018, p. 3). Often, we lack full knowledge of the environment, for example a few frames of video input from a video game do not carry information on the entire internal state of the game. We gather information by interacting with the environment, slowly learning which actions help achieving the goal. As we do not gain full information of a state or action by observing or choosing it once, we must make a decision: Do we explore more states and actions that we have no information of? Or do we exploit the best-known way to achieving the goal, gaining more knowledge on a few select states and actions?

2.2. Markov Decision Process

A Markov decision process is a stochastic process that encompasses observations, actions and rewards. These are the core properties needed for reinforcement learning.

2.2.1. Definition

We consider a finite Markov decision process with finite horizon defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \pi)$. It is comprised of three finite sets of random variables and two probability distribution functions: the set of states $\mathcal{S}$, the set of actions $\mathcal{A}$, the set of rewards $\mathcal{R}$, the dynamics function p and lastly the policy function π.[2]

[2] Definitions by other researchers may differ slightly, e.g., Schulman, Levine, Moritz, Jordan, and Abbeel (2015) do not include the policy π and add an initial state distribution. However, elements not mentioned are often included implicitly. Hence, there are no major differences in proves and subsequent definitions.

Firstly, we introduce the sets of random variables. Secondly, we examine the probability distribution functions, as they depend on the random variables. Finally, we introduce the concept of a *finite horizon* and discuss *episodes*.

In this thesis, we observe events at discrete time steps $t = 0, 1, 2, \ldots$. At each time step, agent and environment interact once—the agent chooses an action and observes the environment.

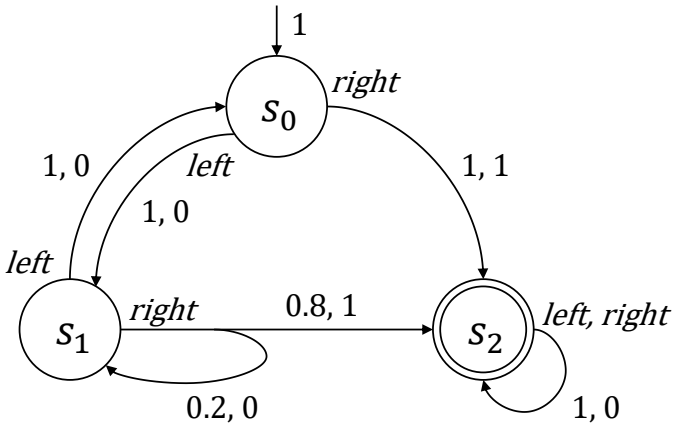

Figure 2: A simple Markov decision process with three states s_0, s_1, s_2, actions *left* and *right* as well as rewards 0 and 1.

A simple Markov decision process is displayed in figure 2. Like Markov chains, it contains states (Grinstead & Snell, 1997, chapter 11.1): s_0, s_1 and s_2 with s_0 being the initial state and s_2 being a terminal state. Unlike Markov chains, it also contains actions *left* and *right* as well as rewards 0 and 1. The rewards are written alongside transition probabilities and can be found on the edges connecting the states. Further explanations and an example using elements of this Markov decision process are given in chapter 2.2.3.

Markov decision processes extend Markov chains by introducing actions and rewards. Whereas in Markov chains a transition to a successor state depends on the current state only, in Markov decision processes the transition function p takes actions into account as well. Thus, the transition can be affected, although it remains stochastic in nature. We assume that the Markov property holds true for Markov decision processes, which means that these transitions depend on the current state and action only (Sutton & Barto, 2018, p. 49).

2.2.2. States, Actions and Rewards

States, actions and rewards are core concepts of reinforcement learning. States and actions describe an environment and an agent's capabilities within this environment. Rewards on the other hand are used to teach the agent.

States. We describe the environment using *states*: Let $S_t \in \mathcal{S}$ denote the state of the environment at time step t and let $\mathcal{S}$ denote the finite set of states (Sutton & Barto, 2018, p. 48). A state must contain all details an agent requires to make well-informed decisions.

Figure 3 displays screenshots from two ATARI 2600 games, which may be used as states. Pending several adaptation steps, we work with $\mathcal{S} \subseteq [0,1]^{4\times84\times84}$ for all ATARI 2600 games, as we use a sequence of 4 consecutive grayscale images resized to 84×84 pixels (cf. chapter 4.1).

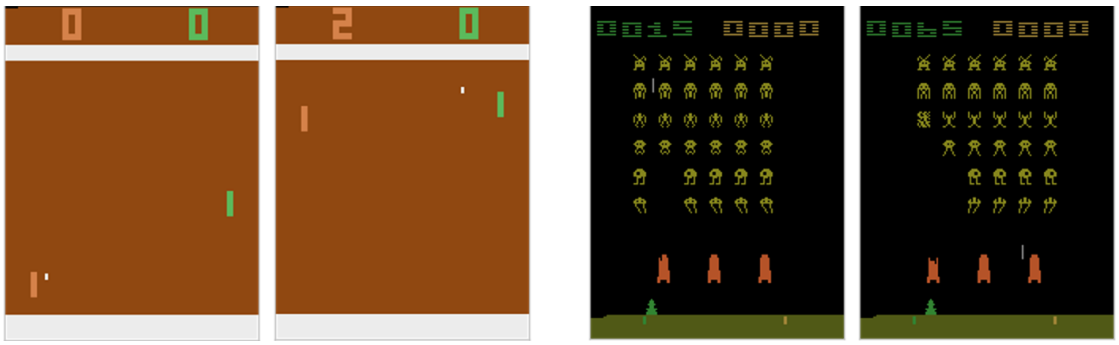

Figure 3: Pong (left) and SpaceInvaders (right) are two ATARI 2600 games. Images like these can be used as states.

Actions. At each time step t our agent takes an *action* depending on the state S_t of the environment, which causes a transition to one of several successor states. Let $A_t \in \mathcal{A}(s)$, $S_t = s$, denote the action at time step t and let $\mathcal{A}(s)$ denote the set of actions available in the state s (Sutton & Barto, 2018, p. 48).

The actions available to an agent depend on the environment and the state. An agent learning SpaceInvaders effectively chooses from $\mathcal{A}(s) = \{noop, fire, left, right\}$ for all $s \in \mathcal{S}$. *noop* means *no operation*, which is the action an agent can choose to wait a time step. If the actions do not differ between states, we write $\mathcal{A}$ to denote the set of actions for convenience.

Rewards. Every time an agent completes an action, it observes the environment. These *observations* consist of two components. The first component of an observation is the *reward*. Let $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ denote the reward observed at time step $t+1$ and let $\mathcal{R}$ denote the set of rewards (Sutton & Barto, 2018, p. 48).

Rewards are our primary means of communicating with an agent. We use them to inform an agent whether it achieved its goal or not. As stated in chapter 2.1, the agent may receive positive rewards only for achieving its goal, which raises the issue of delayed rewards. We will remedy this issue in chapter 2.3.2 by introducing the concept of a *return*.

The second component of an observation is the successor state S_{t+1}. Note that we define the reward to be R_{t+1}, as it is the reward that is observed with the state S_{t+1}. Consequently, there is no reward R_0.

In ATARI 2600 games, the reward we provide to our agent is the change in score immediately following an action, e.g., if the agent scores a point in Pong, we provide a reward $R_{t+1} = 1$ in the next observation.

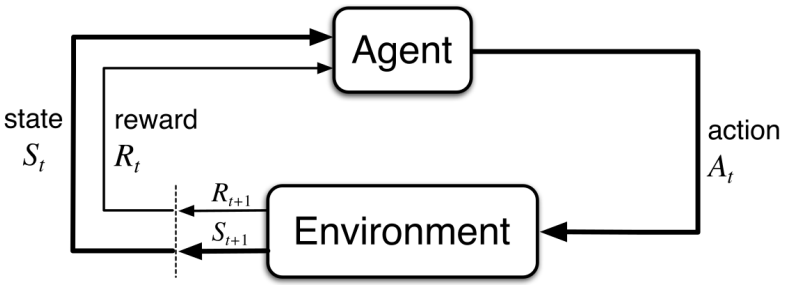

Figure 4: An agent always observes the current state S_t to decide on an action A_t. After it has acted, it observes the environment again. It obtains a reward R_{t+1} and perceives successor state S_{t+1} (Sutton & Barto, 2018).

Figure 4 shows the interaction of agent and environment in greater detail: Following each action A_t, the agent observes the change in environment represented by the reward R_{t+1} and the successor state S_{t+1}. Then, the agent acts again and obtains another observation.

2.2.3. Dynamics Function and Policy

The dynamics function p and the policy π describe the behavior of the environment and the agent. Combined, they determine the path through the respective Markov decision process.

Dynamics. Transitions to successor states and the values of rewards associated with these states are determined by the *dynamics* of the environment. Let the successor state $S_{t+1} = s'$ and the reward $R_{t+1} = r$ be an observation. Its likelihood given the current state $S_t = s$ and the agent's action $A_t = a$ is determined by the following probability distribution (Sutton & Barto, 2018, p. 48):

$$p(s', r \mid s, a) \doteq P(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a), \tag{1}$$
$$p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1].$$

We call p the dynamics function or dynamics of the environment. Since p is a probability distribution, it follows that

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1 \; \forall \; s \in \mathcal{S}, a \in \mathcal{A}(s).$$

The simple Markov decision process from chapter 2.2.1 is shown again in figure 5. Each edge describes a transition from one state to a successor state. The reward along the edge and the successor state form an observation. The probability of the observation is stated with the reward next to its respective edge. We assign a probability of 0 to observations not described by an edge. In the displayed Markov decision process, the probability of observing the state $S_{t+1} = s_2$ as well as the reward $R_{t+1} = 1$ given the current state $S_t = s_1$ and the action $A_t = \textit{right}$ is $p(s_2, 1 \mid s_1, \textit{right}) = 0.8$.

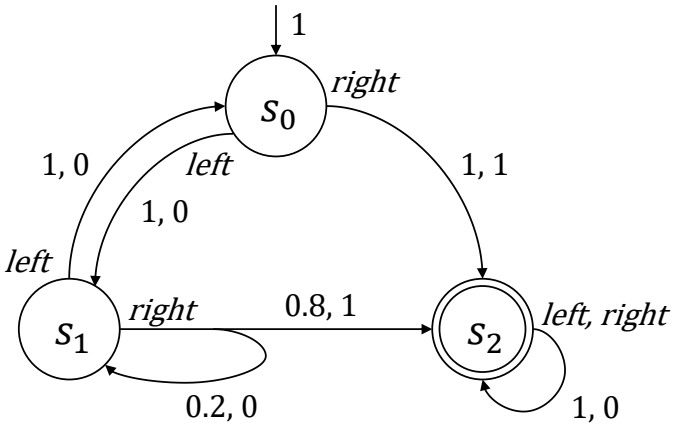

Figure 5: This Markov decision process consists of three states $\mathcal{S} = \{s_0, s_1, s_2\}$, offers two actions $\mathcal{A} = \{\textit{left}, \textit{right}\}$ and can yield two rewards $\mathcal{R} = \{0, 1\}$. Each edge is labeled with an action at the current state and a tuple consisting of the probability of the transition and the reward.

Policy. An agent's actions are determined by the *policy* function, which Sutton and Barto (2018, p. 58) define as follows:

$$\pi(a \mid s) \doteq P(A_t = a \mid S_t = s), \tag{2}$$
$$\pi : \mathcal{A} \times \mathcal{S} \to [0, 1].$$

The policy π returns the probability of the agent picking the action a in state s. We say that the agent follows the policy π. A means for learning a policy π is introduced in chapter 2.5 and elaborated on in chapter 3.

Although the policy is stochastic and does not yield actions, an agent can determine actions by sampling actions proportional to the probability distribution. For example, an agent following the policy $\pi(\textit{noop} \mid s) = \frac{2}{3},\ \pi(\textit{fire} \mid s) = \frac{1}{3}$ for all $s \in \mathcal{S}$ will choose *noop* twice as often as *fire* on average.

2.2.4. Finite Horizon and Episodes

When an agent and an environment interact with each other over a series of discrete time steps $t = 0, 1, 2, \ldots$, we can observe a sequence of states, actions and rewards $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$. We call this sequence the *trajectory* (Sutton & Barto, 2018, p. 48). When we train an agent, we often record trajectories of states, actions and rewards to use as training data by running the agent in the environment.

ATARI 2600 games have a plentitude of *terminal states* that indicate the game has come to an end, $\#\{s \mid s \in \mathcal{S} \wedge s \text{ is terminal}\} \gg 1$. For example, in Pong the game ends once either player scored 21 points, whereas the loser's score can have any value in the range of 0 to 20. Furthermore, the paddles may be in any position at the end of the game, which facilitates even more terminal states.

Whenever the agent transitions to a terminal state, it cannot transition to other states anymore. Therefore, there is a final time step T after which no meaningful information can be gained. We call T the *finite horizon*; it marks the end of an episode. Each episode begins with an initial state S_0 and ends once an agent reaches a terminal state S_T. Usually, agents attempt episodic tasks many times with each episode giving rise to a different trajectory (Sutton & Barto, 2018, p. 54); the horizon T may differ, too.

2.3. Value Function and Advantage Function

Based on the components present in a Markov decision process, we can define a formal goal an agent shall achieve. As an agent's behavior depends on the policy it follows, we

require a means to assess choices made by the policy as well. Therefore, we introduce *returns* and define two functions: The *value function* allows us to judge states, whereas the *advantage function* can be used to assess the impact of an action. These concepts are key in the learning approach and algorithm we introduce in chapters 2.5 and 3.

2.3.1. Return

For any given time step $t < T$, an agent aims to maximize the remaining sequence of rewards $R_{t+1}, R_{t+2}, \ldots, R_T$. Using this sequence we define the return

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \; 0 \leq \gamma \leq 1. \tag{3}$$

An agent strives to maximize the return G_0 and thusly all following returns G_t, too. As early returns include the reward sequence of later returns, returns may be expressed recursively: $G_t = R_{t+1} + \gamma G_{t+1}$.

γ denotes the discount factor, which discounts future rewards. In finite-horizon Markov decision processes, γ may equal 1. In infinite-horizon Markov decision processes, we must choose $\gamma < 1$ so G_t remains finite (Sutton & Barto, 2018, p. 54).

2.3.2. Value Function

Returns vary depending on the current state of the environment and on the actions an agent chooses. For example, an agent playing Pong that fails to deflect a ball will receive a negative reward, causing diminished returns. In order to maximize returns, agents must avoid actions and states that have a high likelihood of yielding negative rewards. We rate states based on the returns we may obtain by using the value function

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s], \tag{4}$$

with $\mathbb{E}_\pi$ denoting the expected value when following the policy π. The higher the value of a state, the higher the return an agent is expected to obtain. As $v_\pi(s)$ depends on the policy π, we must devise a method to optimize the policy an agent follows. However, to do so we first require the advantage function.

We call the policy that yields the maximum return the optimal policy π_* and the corresponding value function the optimal value function v_*. Assuming an optimal policy for the ATARI 2600 game Pong, $v_*(S_0) = 21$ with $\gamma = 1$, as the game ends after either player scored 21 times. Furthermore, an optimal Pong player always deflects the ball.

2.3.3. Advantage Function

The value of a state depends on the expected behavior of an agent following the policy π. In order to increase the probability of obtaining a high return, we require a means to identify actions that prove advantageous or disadvantageous in this endeavor. The required functionality is provided by the *advantage function* (Baird, 1993; Sutton, McAllester, Singh, & Mansour, 2000)[3]

$$a_\pi(s,a) \doteq \sum_{s',r} \left[p(s',r \mid s,a) \cdot (r + \gamma v_\pi(s'))\right] - v_\pi(s) \tag{5}$$

$$= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a\right] - v_\pi(s). \tag{6}$$

The advantage function compares the expected immediate reward and return of a successor state $r + v_\pi(s')$ with the expected return of the current state $v_\pi(s)$. As the environment may be non-deterministic, the former term must be weighted with its probability.

If $a_\pi(s,a) = 0$, which means $\sum_{s',r} \left[p(s',r \mid s,a) \cdot (r + \gamma v_\pi(s'))\right] = v_\pi(s)$, choosing the action a more frequently does not improve the policy. However, if $a_\pi(s,a) > 0$, the given action yields larger returns than the actions commonly chosen under π; increasing the likelihood of choosing the action a improves the policy. Actions that return a negative advantage—$a_\pi(s,a) < 0$—lead to worse results than simply following π would. If $a_\pi(s,a) \leq 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}$, the policy is optimal.

According to Schulman et al. (2016), a common approximation of the advantage function is provided by the estimator

$$\delta_t \doteq R_{t+1} + \gamma v_\pi(S_{t+1}) - v_\pi(S_t). \tag{7}$$

The advantage function and estimator enable us to assess whether choosing a certain action over the policy's expected behavior results in improved performance. Hence, we can improve the agent's behavior by increasing the probabilities of actions with positive advantages. We will do so by combining the advantage function with policy gradients (cf. chapter 2.5).

[3]The advantage function is commonly defined using the q function: $a_\pi(s,a) \doteq q_\pi(s,a) - v_\pi(s)$. The definition we provide is equal, as it merely substitutes $q_\pi(s,a)$ with the term it is defined to be.

2.4. Function Approximation

In simple environments with a small state space, we may construct a tabular solution storing values for all states we encounter. Among the algorithms creating such tables are *SARSA* and *Q Learning* (Sutton & Barto, 2018, chapters 6.4 and 6.5). If the state space grows very large, creating a table becomes infeasible, both in terms of computation time and memory consumption. Hence, we must approximate both the value function and the policy (cf. chapter 2.5).

Let $\hat{v}_\pi(s, \boldsymbol{\omega})$ be an estimator of the value function $v_\pi(s)$, that means $\hat{v}_\pi(s, \boldsymbol{\omega}) \approx v_\pi(s)$ for all $s \in \mathcal{S}$. Instead of utilizing a table, we utilize a function parameterized with $\boldsymbol{\omega} \in \mathbb{R}^d$. As the state space may be very large, we demand that $d \ll \#\mathcal{S}$. Various approaches can be used to design $\boldsymbol{\omega}$. In this thesis, we use a neural network to parameterize $\hat{v}_\pi$ with $\boldsymbol{\omega}$ being the neural network's weights (see chapters 3 and 4.2).

Applying function approximation introduces inaccuracy. Because $d \ll \#\mathcal{S}$, adjusting a single weight of the weight vector $\boldsymbol{\omega}$ changes the value estimations of multiple states. Furthermore, changing the value estimation of a single state causes changes in the value estimations of other states, too. Thus, increasing the accuracy of some states will decrease the accuracy of other states.

In order to alleviate this issue, the error introduced through estimation can be weighted according to the likelihood a state is observed. Let μ denote the stationary distribution of states, then $\mu_\pi(s)$ is the percentage of time an agent following π spends in the state s. $\hat{v}_\pi(s, \boldsymbol{\omega})$ shall be chosen such that the mean squared value error

$$\overline{\mathrm{VE}}(\boldsymbol{\omega}) \doteq \sum_{s \in \mathcal{S}} \mu_\pi(s)[v_\pi(s) - \hat{v}_\pi(s, \boldsymbol{\omega})]^2 \tag{8}$$

is minimized (Sutton & Barto, 2018, p. 199). As we sample trajectories to form estimations from, the states we observe naturally are proportional to the stationary distribution μ. As a consequence, we update the value estimations of states we observe regularly more frequently leading to more accurate estimations. Therefore, we minimize $\overline{\mathrm{VE}}(\boldsymbol{\omega})$ implicitly. We establish a method to learn the value function in chapter 3.3.3.

We can easily extend function approximation to the advantage function $a_\pi(s, a)$ by utilizing $\hat{v}_\pi$. In equation 5, the advantage function was defined to be

$$\begin{aligned} a_\pi(s, a) &\doteq \sum_{s', r} \left[p(s', r \mid s, a) \cdot \left(r + \gamma v_\pi(s')\right)\right] - v_\pi(s) \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a\right] - v_\pi(s). \end{aligned}$$

Then, the advantage function is parameterized by inserting the parameterized value function $\hat{v}_\pi$ instead of v_π:

$$\hat{a}_\pi(s, a, \boldsymbol{\omega}) \doteq \sum_{s',r} \left[p(s', r \mid s, a) \cdot (r + \gamma \hat{v}_\pi(s', \boldsymbol{\omega}))\right] - \hat{v}_\pi(s, \boldsymbol{\omega}) \tag{9}$$

$$= \mathbb{E}_\pi \left[R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}, \boldsymbol{\omega}) \mid S_t = s, A_t = a\right] - \hat{v}_\pi(s, \boldsymbol{\omega}). \tag{10}$$

We further extend function approximation to the advantage estimator

$$\delta_t \doteq R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}, \boldsymbol{\omega}) - \hat{v}_\pi(S_t, \boldsymbol{\omega}). \tag{11}$$

Note that we do not indicate whether the advantage estimator uses the value function v_π (like in equation 7) or a parameterized value function $\hat{v}_\pi$. In subsequent chapters we consistently use function approximation, therefore it can be assumed that advantage estimators rely on parameterized value functions.

2.5. Policy Gradients

Policy gradients are one method of learning a policy. The main idea is parameterizing the policy with a derivable function $\boldsymbol{\theta}$. The policy is then learned by optimizing $\boldsymbol{\theta}$ with gradient ascent.

2.5.1. Policy Parameterization

Just like the value function, the policy too can be parameterized when the state space grows too large. Let $\boldsymbol{\theta} \in \mathbb{R}^{d'},\ d' \ll \#\mathcal{S}$, denote the policy's parameter vector. Then the parameterized policy

$$\pi(a \mid s, \boldsymbol{\theta}) \doteq P(A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}) \tag{12}$$

returns the probability of taking the action a given the state s and parameter vector $\boldsymbol{\theta}$ (Sutton & Barto, 2018, p. 321). Often, we write $\pi_{\boldsymbol{\theta}}$ instead of $\pi(a \mid s, \boldsymbol{\theta})$.

Usually, $\boldsymbol{\theta}$ is the parameterization of a function that maps a state to a vector of numerical weights. Each action corresponds to a weight that indicates the likelihood of picking the respective action. The higher the weight, the more likely picking its respective action should be. The policy π uses the numerical weights to determine a probability distribution (Sutton & Barto, 2018, p. 322), e.g., by applying a softmax function (Goodfellow, Bengio, & Courville, 2016, pp. 184–185). The method used to determine the probabil-

ity distribution remains fixed. Instead we optimize the parameterization $\boldsymbol{\theta}$ so that the likelihood of choosing advantageous actions is increased.

Let $S_t = s$ and $\mathcal{A}(s) = \{1, 2\}$. A function h parameterized with $\boldsymbol{\theta}_t$ could return a vector with two elements, for example $h(s, \boldsymbol{\theta}_t) = \begin{pmatrix} 4 \\ 2 \end{pmatrix}$. The elements of this vector are numerical weights a policy could use to determine probabilities:[4]

$$\pi(a \mid s, \boldsymbol{\theta}_t) = (h(s, \boldsymbol{\theta}_t))_a \cdot \left(\sum_{b \in \mathcal{A}(s)} (h(s, \boldsymbol{\theta}_t))_b \right)^{-1}, \text{ e.g.,}$$

$$\pi(1 \mid s, \boldsymbol{\theta}_t) = \begin{pmatrix} 4 \\ 2 \end{pmatrix}_1 \cdot \left(\begin{pmatrix} 4 \\ 2 \end{pmatrix}_1 + \begin{pmatrix} 4 \\ 2 \end{pmatrix}_2 \right)^{-1} = \frac{4}{4+2} = \frac{2}{3}.$$

Parameterizing the policy allows us to employ policy gradient methods. We transform the issue of action selection into an optimization problem. This problem can be solved by calculating the gradient, which is a vector of partial derivatives. With full information of the environment, we may finally search for maxima to find an approximately optimal policy.

2.5.2. Gradient Ascent

Policy gradient algorithms seek to optimize the policy directly by performing stochastic gradient ascent $\nabla_{\boldsymbol{\theta}} \pi(a \mid s, \boldsymbol{\theta})$ to maximize a fitness function $J(\boldsymbol{\theta})$. As we lack full knowledge of the environment's dynamics, we cannot compute accurate gradients but have to resort to gradient estimators $\hat{g}$:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \cdot \hat{g},$$
$$\hat{g} \approx \nabla J(\boldsymbol{\theta}).$$

This approach matches *stochastic gradient descent* as commonly used in machine learning (Goodfellow et al., 2016, pp. 151–153): The accurate gradient is estimated using a set of samples—we record a trajectory. The learn rate α controls the step size along the gradient, as we do not desire to optimize $\boldsymbol{\theta}$ for one gradient estimation only. Instead, we perform stochastic gradient ascent many times, performing a step along the gradient. By performing many small steps along different gradients, we strike a balance between different estimations that are calculated from different trajectories.

[4] We write $(h(s, \boldsymbol{\theta}_t))_n$ to obtain the n-th component of $h(s, \boldsymbol{\theta}_t)$, e.g., $\begin{pmatrix} 4 \\ 2 \end{pmatrix}_1 = 4$.

In contrast to common stochastic gradient descent, we do not intend to minimize an error. Rather, we maximize the value function when following $\pi_{\boldsymbol{\theta}}$ and thusly define the fitness function to be

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(S_0). \tag{13}$$

An example of a gradient estimator can be seen in equation 16 at the end of this chapter. In order to perform gradient ascent, $\pi(a \mid s, \boldsymbol{\theta})$ must be differentiable with respect to $\boldsymbol{\theta}$. We utilize neural networks as a differentiable function (cf. chapter 4.2).

According to Sutton and Barto (2018, pp. 322–323), policy gradient methods provide two major advantages over other methods. Firstly, parameterized policies can approach a deterministic policy gradually. Ideally, we must only assign a reasonable value to the learn rate α, but no further hyperparameters are required to ensure sufficient exploration and a smooth transition to exploitation.[5] Secondly, parameterized policies enable us to assign arbitrary probabilities to actions. This gives us the opportunity to discover stochastic approximates of optimal policies.[6] Sutton and Barto (2018, p. 323) give an example of a Markov decision process that can only be solved by a stochastic policy.

Sutton and Barto (2018, p. 324) point out that combining a parameterized policy with function approximation may pose a challenge: The performances of both $\pi_{\boldsymbol{\theta}}$ and $\hat{v}(s, \boldsymbol{\omega})$ depend on the action selections and on the distribution of the states $\mu_{\pi_{\boldsymbol{\theta}}}(s)$ that these actions are selected in. Adjusting the policy results in different action choices, which in turn changes $\mu_{\pi_{\boldsymbol{\theta}}}(s)$. Thus, we might assume that we require the derivative of $\mu_{\pi_{\boldsymbol{\theta}}}(s)$ to compute gradient estimates. This issue is remedied by the *policy gradient theorem*, which proves that

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu_{\pi_{\boldsymbol{\theta}}}(s) \sum_a q_{\pi_{\boldsymbol{\theta}}}(s, a) \nabla_{\boldsymbol{\theta}} \pi(a \mid s, \boldsymbol{\theta}), \tag{14}$$

with $\propto$ meaning "proportional to". These gradients may not share the same magnitude, but they share the same direction. Accordingly, we must only adjust the step size α that we perform stochastic gradient ascent with.

Originally, this proof held true only with $q_{\pi_{\boldsymbol{\theta}}}(s, a)$, a function that returns the value of an action a given the state s. However, the policy gradient theorem was expanded

[5] In practice other parameters are required to prevent forgetting and performance collapses (cf. chapter 3.1).

[6] A policy that is not capable of doing so is the ε-greedy policy, which chooses the best available action with a probability of $1 - \varepsilon$ and a random one otherwise (Sutton & Barto, 2018, p. 322). ε typically converges to 0 over the course of the training resulting in a deterministic policy.

upon to allow the combination of policy gradients with the advantage function (Sutton et al., 2000):

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu_{\pi_{\boldsymbol{\theta}}}(s) \sum_a a_{\pi_{\boldsymbol{\theta}}}(s, a, \boldsymbol{\omega}) \nabla_{\boldsymbol{\theta}} \pi(a \mid s, \boldsymbol{\theta}). \tag{15}$$

Using the advantage function is beneficial, as the policy gradient in equation 15 has a lower variance than the gradient in equation 14 (Schulman et al., 2016).

Like the mean squared value error $\overline{\text{VE}}$ in equation 8, the gradient is weighted according to the distribution of states μ, but it does not depend on its derivative. By virtue of weighting with μ, states an agent encounters regularly have a greater effect on the gradient. Moreover, the magnitude of the gradient is controlled by the advantage of an action: A large advantage mandates a large gradient step.

2.5.3. Gradient Estimation

Although $\mu_{\pi_{\boldsymbol{\theta}}}(s)$ is unknown, we can estimate the above gradient by recording trajectories. Naturally, the states encountered when following a policy $\pi_{\boldsymbol{\theta}}$ should match the distribution of states under this policy. This gives us the gradient estimator

$$\hat{g} \doteq \mathbb{E}_{a,s\sim\pi_{\boldsymbol{\theta}}} \left[\hat{a}_{\pi_{\boldsymbol{\theta}}}(s, a, \boldsymbol{\omega}) \nabla_{\boldsymbol{\theta}} \pi(a \mid s, \boldsymbol{\theta}) \right], \tag{16}$$

with $a, s \sim \pi_{\boldsymbol{\theta}}$ indicating that actions and states are taken from a trajectory recorded by an agent following $\pi_{\boldsymbol{\theta}}$. That means we sample actions from $\pi_{\boldsymbol{\theta}}$ and states are determined by the dynamics function p.

We could use this gradient estimator to train an agent already. However, it suffers from some disadvantages that complicate finding a good or optimal policy. We address these disadvantages in chapter 3.

3. Proximal Policy Optimization

Deep learning methods achieved success on a variety of tasks, such as computer vision and speech recognition (Goodfellow et al., 2016, chapter 1.2.4). For this reason, Mnih et al. (2013) combined reinforcement learning techniques and deep learning methods in an algorithm called DQN. The benefits of using neural networks to parameterize the value function and—occasionally—the policy were demonstrated by DQN and further *deep reinforcement learning* algorithms, e.g., by A3C (Mnih et al., 2016) and Rainbow DQN (Hessel et al., 2017).

The algorithm introduced in this chapter is called *Proximal Policy Optimization* (PPO) (Schulman et al., 2017). Because PPO is a deep reinforcement learning algorithm, we adjust notation and replace our gradient estimator $\hat{g}$. Instead we utilize a loss $\mathcal{L}$, as is common in deep learning (Goodfellow et al., 2016, chapter 4.3):

$$\mathcal{L}(\boldsymbol{\theta}) \doteq \mathbb{E}_{a,s\sim\pi_{\boldsymbol{\theta}}} \left[\hat{a}_{\pi_{\boldsymbol{\theta}}}(s, a, \boldsymbol{\omega})\pi_{\boldsymbol{\theta}}(a \mid s)\right] \tag{17}$$

The gradient estimator $\hat{g}$ can be obtained by deriving the loss in equation 17. Unlike in deep learning, a loss notation is also used to denote objectives that shall be maximized rather than minimized in deep reinforcement learning. Therefore, it depends on the specific objective whether gradient ascent or gradient descent is performed.

We begin by highlighting issues with the gradient estimator from equation 16 and motivate the use of advanced algorithms such as Proximal Policy Optimization. Then we introduce sophisticated advantage and return estimators as used in many modern policy gradient methods. Afterwards the loss is introduced and explained. We close by providing the complete Proximal Policy Optimization algorithm.

3.1. Motivation

Policy gradient estimators such as the one introduced in equation 16 in chapter 2.5 or REINFORCE[7] by Williams (1992) suffer from a significant drawback. As Kakade and Langford (2002) demonstrate, some tasks require that we record long trajectories to guarantee a policy is improved when performing stochastic gradient ascent. The longer the trajectories are the higher their variance grows, as both the dynamics p of the environment and the policy π introduce non-deterministic behavior.

[7] A well-known reinforcement learning algorithm that is a simple improvement over the estimator we introduced.

As the variance of the trajectories grows, the quality of the gradient estimator declines. Performing stochastic gradient ascent with this estimator is no longer guaranteed to yield an improved policy—in fact, the performance of the policy might deteriorate (Seita, 2017). Even a single bad step might cause a collapse of policy performance (OpenAI Inc., 2018). More importantly, the performance collapse might be irrecoverable. As the policy was likely trained for several steps already, it has begun transitioning from exploration to exploitation. If an agent now strongly favors bad actions, it cannot progress towards the goal. In turn, we cannot sample meaningful data to train the policy further.

Mnih et al. (2013) further note that deep learning algorithms assume independent data samples. This poses another challenge, as trajectories are commonly sequences of highly correlated states, actions and rewards.

Proximal Policy Optimization addresses these challenges by combining multiple known concepts. In order to obtain less correlated data, multiple environments are executed simultaneously—the agent uses one *actor* in each environment. Furthermore, policy updates are restricted so the new policy does not deviate far from the old one. Instead of performing only one optimization pass on recorded data, multiple passes are performed so agents learn faster.

3.2. Advantage and Value Estimation

Both the loss described in equation 17 and the loss proposed by Schulman et al. (2017) depend on advantage estimations. However, to estimate advantages value estimations are required. By definition (cf. equation 4), the value function can be estimated using the return. Figure 6 highlights these relations.

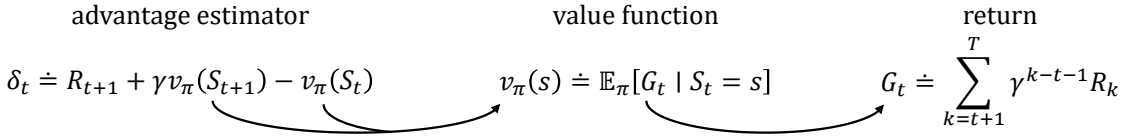

Figure 6: The advantage estimations required to calculate a loss or gradient estimator depend on the value function. The value function in turn requires returns. Finally, these estimators and functions build upon rewards R_{t+1}.

Both the advantage estimator (cf. equation 11) and the return (cf. equation 3) are suboptimal, as they suffer from being biased and having high variance, respectively. We further explain these issues and introduce advanced methods that alleviate this issue in this chapter.

3.2.1. Generalized Advantage Estimation

In chapter 2.3.3 we introduced the advantage function $a_\pi(s, a)$ and an estimator δ_t that we combined with function approximation (cf. equations 9 and 11 in chapter 2.4):

$$\begin{aligned}\hat{a}_\pi(s, a, \boldsymbol{\omega}) &\doteq \mathbb{E}_\pi \left[R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}, \boldsymbol{\omega}) \mid S_t = s, A_t = a\right] - \hat{v}_\pi(s, \boldsymbol{\omega}),\\ \delta_t &\doteq R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}, \boldsymbol{\omega}) - \hat{v}_\pi(S_t, \boldsymbol{\omega}).\end{aligned}$$

δ_t comes with a considerable disadvantage as it is biased. In the above definitions there are two instances of a value estimation $\hat{v}_\pi$ present, which are by definition inaccurate. Consequently, the advantage estimator is inaccurate as well.

The term $-\hat{v}_\pi(S_t, \boldsymbol{\omega})$ is necessary by definition of the advantage function, but we can replace the γ-discounted value estimation with a longer sequence of rewards. For example, we could replace $\gamma \hat{v}_\pi(S_{t+1}, \boldsymbol{\omega})$ with $\gamma R_{t+2} + \gamma^2 \hat{v}_\pi(S_{t+2}, \boldsymbol{\omega})$.

Let $\delta_t^{(i)}$ denote the advantage estimator that contains i rewards. Then

$$\begin{aligned}
\delta_t^{(1)} &\doteq \delta_t && = R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}, \boldsymbol{\omega}) - \hat{v}_\pi(S_t, \boldsymbol{\omega}) && (18)\\
\delta_t^{(2)} &\doteq \delta_t + \gamma \delta_{t+1} && = R_{t+1} + \gamma R_{t+2} + \gamma^2 \hat{v}_\pi(S_{t+2}, \boldsymbol{\omega}) - \hat{v}_\pi(S_t, \boldsymbol{\omega})\\
\delta_t^{(3)} &\doteq \delta_t + \gamma \delta_{t+1} + \gamma^2 \delta_{t+2} && = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 \hat{v}_\pi(S_{t+3}, \boldsymbol{\omega}) - \hat{v}_\pi(S_t, \boldsymbol{\omega})\\
&\vdots\\
\delta_t^{(i)} &\doteq \sum_{k=0}^{i-1} \gamma^l \delta_{t+k} && && (19)\\
&= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{i-1} R_{t+i} + \gamma^i \hat{v}_\pi(S_{t+i}, \boldsymbol{\omega}) - \hat{v}_\pi(S_t, \boldsymbol{\omega})
\end{aligned}$$

as proposed by Schulman et al. (2016).

The more rewards are added, the more the value estimation $\hat{v}_\pi(S_{t+i}, \boldsymbol{\omega})$ is discounted, because $\delta_t^{(i)}$ contains a γ^i-discounted value estimation and $\gamma < 1$. Consequently, the inaccuracy contributed by this term is reduced, yielding a less biased advantage estimator. However, as i is increased, the number of stochastic elements in the calculation grows. This increases the variance of the estimator. Thus, $\delta_t^{(1)}$ has the highest bias and the lowest variance, whereas $\delta_t^{(T-t)}$ possesses the lowest bias and the highest variance. Schulman et al. (2016) found that no particular advantage estimator $\delta_t^{(i)}$ grants the best results—instead, they must be combined.

Let $\lambda \in [0, 1)$ denote a variance tuning factor.[8] Then *Generalized Advantage Estimation* (GAE) is defined to be

$$\begin{aligned}
\delta_t^{\mathrm{GAE}(\gamma,\lambda)} &\doteq (1-\lambda)\left(\delta_t^{(1)} + \lambda\delta_t^{(2)} + \lambda^2\delta_t^{(3)} + \dots\right) && (20)\\
&= (1-\lambda)\left(\delta_t + \lambda(\delta_t + \gamma\delta_{t+1}) + \lambda^2(\delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2}) + \dots\right)\\
&= (1-\lambda)\Big(\delta_t(1 + \lambda + \lambda^2 + \dots) + \gamma\delta_{t+1}(\lambda + \lambda^2 + \lambda^3 + \dots)\\
&\qquad +\gamma^2\delta_{t+2}(\lambda^2 + \lambda^3 + \lambda^4 + \dots) + \dots\Big)\\
&= (1-\lambda)\left(\delta_t\left(\frac{1}{1-\lambda}\right) + \gamma\delta_{t+1}\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2\delta_{t+2}\left(\frac{\lambda^2}{1-\lambda}\right) + \dots\right)\\
&= \sum_{k=0}^{\infty}(\gamma\lambda)^k\delta_{t+k}. && (21)
\end{aligned}$$

The first line of the definition is merely a λ-discounted sum of $\delta_t^{(i)}$ advantage estimations with $(1-\lambda)$ normalizing the sum. In the third step, the sum $1 + \lambda + \lambda^2 + \dots$ is replaced with the result of the geometric series; the same can be achieved for sums $\lambda + \lambda^2 + \lambda^3 + \dots$ by factoring λ first.

λ allows us to control the impact of high-bias terms versus high-variance terms. If we choose $\lambda = 0$, then the bias is the highest, as only δ_t is used for advantage estimation. As λ approaches 1, the bias decreases whereas the variance of the estimator grows.

Note that Generalized Advantage Estimation is defined for infinite-horizon Markov decision processes. However, GAE is commonly used with finite-horizon Markov decision processes, too. In the above definition we replace ∞ with T at the cost of adding minor inaccuracy. The inaccuracy grows larger as the time step t approaches the horizon T.

3.2.2. λ-return

The same approach can be applied to return estimation. Remember that we defined the return to be

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1}R_k,$$

which has high variance.

Let $G_t^{(n)}$ denote the return that sums n γ-discounted rewards and then adds a discounted value estimation. Then $G_t^{(T-t)} \doteq G_t$. Instead of summing rewards, we may add

[8] Note that Schulman et al. (2016) allow $\lambda = 1$, but we do not require this special case.

a value estimation. No value estimation has to be added to $G_t^{(T-t)}$, as the value of the terminal state $S_T = 0$. We define

$$\begin{aligned} G_t^{(1)} &\doteq R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}, \boldsymbol{\omega}) \\ G_t^{(2)} &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 \hat{v}_\pi(S_{t+2}, \boldsymbol{\omega}) \\ &\vdots \end{aligned}$$

These returns are called *n-step returns*. They approximate the return G_t (Sutton & Barto, 2018, p. 143). Instead of choosing a specific n, we combine many n-step returns discounted by $\lambda \in [0, 1]$. We call this return the *λ-return* (Sutton & Barto, 2018, chapter 12.1):

$$G_t^\lambda \doteq (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}. \tag{22}$$

Again, we utilize $(1 - \lambda)$ to normalize the sum. As above, if we choose $\lambda = 0$, the return approximation has a high bias. For λ approaching 1 the return approximation has high variance.

Generalized Advantage Estimation can be expressed in terms of the λ-return (Peng, Abbeel, Levine, & van de Panne, 2018):

$$\delta_t^{\mathrm{GAE}(\gamma,\lambda)} = G_t^\lambda - \hat{v}_\pi(S_t, \boldsymbol{\omega}).$$

3.3. Loss

Proximal Policy Optimization consists of two components: A loss and an algorithm that incorporates this loss. In the following sections we introduce and explain the clipped Proximal Policy Optimization loss. Afterwards, we reveal the learning algorithm that uses this loss to optimize a policy.

3.3.1. Background

Since policy gradients are prone to collapse, Kakade and Langford (2002) introduce a method to construct an improved policy, which is called *Conservative Policy Iteration*. With full information of the environment's dynamics p, they derive a lower bound on the new policy's performance and provide confidences for the success of their algorithm.

However, this algorithm can only be applied to mixed policies of the form

$$\alpha \cdot \pi^{(0)} + (1 - \alpha) \cdot \pi^{(1)},$$

where $\pi^{(0)}$ and $\pi^{(1)}$ are different policies and $\alpha < 1$.

Schulman et al. (2015) prove that this method can be applied to stochastic policies—as defined in equation 2—by changing the lower bound to a *Kullback-Leibler* (KL) *divergence*. The KL divergence can be used to assess how much two probability distributions deviate from one another (Goodfellow et al., 2016, pp. 74–75). They further transform the loss to a likelihood ratio of the candidate for the new policy and the current policy (similar to equation 23) that is penalized using the KL divergence. This way they incentivize finding a new policy that is close to the old policy.

As the new loss still imposes a lower bound on the fitness $J(\boldsymbol{\theta})$, it is a minorant to the fitness function. The authors prove that maximizing this minorant guarantees a monotonously rising fitness $J(\boldsymbol{\theta})$; given sufficient optimization steps the algorithm converges to a local optimum.[9] The ensuing objective is called a *surrogate* objective.

The mathematically proven algorithm is computationally demanding, as it needs to compute the KL divergence on all states in the state space for each optimization step. Hence, Schulman et al. (2015) perform multiple approximations. The resulting algorithm is called *Trust Region Policy Optimization* (TRPO).

However, TRPO achieves suboptimal results on tasks like video games and is both complicated to realize as well as to execute (Schulman et al., 2017). In order to compute an approximately optimal solution of the KL divergence, the inverse of a large Hessian matrix[10] is required. This leads Schulman et al. (2017) to devise a simpler algorithm called *Proximal Policy Optimization* (PPO). We discuss this algorithm in the following chapters.

3.3.2. Clipped Surrogate Objective

Proximal Policy Optimization is a policy gradient method that uses advantage estimation, e.g., Generalized Advantage Estimation, to estimate the gradient. PPO performs multiple optimization steps on the same trajectory. The policy that recorded the trajectory is denoted $\pi_{\boldsymbol{\theta}_{\text{old}}}$. With each optimization step, $\boldsymbol{\theta}$ moves further from $\boldsymbol{\theta}_{\text{old}}$.

[9] Algorithms like this are called MM algorithms. This one is a *minorize maximization* algorithm (Hunter & Lange, 2004).

[10] The Hessian matrix is a second-order derivative. It can be determined by deriving the gradient of a function (Goodfellow et al., 2016, pp. 86–87).

As a consequence, an action that has become very unlikely under $\pi_{\boldsymbol{\theta}}$ might have a large advantage. Since the probability of sampling this action when following $\pi_{\boldsymbol{\theta}}$ instead of $\pi_{\boldsymbol{\theta}_{\text{old}}}$ is a lot smaller, we must weight the advantage accordingly. If we do not weight the advantage, an unlikely action would have a large effect on the gradient. The same holds true for actions with an increased likelihood that are given small advantages.

A commonly employed technique is *importance sampling* (Sutton & Barto, 2018, pp. 103–104). We weight advantages using the likelihood ratio

$$\rho_t(\boldsymbol{\theta}) \doteq \frac{\pi_{\boldsymbol{\theta}}(A_t \mid S_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(A_t \mid S_t)}, \tag{23}$$

where $\pi_{\boldsymbol{\theta}}$ is the current policy and $\pi_{\boldsymbol{\theta}_{\text{old}}}$ is a previous policy (Schulman et al., 2017).

The action and the state were sampled by an agent following $\pi_{\boldsymbol{\theta}_{\text{old}}}$. By evaluating the ratio $\rho_t(\boldsymbol{\theta})$, we determine if the probability of taking the action a in the state s has increased or decreased. For example, if $\rho_t(\boldsymbol{\theta}) > 1$, the action is more likely under $\pi_{\boldsymbol{\theta}}$ than it is under $\pi_{\boldsymbol{\theta}_{\text{old}}}$. We note that $\rho_t(\boldsymbol{\theta}_{\text{old}}) = 1$.

A loss derived from *Conservative Policy Iteration* is constructed by multiplying the likelihood ratio and advantage estimations (Schulman et al., 2017):

$$\mathcal{L}^{\text{CPI}}(\boldsymbol{\theta}) \doteq \mathbb{E}_{a,s\sim\pi_{\boldsymbol{\theta}_{\text{old}}}}\left[\frac{\pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a \mid s)} \cdot \hat{a}_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s, a, \boldsymbol{\omega})\right]. \tag{24}$$

Despite the use of importance sampling this loss can be unreliable. For actions with a very large likelihood ratio, for example, $\rho_t(\boldsymbol{\theta}) = 100$, gradient steps become excessively large possibly leading to performance collapses (Kakade & Langford, 2002).

Hence, Proximal Policy Optimization algorithms intend to keep the likelihood ratio close to one (Schulman et al., 2017). There are two variants of PPO, one that relies on clipping the ratio, whereas the other is penalized using the Kullback-Leibler divergence. We examine the clipped variant, as it is more widely used and achieved better results on various tasks such as robotics and video games (Schulman et al., 2017; OpenAI Inc., 2017b; Kostrikov, 2018).

The probability ratio $\rho_t(\boldsymbol{\theta})$ is clipped using the following function:

$$\text{clip}(\rho_t(\boldsymbol{\theta}), \epsilon) \doteq \begin{cases} (1-\epsilon) & \text{if } \rho_t(\boldsymbol{\theta}) \leq 1-\epsilon, \\ (1+\epsilon) & \text{if } \rho_t(\boldsymbol{\theta}) \geq 1+\epsilon, \\ \rho_t(\boldsymbol{\theta}) & \text{otherwise.} \end{cases} \tag{25}$$

ϵ is a hyperparameter and $[1-\epsilon, 1+\epsilon]$ is called the clip range. With clipping, the likelihood ratio becomes a constant if $\boldsymbol{\theta}$ moves too far from $\boldsymbol{\theta}_{\text{old}}$. If we apply clipping to the loss defined in equation 24, $\mathcal{L}^{\text{CPI}}$ becomes constant if $\boldsymbol{\theta}$ moves too far from $\boldsymbol{\theta}_{\text{old}}$. Therefore, the gradient of $\mathcal{L}^{\text{CPI}}$ is 0 outside of the clip range. Figure 7 shows the behavior of $\text{clip}(\rho_t(\boldsymbol{\theta}), \epsilon) \cdot \delta$, with δ being a single advantage estimation.

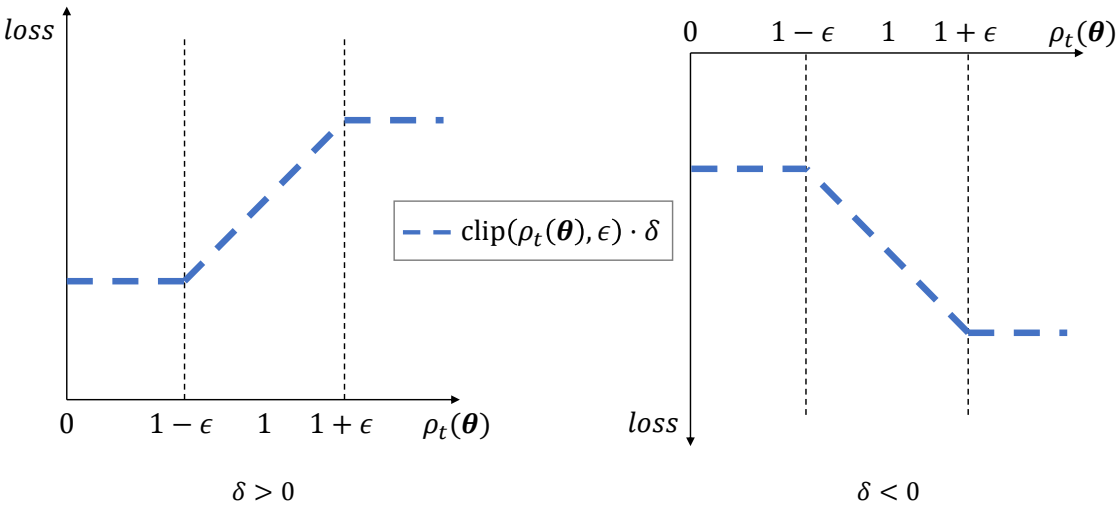

Figure 7: The left side of this figure shows the loss for an action with a positive advantage, whereas the loss for an action with negative advantage is shown on the right side. The loss is clipped if the likelihood ratio (cf. equation 23) moves outside of the clip range $[1-\epsilon, 1+\epsilon]$. Figure inspired by Schulman et al.'s (2017) and Wang et al.'s (2019) publications.

Due to clipping the likelihood ratio, no further gradient steps are taken when the probability of taking an action with positive advantage, $\delta > 0$, was increased past $\rho_t(\boldsymbol{\theta}) \geq 1+\epsilon$. The same holds true for actions with negative advantage, $\delta < 0$, if their respective probability was decreased past $\rho_t(\boldsymbol{\theta}) \leq 1-\epsilon$.

Simply clipping at both ends of the clip range raises an issue. If we decrease the likelihood of an action with positive advantage too much, the gradient will become 0. That means we cannot correct this mistake, as we cannot perform a gradient step. This also happens when we raise the likelihood of actions with negative advantages.

Consider the following example: $\pi_{\boldsymbol{\theta}_{\text{old}}}(noop \mid s) = 0.5$ given the state s. Although the advantage estimation is positive, $\delta = 1$, the probability was wrongly decreased—$\pi_{\boldsymbol{\theta}}(noop \mid s) = 0.25$. If ϵ is small, for example $\epsilon = 0.1$, $\rho_t(\boldsymbol{\theta}) = 0.5$ with $A_t = noop$ and $S_t = s$ is smaller than $1-\epsilon = 0.9$. As the ratio moved outside of the clip range, it is

clipped and the gradient becomes 0. This means that we cannot correct the error we made in decreasing the likelihood of picking the action.

We solve this issue by taking an elementwise minimum. Then, $\mathcal{L}^{\text{CLIP}}$ denotes the clipped surrogate objective (Schulman et al., 2017)

$$\mathcal{L}^{\text{CLIP}}(\boldsymbol{\theta}) \doteq \mathbb{E}_{a,s\sim\pi_{\boldsymbol{\theta}_{\text{old}}}}\left[\min\left(\rho_t(\boldsymbol{\theta})\cdot\hat{a}_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s,a,\boldsymbol{\omega}),\ \text{clip}(\rho_t(\boldsymbol{\theta}),\epsilon)\cdot\hat{a}_{\pi_{\boldsymbol{\theta}_{\text{old}}}}(s,a,\boldsymbol{\omega})\right)\right]. \quad (26)$$

In practice, the advantage function $\hat{a}_{\pi_{\boldsymbol{\theta}_{\text{old}}}}$ is replaced with Generalized Advantage Estimations $\delta_t^{\text{GAE}(\gamma,\lambda)}$ as defined in equation 21 (Schulman et al., 2017). $\mathcal{L}^{\text{CLIP}}(\boldsymbol{\theta})$ is called a surrogate objective, as it approximates the original objective $J(\boldsymbol{\theta})$.

Figure 8 compares $\text{clip}(\rho_t(\boldsymbol{\theta},\epsilon)\cdot\delta$ and $\mathcal{L}^{\text{CLIP}}(\boldsymbol{\theta})$. Using an elementwise minimum has the following effect:

- If the advantage estimation $\delta > 0$ and $\rho_t(\boldsymbol{\theta}) \leq 1-\epsilon$, the loss is not clipped and the likelihood of the corresponding action can be increased.
- If the advantage estimation $\delta < 0$ and $\rho_t(\boldsymbol{\theta}) \geq 1+\epsilon$, the loss is not clipped and the likelihood of the corresponding action can be decreased.

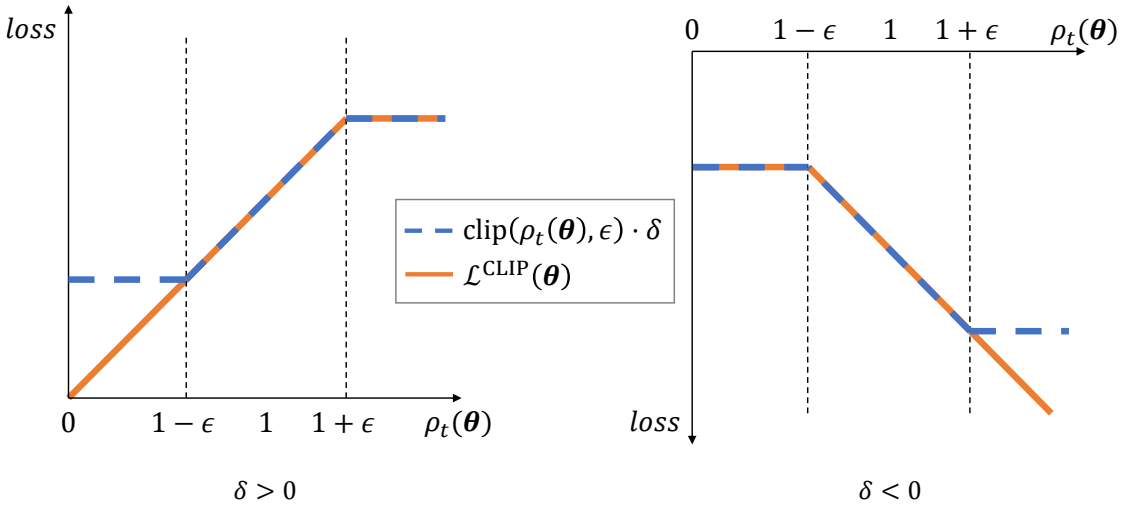

Figure 8: The loss for an action with positive advantage can be seen on the left side, whereas the loss for an action with a negative advantage is shown on the right side. By using a minimum in $\mathcal{L}^{\text{CLIP}}(\boldsymbol{\theta})$, we ensure that we can correct errors we made in previous gradient steps: We can raise the probability of actions with positive advantage even when $\rho_t(\boldsymbol{\theta}) \leq 1-\epsilon$. Vice versa, we can decrease the probability of actions with negative advantage when $\rho_t(\boldsymbol{\theta}) \geq 1+\epsilon$. Figure as in Wang et al.'s (2019) publication.

Clipping with ϵ keeps the new policy close to the old policy. The region determined by ϵ is called a *trust region*. The larger ϵ the more the new policy can deviate. On the one hand, if ϵ is too large, the performance of the policy may collapse again. On the other hand, an agent may learn too slowly if ϵ is too small.

By optimizing the loss $\mathcal{L}^{\text{CLIP}}(\boldsymbol{\theta})$, we can optimize the policy $\pi_{\boldsymbol{\theta}}$ without making excessively large gradient steps and still correct errors of previous optimizations given a suitable choice for ϵ.

3.3.3. Value Function Loss

In order to determine advantages, we require to know the value function and therefore a means to learn the value function. As we perform function approximation, the value function is parameterized by $\boldsymbol{\omega}$, a neural network. To learn reliable value estimations, we perform stochastic gradient descent

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t - \alpha \cdot \nabla_{\boldsymbol{\omega}_t} \mathcal{L}^{\text{VF}}(\boldsymbol{\omega}_t)$$

with the value function loss $\mathcal{L}^{\text{VF}}$.

We estimate the value function by recording a trajectory and calculating λ-returns G_t^λ for each state encountered. Then, the loss is the mean squared error of value estimations of $\boldsymbol{\omega}_t$ and the observed λ-returns (Schulman et al., 2017):

$$\mathcal{L}^{\text{VF}}(\boldsymbol{\omega}) \doteq \frac{1}{2} \cdot \mathbb{E}_{s,G\sim\pi_{\boldsymbol{\theta}_{\text{old}}}} \left[(\hat{v}_{\pi_{\boldsymbol{\theta}}}(s, \boldsymbol{\omega}) - G)^2 \right], \tag{27}$$

with G being λ-returns calculated from rewards observed by an agent following $\pi_{\boldsymbol{\theta}}$ (cf. equation 22).

3.3.4. Shared Parameterization Loss

When combining function approximation with policy parameterization, the value function and the policy may share the same parameters. $\hat{v}(s, \boldsymbol{\theta})$ and $\pi(a \mid s, \boldsymbol{\theta})$ share the same neural network architecture and weights with differing output layers (cf. chapter 4.2). We choose to share parameters because we require less computation power, as only one neural network needs to be trained and executed.

In this case, the loss must contain a loss for the policy and a loss for the value function. Since we perform gradient ascent on the policy loss but gradient descent on the value function loss, $\mathcal{L}^{\text{VF}}(\boldsymbol{\theta})$ is subtracted from $\mathcal{L}^{\text{CLIP}}(\boldsymbol{\theta})$.

Schulman et al. (2017) propose the following loss:

$$\mathcal{L}^{\text{CLIP+VF+}S}(\boldsymbol{\theta}) \doteq \mathcal{L}^{\text{CLIP}}(\boldsymbol{\theta}) - c_1 \cdot \mathcal{L}^{\text{VF}}(\boldsymbol{\theta}) + c_2 \cdot \mathbb{E}_{s\sim\pi_{\boldsymbol{\theta}}}\left[S\left[\pi_{\boldsymbol{\theta}}\right](s)\right], \tag{28}$$

with c_1 and c_2 being hyperparameters. c_1 controls the impact of the value function loss $\mathcal{L}^{\text{VF}}$, whereas c_2 adjusts the impact of the *entropy bonus* S.

S denotes an entropy bonus encouraging exploration, which addresses an issue raised by Kakade and Langford (2002): Policy gradient methods commonly transition to exploitation too early, resulting in suboptimal policies. The closer the distribution π is to a uniform distribution, the larger is its entropy. If all actions are assigned the same probability, an agent following π explores properly. An agent following a deterministic policy does not explore and the entropy of this policy will be 0.

Hence, the entropy bonus naturally declines over the course of the training as the policy transitions from exploration to exploitation and approaches a (locally) optimal policy. A common choice for S is to determine the mean entropy of $\pi_{\boldsymbol{\theta}}$ over all observed states.

3.4. Algorithm

By optimizing the shared loss $\mathcal{L}^{\text{CLIP+VF+}S}$ through gradient ascent, we calculate an improved policy. As not all information is learned with a single gradient step, Schulman et al. (2017) propose performing multiple optimization passes on the same trajectory. This approach increases the sample efficiency of Proximal Policy Optimization, as an agent can learn quicker than an agent that trains on a trajectory once.

More precisely, each training iteration consists of K epochs. In each epoch, the entire data set is split into randomly drawn disjunct minibatches of size M.[11] This approach alleviates the issue of learning on highly correlated observation as observed by Mnih et al. (2013).

We further ensure that we learn on sufficiently independent observations by running N *actors* at the same time.[12] Although each actor is embedded in its own environment, the N environments are described by the same Markov decision process (Mnih et al., 2016). However, the environments are independent and therefore give rise to different trajectories. The observations of all actors are then combined to optimize the policy $\pi_{\boldsymbol{\theta}}$ that all N actors follow.

[11] For a detailed explanation of minibatch gradient methods, refer to the work of Goodfellow et al. (2016, chapter 8.1.3).

[12] *Actor* is the technical term used to describe agents that are run in parallel. Often, these agents follow the same policy $\pi_{\boldsymbol{\theta}}$, as is the case with Proximal Policy Optimization.

PPO—and many other reinforcement learning algorithms—operate in two phases. In the first phase, an agent interacts with its environment and generates a *rollout* τ. τ contains not only the states and rewards the agent observed, but also the chosen actions, their respective probabilities and the values of the states. This phase can be seen in lines 5–17 of algorithm 1.

In the second phase, the value function approximation and the policy are optimized with the data the agent collected. These two steps are repeated for a certain time or until the value function and policy converge to their respective optimal functions. Afterwards, the learn rate α and the clipping parameter ϵ are decreased, so they linearly approach 0 over the course of the training; the trust region shrinks. Lines 18–25 of algorithm 1 show this phase.

Rollouts always have a length of T time steps. Hence, the horizon of an episode might not align with the horizon of a rollout—an episode might terminate earlier, or it might not terminate at all. We address these issues as follows:

- If an episode does not terminate whilst recording a rollout, we must adjust advantage and return calculation. We can only compute λ-returns and GAE advantages up to the rollout time step T. Then we bootstrap the missing rewards by adding an appropriately discounted value estimation $\hat{v}_\pi(S_T, \boldsymbol{\omega})$ of the final state included in the rollout.

- If an episode terminates early, we only include rewards and values up until the episode terminated. Let T_{episode} denote this time step. Then all advantage and return estimations for time steps $t \leq T_{\text{episode}}$ use the rollout time step that corresponds with T_{episode} as the upper bound of summation (cf. equations 21 and 22).

Algorithm 1 Proximal Policy Optimization, modified from Schulman et al.'s (2017) and Peng et al.'s (2018) works

Require: number of iterations I, rollout horizon T, number of actor N, number of epochs K, minibatch size M, discount γ, GAE weight λ, learn rate α, clipping parameter ϵ, coefficients c_1, c_2

$\boldsymbol{\theta} \leftarrow$ random weights
Initialize environments E
Number of minibatches $B \leftarrow N \cdot T \ / \ M$
for iteration=$1, 2, \ldots, I$ **do**
    $\tau \leftarrow$ empty rollout
    for actor=$1, 2, \ldots, N$ **do** ▷ record data
        $s \leftarrow$ current state of environment E_{actor}
        Append s to τ
        for step=$1, 2, \ldots, T$ **do**
            $a \sim \pi_{\boldsymbol{\theta}}(a \mid s)$
            $\pi_{\boldsymbol{\theta}_{\text{old}}}(a \mid s) \leftarrow \pi_{\boldsymbol{\theta}}(a \mid s)$
            Execute action a in environment E_{actor}
            $s \leftarrow$ successor state
            $r \leftarrow$ reward
            Append $a, s, r, \pi_{\boldsymbol{\theta}_{\text{old}}}(a \mid s), \hat{v}_\pi(s, \boldsymbol{\theta})$ to τ
        end for
    end for
    Compute Generalized Advantage Estimations $\delta_t^{\text{GAE}(\gamma,\lambda)}$
    Compute λ-returns G_t^λ
    for epoch=$1, 2, \ldots, K$ **do** ▷ optimize
        for minibatch=$1, 2, \ldots, B$ **do**
            $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \nabla_{\boldsymbol{\theta}} \, \mathcal{L}^{\text{CLIP+VF+}S}(\boldsymbol{\theta})$ on minibatch with size M
        end for
    end for
    Anneal α and ϵ linearly
end for

4. Realization

In order to train an agent for ATARI games with Proximal Policy Optimization, adaptations to the ATARI environment (cf. chapter 5.1) need to be made. Furthermore, an architecture for the neural network $\boldsymbol{\theta}$ that parameterizes the policy and the value function is required.

Although algorithm 1 enables learning, several improvements such as a specific initialization scheme or extending clipping to the value function loss are vital to the performance of PPO. As a consequence, doubts were raised on the mathematical foundation of the algorithm (Ilyas et al., 2018; Engstrom et al., 2019; Wang et al., 2019).

In this chapter, we introduce the required operations to adapt the ATARI environment to PPO as well as the algorithmic improvements. Often, mathematical reasoning for an operation is given neither in its respective original publication nor in publications investigating it. In these cases, it can be assumed that the operation has been proven to be beneficial empirically. We evaluate some of these choices in chapter 5.

4.1. Environment Adaptation

A sequence of adaptation steps is performed on observations returned by the ATARI 2600 emulator. Except for *reward clipping*, these adaptations are common when learning ATARI 2600 games. Operations such as *fire resets*, *frame skipping* and handling *episodic life* adapt specific behavior of ATARI games to the requirements of reinforcement learning algorithms. Other operations reduce the complexity of the state space or lead to more diverse starting conditions.

We use ϕ to denote the adaptation operations and overload it as follows: $\phi_s(S_t)$ processes states, whereas $\phi_r(R_t)$ transforms rewards. In practice, both functions are executed at the same time due to the nature of the adaptation steps. Mathematically, ϕ is part of the environment and therefore not visible in terms of the Markov decision process.

No operation resets. Whenever an environment is reset, a random number of *noop* actions are performed for up to 30 frames in order to guarantee differing initial conditions across different trajectories. Originally, this approach was proposed to ensure that an agent under evaluation does not overfit (Mnih et al., 2015), but it has since been adopted to training. This might prove beneficial, as the random initial conditions can lead to more diverse trajectories. However, it appears that no research has been conducted on the full implications of this choice.

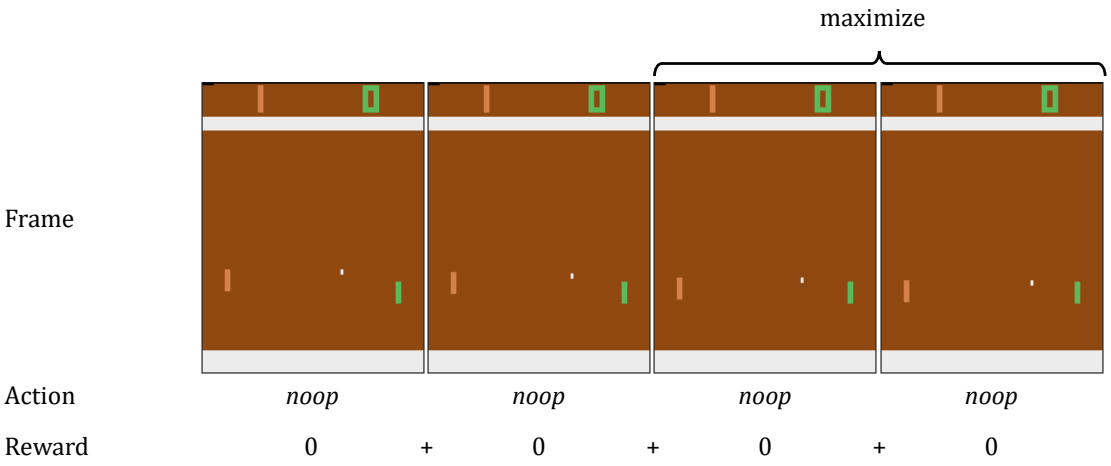

Figure 9: The agent (green) repeats the action *noop* for $k = 4$ frames.

Frame skipping. Instead of prompting the agent for an action every frame, we show the agent every k-th frame only (Bellemare, Naddaf, Veness, & Bowling, 2013; Mnih et al., 2013). As humans rarely perform frame-by-frame action selections, repeating an action for several frames should prove no detriment for an agent. Thus, an agent can easily train on up to k times more frames without the computation time of training being affected significantly. A common selection is $k = 4$, which means that an agent repeats an action four times and computes a maximized frame. Rewards observed during these k frames are summed up (cf. figure 9).

Because there are some objects that appear only on even frames whilst others appear only on odd frames, showing an agent only the final frame causes a loss of information.[13] We solve this issue by taking the maximum value of each pixel across the last two frames (Mnih et al., 2015).

Episodic life. Some games like Breakout provide the player with several attempts, often called lives. Whenever the player loses a life, they may resume the game, usually at the cost of a reduced score. Mnih et al. (2015) propose ending a training episode once a life is lost. As PPO operates on a steady number of samples, this approach is modified slightly. Instead of ending the episode, we simulate the end of the game. Thus, the return and advantage calculation are cut off at this time step. Finally, we reset the observation stack as detailed in the operation *observation stacking*.

[13] This is due to a constraint of the ATARI 2600 console.

This adaptation is applied to some of the games chosen for this thesis only (see chapter 5.1 for more information on the selected games). BeamRider, Breakout, Seaquest and SpaceInvaders provide a player with multiple lives, whereas Pong does not.

Fire resets. A few games require that the player performs a *fire* action at the beginning of a game episode or after the loss of a life to start or resume the game (OpenAI Inc., 2017a, baselines/common/atari_wrappers.py). This operation performs the necessary action.

Image transformation. The maximized frame returned by the frame skipping operation is further modified (Mnih et al., 2015). Firstly, the image is converted to grayscale, as the red, blue and green channels are not required. Secondly, the image is resized to 84×84 pixels. Lastly, the grayscale image is divided by 255, so each pixel's value is in the interval $[0, 1]$. Resizing the image and converting it to grayscale greatly reduces the complexity of the state space. Normalizing input is a common choice when using neural networks.

Reward clipping. According to Ilyas et al. (2018), rewards are clipped to the interval $[-5, 5]$. However, an examination of the baselines repository reveals that rewards are binned (OpenAI Inc., 2017a, baselines/common/atari_wrappers.py):

$$\phi_r(r) \doteq \operatorname{sign} r. \tag{29}$$

We discuss both choices in chapter 5.5.1.

Observation stacking. Finally, the four most recent maximized images seen by an agent are combined to a tensor with shape $4 \times 84 \times 84$. This operation is often called observation stacking or frame stacking. At the beginning of an episode, all elements of the tensor are set to 0, which represents the color black. The same applies to simulated episode ends and beginnings as performed by the *episodic life* operation.

Although no explanation is given by Mnih et al. (2015), one benefit is apparent: By stacking images, we provide an agent further information on the state of the game such as direction and movement. If the agent would see one frame only, it could not determine which direction the ball is moving in Pong. By showing it four frames at once, the agent can discern if the ball is moving towards itself or the enemy or whether it will hit a wall or not (cf. figure 10).

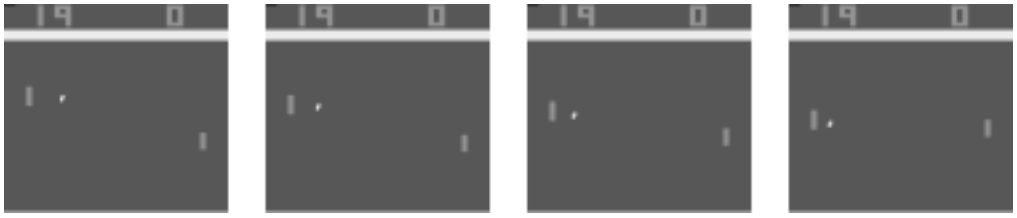

Figure 10: A state of Pong consists of the four most recent transformed frames; the oldest frame is to the left. The movement of the ball can easily be seen in these four frames.

Due to transforming the image to grayscale and scaling it to $[0, 1]$, the set of states $\mathcal{S}$ is a subset of $[0, 1]^{4\times84\times84}$.

4.2. Model Architecture

Although Proximal Policy Optimization may be used with any differentiable function, neural networks are the most commonly used solution, e.g., in the baselines repository by OpenAI Inc. (2017a, baselines/common/models.py). With few exceptions, most deep policy gradient algorithms use the architecture established by Mnih et al. (2015) when learning ATARI 2600 games.

This architecture is composed of three convolutional layers[14] followed by a linear layer[15] and two output layers (cf. figure 11). The input is scaled to $4 \times 84 \times 84$, that is 4 grayscale images each sized 84×84. Table 1 outlines the structure of the convolutional layers:

Convolutional layer	Number of filters	Kernel size	Stride
1	32	8×8	4
2	64	4×4	2
3	64	3×3	1

Table 1: The neural network contains three convolutional layers that utilize different numbers of filters, kernel sizes and strides.

Each convolutional layer is follow by a rectified linear unit[16]. The results computed by the third rectified linear unit are flattened and used as input to a linear layer with 512 outputs. The activation function of this linear layer is once again a rectified linear unit.

[14] Convolutional neural networks are explained in detail by Goodfellow et al. (2016, chapter 9).
[15] Also known as dense layers, fully-connected layers or occasionally multi-layer perceptrons.
[16] An explanation of rectified linear units is given by Goodfellow et al. (2016, pp. 174–175).

Figure 11: This figure shows the architecture used in this thesis. The first four execution steps—from left to right—use a rectified linear unit. The fourth layer feeds into two output layers. The action output utilizes a softmax to generate a probability distribution, whereas the value function output has no particular activation function.

In this thesis, the policy and the value function share parameters. We choose this approach because this way we must only train one neural network instead of two. This in turn means that the computation time is reduced slightly. Furthermore, popular PPO implementations follow the same approach (OpenAI Inc., 2017a; Kostrikov, 2018). In order to share parameters, we construct two output layers, both of which follow the size 512 linear layer:

- The policy output is computed by a linear layer with one output for each action available to the agent. $\pi_{\boldsymbol{\theta}}$ is given by performing a softmax on the results of this output layer. Softmax functions can be used to represent a probability distribution over discrete random variables (Goodfellow et al., 2016, pp. 184–185), as is the case with actions in ATARI games.
- The value function output is composed of a single artificial neuron. The output of this neuron is the value estimation $\hat{v}_\pi(s, \boldsymbol{\theta})$.

4.3. Advantage Normalization

Albeit Schulman et al. (2017) do not mention it, advantage estimations δ are normalized before loss computation. Normalizing advantages is a well-known operation that lowers the variance of the gradient estimator.

$$\delta \leftarrow \frac{\delta - \text{mean}(\delta)}{\text{std}(\delta) + 10^{-5}} \tag{30}$$

Due to the update[17] seen in equation 30, advantages are normalized, so they have a mean of 0 and a standard deviation of 1. We add 10^{-5} to the divisor for numerical stability.

4.4. Initialization

Ilyas et al. (2018) found that the weights of the neural network used by OpenAI Inc. (2017a, baselines/ppo2) are initialized using orthogonal initalization.[18] The impact of this choice appears to be subject of empirical examinations only (Ilyas et al., 2018; Engstrom et al., 2019). Table 2 lists each layer of the neural network and the corresponding scaling factor that is used to initialize the layer.

[17] As common in computer science, we use $\leftarrow$ to denote an assignment. Hence, the advantage estimations δ are updated with their normalized values.

[18] Using orthogonal initialization with large neural networks was proposed by Saxe, McClelland, and Ganguli (2014). The authors also provide mathematical foundations and examine the benefits of various initialization schemes.

Layers	Gain
Convolutional layers	$\sqrt{2}$
Size 512 linear layer	$\sqrt{2}$
Policy output	0.01
Value function output	1

Table 2: Each layer is subject to orthogonal initialization but the scaling factors differ. Layers that use a rectified linear unit are initialized with $\sqrt{2}$, whereas the policy output is scaled by 0.01.

4.5. Value Function Loss Clipping

Although it is neither mentioned nor motivated by Schulman et al. (2017), clipping is applied to the value function loss as well (Ilyas et al., 2018). Let clip_v denote a clipping function similar to equation 25:

$$\text{clip}_v(\boldsymbol{\omega}, \boldsymbol{\omega}_{\text{old}}, \epsilon, S_t) \doteq \begin{cases} \hat{v}_\pi(S_t, \boldsymbol{\omega}_{\text{old}}) - \epsilon & \text{if } \hat{v}_\pi(S_t, \boldsymbol{\omega}) \leq \hat{v}_\pi(S_t, \boldsymbol{\omega}_{\text{old}}) - \epsilon, \\ \hat{v}_\pi(S_t, \boldsymbol{\omega}_{\text{old}}) + \epsilon & \text{if } \hat{v}_\pi(S_t, \boldsymbol{\omega}) \geq \hat{v}_\pi(S_t, \boldsymbol{\omega}_{\text{old}}) + \epsilon, \\ \hat{v}_\pi(S_t, \boldsymbol{\omega}) & \text{otherwise.} \end{cases} \tag{31}$$

Then the clipped value function loss $\mathcal{L}^{\text{VFCLIP}}$ is defined to be

$$\mathcal{L}^{\text{VFCLIP}}(\boldsymbol{\omega}) \doteq \max \left[\mathcal{L}^{\text{VF}}(\boldsymbol{\omega}), \mathbb{E}_{s,G\sim\pi} \left[\left(\frac{1}{2} \cdot \text{clip}_v(\boldsymbol{\omega}, \boldsymbol{\omega}_{\text{old}}, \epsilon, s) - G \right)^2 \right] \right], \tag{32}$$

with ϵ begin the same hyperparameter that is used to clip the likelihood ratio in $\mathcal{L}^{\text{CLIP}}$ (cf. equation 26 in chapter 3.3.2). $\boldsymbol{\omega}_{\text{old}}$ denotes the parameter vector that was used when the rollout was recorded.

Intuitively, this approach may be like clipping the probability ratio $\rho_t(\boldsymbol{\theta})$. To avoid gradient collapses, a trust region is created with the clipping parameter ϵ. Then, an elementwise maximum is taken so errors from previous gradient steps can be corrected. A maximum is applied instead of a minimum because the value function loss is minimized. Proper analysis on the ramifications of using this surrogate loss was recently published by Ilyas et al. (2020).

4.6. Gradient Clipping

Before performing a gradient step, the gradient $\nabla_\theta \mathcal{L}^{\text{CLIP+VFCLIP+}S}$ is clipped (Ilyas et al., 2018). All changes in weights are concatenated into a single vector. The weight

changes are adjusted such that the Euclidean norm of the vector does not exceed 0.5. Gradient clipping is a technique commonly used to prevent large gradient steps (Goodfellow et al., 2016, pp. 413–415).

4.7. Complete Algorithm

Algorithm 2 (on page 42) shows the complete Proximal Policy Optimization algorithm when learning ATARI 2600 games. Notable changes from algorithm 1 in chapter 3.4 are the inclusion of the environment adaptation ϕ (lines 7, 13 and 14), the introduction of orthogonal initialization (line 1), the replacement of $\mathcal{L}^{\text{VF}}$ with $\mathcal{L}^{\text{VFCLIP}}$ and the use of gradient clipping (both in line 23).

Table 3 lists popular values for all hyperparameters (OpenAI Inc., 2017a; Kostrikov, 2018). It differs slightly from the one used by Schulman et al. (2017), as the authors trained agents for $K = 3$ epochs with a value function loss coefficient $c_1 = 1.0$.

Hyperparameter	Value
Rollout horizon T	128
Number of actors N	8
Iterations I	9765
Number of epochs K	4
Minibatch size M	256
Learn rate α	$2.5 \cdot 10^{-4}$
Discount γ	0.99
Variance tuning parameter λ	0.95
Clipping parameter ϵ	0.1
Value function coeff. c_1	0.5
Entropy coeff. c_2	0.01
Maximum gradient norm	0.5
Gradient optimizer	Adam[19]

Table 3: This table contains the most commonly used configuration of Proximal Policy Optimization for ATARI 2600 games.

4.8. Performance Optimization

Since Proximal Policy Optimization executes N actors simultaneously, the runtime of a training can be greatly reduced on multicore processors by parallelizing the execution. In this thesis, each actor operates in a dedicated process. They are coordinated by a

[19] Consult Goodfellow et al. (2016, chapter 8.5.3) for an explanation of the Adam optimization algorithm.

parent process that gathers the observations and optimizes the policy and the value function.

The child processes communicate with the parent process through pipes. The actors transmit observations consisting of their environment's current state, the last reward and information on the remaining lives to the parent process. Afterwards, they await an action to perform.

The parent process determines each actor's action by sampling from the policy using the respective environment's state. It creates rollouts from the actors' observations, the actions and their respective probabilities as well as value estimations. After rollout generation has concluded, the parent process computes the loss and performs stochastic gradient descent.

Algorithm 2 Full Proximal Policy Optimization for ATARI 2600 games, modified from Schulman et al.'s (2017) and Peng et al.'s (2018) publications. Lines commented with a ▷ are changed from algorithm 1.

Require: number of iterations I, rollout horizon T, number of actor N, number of epochs K, minibatch size M, discount γ, GAE weight λ, learn rate α, clipping parameter ϵ, coefficients c_1, c_2, adaptation function ϕ

$\boldsymbol{\theta} \leftarrow$ orthogonal initialization ▷
Initialize environments E
Number of minibatches $B \leftarrow N \cdot T \;/\; M$
for iteration=$1, 2, \ldots, I$ **do**
    $\tau \leftarrow$ empty rollout
    for actor=$1, 2, \ldots, N$ **do**
        $s \leftarrow \phi_s$(current state of environment E_{actor}) ▷
        Append s to τ
        for step=$1, 2, \ldots, T$ **do**
            $a \sim \pi_{\boldsymbol{\theta}}(a \mid s)$
            $\pi_{\boldsymbol{\theta}_{\text{old}}}(a \mid s) \leftarrow \pi_{\boldsymbol{\theta}}(a \mid s)$
            Execute action a in environment E_{actor}
            $s \leftarrow \phi_s$(successor state) ▷
            $r \leftarrow \phi_r$(reward) ▷
            Append $a, s, r, \pi_{\boldsymbol{\theta}_{\text{old}}}(a \mid s), \hat{v}_\pi(s, \boldsymbol{\theta})$ to τ
        end for
    end for
    Compute Generalized Advantage Estimations $\delta_t^{\text{GAE}(\gamma,\lambda)}$
    Normalize advantages ▷
    Compute λ-returns G_t^λ
    for epoch=$1, 2, \ldots, K$ **do**
        for minibatch=$1, 2, \ldots, B$ **do**
            $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \text{clip}(\alpha \cdot \nabla_{\boldsymbol{\theta}}\, \mathcal{L}^{\text{CLIP+VFCLIP+}S}(\boldsymbol{\theta}))$ on minibatch with size M ▷
        end for
    end for
    Anneal α and ϵ linearly
end for

5. Evaluation

Recalling from chapter 1.1, modern video games such as StarCraft II or Dota 2 pose a great challenge for reinforcement learning. Solving them requires significant resources and a combination of reinforcement learning algorithms with other methods.[20] As a consequence, these games are infeasible as a means of comparing different algorithms.

However, benchmarking reinforcement learning algorithms is a necessity when developing improved algorithms. Key metrics are not only the rewards an agent collects after training, but also the sample efficiency that describes how quickly an agent learns. Even the required computation time can be an important criterion, as algorithms with a low computation time can be trained on more time steps than highly demanding algorithms in the same duration.

In this chapter, the *Arcade Learning Environment* (ALE) is introduced, which is the benchmarking framework we use to evaluate Proximal Policy Optimization. Afterwards, we discuss the evaluation method and issues one encounters when attempting to reproduce results. We close by highlighting three experiments and discussing the results.

5.1. Arcade Learning Environment

Typical benchmarks involve classic control tasks such as balancing an inverse pendulum, robotics tasks or ATARI video games. Although the ATARI 2600 console was released in 1977, the games developed for the console prove challenging even today (Bellemare et al., 2013). The authors note that despite the simple graphics, ATARI games can be difficult to predict: For example, the 18 most recent frames are used to determine the movement of the paddle in Pong. Moreover, agents are trained on video input from ATARI games, which leads to a large state space (cf. chapter 4.1). Lastly, video games are diverse challenges with some—such as Pong—being easier to solve, whereas others like the game Montezuma's Revenge remain challenging. For these reasons, ATARI 2600 video games are a popular benchmarking environment.[21]

The *Arcade Learning Environment* (ALE) provides a benchmarking environment containing 57 games (Bellemare et al., 2013).[22] It is included, among other benchmarking environments, in the popular OpenAI Gym framework (Brockman et al., 2016).

We conduct experiments on a selection of five games of the ALE: BeamRider, Breakout, Pong, Seaquest and SpaceInvaders are highlighted by Mnih et al. (2013) due to their

[20] One technique is imitation learning, which allows an agent to learn from human example.
[21] One might note that video games are well-known and could pique people's interest as well.
[22] When it was published, the Arcade Learning Environment contained fewer games. It contains 57 games at the time of writing this thesis.

differing complexity. Reinforcement learning agents easily surpass human performance on Breakout and Pong, whereas they fail to achieve human performance on Seaquest and SpaceInvaders. Mnih et al. (2013) state that the latter games require long-term strategy and cannot be solved by reacting quickly and accurately. On BeamRider, agents achieve close to human performance.

OpenAI Gym and ALE simplify evaluation greatly, as they provide a unified interface for all environments. We simply provide an action to the environment and obtain observations containing the current state of the environment as a 210×160 RGB image (before adaptations), the reward and further information such as the number of lives remaining or if the game terminated. The Arcade Learning Environment runs at 60 frames per second when run in real-time—without frame skipping we would obtain 60 observations per second.

Since the ALE returns a reward, we do not have to design a reward function. Instead, all reinforcement learning algorithms are trained using the same reward function bar environment adaptations. The reward of an action is the change in score, making the cumulated rewards of an episode the high score.

The action space of ATARI games is discrete. Every game accepts the actions $\mathcal{A} = \{noop, fire, left, right, up, down\}$. Actions that are not supported by a game—e.g., moving *left* or *right* in Pong—cause no operation instead. As these actions are discrete, there is no intensity with which they can be performed. An agent either moves left or it does not, it cannot choose to move left slowly.

5.2. Method

Benchmarking reinforcement learning algorithms on ATARI 2600 video games is commonly done by training agents on $10,000,000$ time steps of data. Since Proximal Policy Optimization executes N actors simultaneously, the number of iterations must be adjusted to accord for N as well as the rollout horizon T. The number of iterations I then is

$$I \leftarrow \text{floor}(10,000,000 \;/\; (T \cdot N)).$$

With the common choices of $N = 8$ and $T = 128$ we train for $I = 9765$ iterations.

In order to evaluate the performance of the Proximal Policy Optimization algorithm implemented for this thesis, two of the three metrics mentioned in the introduction to this chapter are used. The runtime is not subject to evaluation for several reasons. Firstly, the goal of this thesis is the reproduction and evaluation of results published by

Schulman et al. (2017). The authors provide no measure of the computation time of their algorithm other than stating that it runs faster than others. Furthermore, a comparison of stated computation times is impractical, as it is reasonable to assume that hardware configurations differ greatly. Nevertheless, the computation time of the code written for this thesis roughly matches that of the code written by Kostrikov (2018) on a machine with an Intel i7-6700, 32 GB of RAM and a GeForce GTX 1060 6GB.

The two points of interest in evaluation are how quickly an agent learns and the performance of an agent at the end of its training. We evaluate how quickly an agent learns by drawing an *episode reward graph*. Whenever an episode of a game terminates, the score is plotted. As we run N environments simultaneously, several episodes could terminate at the same time. If this occurs, we plot the average score of all terminated episodes. Figure 12 displays an unsmoothed episode reward graph for the game Breakout. The x axis displays the training time step and the y axis shows the score. As a result, we get a graph that shows the performance of an agent over the course of its training. A smoothed graph of the same data can be seen in figure 14 on page 50.

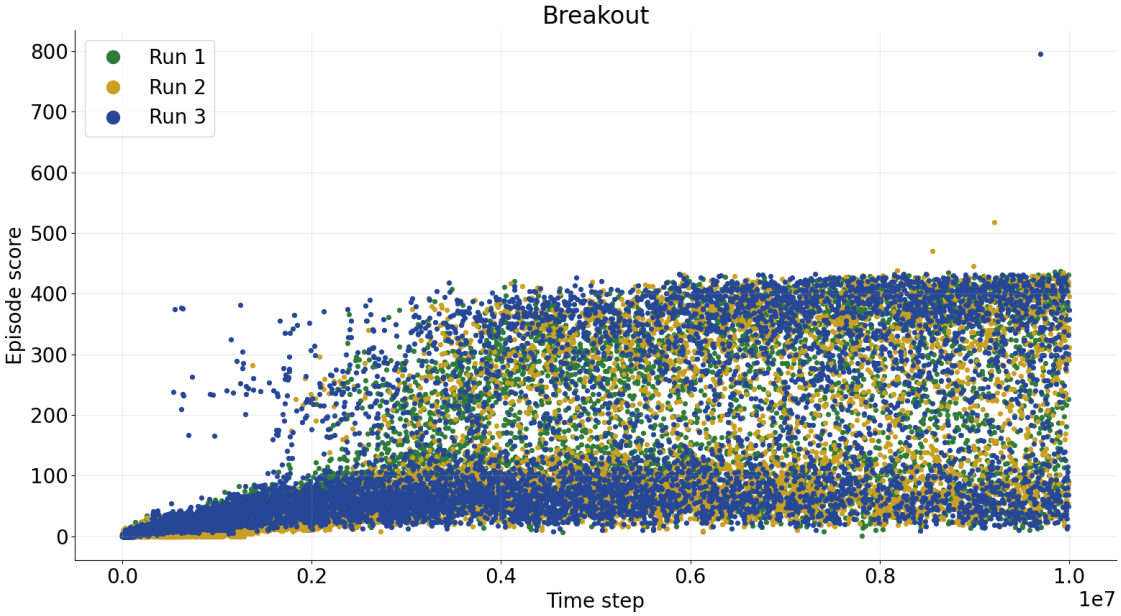

Figure 12: Unsmoothed episode reward graphs like this one are very noisy and therefore hard to examine. We address this issue by smoothing the graphs. Smoothed episode reward graphs may be seen in chapter 5.4.

Since agents encounter novel states regularly in the beginning of the training, the performance of episodes can vary greatly. This results in a noisy episode reward graph. Depending on the complexity of the game, the graph may remain noisy even until the

end of the training as seen in figure 12. We alleviate this issue by smoothing the reward graph—outliers are featured less prominently and the noise is reduced, whilst the overall trend of the training is preserved. Applying a suitable method to evaluate noise, such as confidence intervals, will be a topic for future research.

The reward graphs are smoothed by computing the average of a sliding window with 16 data points. This window is then centered, so each data point is the average of the 8 previous episodes and the 7 following ones. We note that this operation does not remove all noise, which allows us to compare if certain choices have a strong stabilizing effect. Furthermore, this is an educational choice as noisy graphs are to be expected depending on the environment.

As the performance in training depends on chance, Schulman et al. (2017) train three agents on each game for any given configuration. To enable easier comparison of configurations, we add a simple trendline. We compute this trendline by combining the data of all three runs ordered by time steps. Then, we compute the average of a sliding window with 256 data points. As with the individual runs, this window is centered such that each data point is the average of the 128 preceding data points and the 127 following ones. To compare the performance of a given configuration with that of a reference configuration, we add a trendline generated with the reference configuration.

We note that more advanced regression methods are available. However, their suitability for the task at hand will be a topic for future research. Although simple, the chosen method is intuitive and suitable for comparisons as done in the following.

The final score is determined by computing the average of the final 100 terminating episodes in training (Schulman et al., 2017). The final scores given by Schulman et al. (2017) are averaged scores of the three training runs.

The following experiments were executed on a cluster consisting of 15 machines. Each machine possesses an Intel i7-6700, 32 GB of RAM and a GeForce GTX 1060 6GB. The runtime of a single experiment amounts to approximately three hours on this cluster.

5.3. Reproduction

Reproducing results of deep reinforcement learning algorithms can be challenging for multiple reasons. Since the learning speed of Proximal Policy Optimization is assessed with reward graphs, it is hard to compare results from various sources as one needs to compare multiple graphs in different figures. This issue is exacerbated by the fact that the reward graphs plotted by Schulman et al. (2017) are smoothed, but the smoothing method is not disclosed in the paper.

Another problem stems from the fact that the exact configuration of the algorithm used to achieve the published performance is unknown. It is unclear if optimizations such as advantage normalization (cf. chapter 4.3) and value function loss clipping (cf. chapter 4.5) were used in the publication. These optimizations are not part of the initial PPO code (OpenAI Inc., 2017a, baselines/ppo1) and were not mentioned in an update on the repository either (OpenAI Inc., 2017b).

Debugging can be troublesome due to the frameworks used. Even when multiple implementations are available, ensuring identical results is all but simple. The Arcade Learning Environment and the deep learning framework may rely on randomness, for example when sampling actions. Thus, one must ensure identical seeds and the same way of interfacing with the random number generator to make sure that its state remains the same in all implementations used for comparison. If the different implementations do not use the same deep learning frameworks, this issue may be hard to overcome.

But even when the same framework is used—such as PyTorch, which is used for this thesis—ensuring identical seeds and random number generator states might not be sufficient: Some operations are inherently non-deterministic, so these operations must not be used (PyTorch Contributors, 2019).

Because PPO as executed in the following experiments is trained on minibatches of size 256, manually calculating the loss is infeasible. Manually solving a backpropagation training and calculating weight changes for a neural network with approximately 80000 parameters is unrealistic.

In order to ensure that the implementation written for this thesis is working as intended, several tests were conducted. Approximately 250 unit and integration tests cover the implementation. Most of these tests validate the loss calculation with many small scale tests being the results of manual calculations. A few larger tests spanning losses calculated from up to 128 time steps of data were generated from data obtained from the implementation of Kostrikov (2018). Furthermore, several tests were run by replacing components of the code of both Kostrikov (2018) and Jayasiri (n.d.) with components written for this thesis.

As the implementation is tested extensively and the results of the experiment shown in chapter 5.4.1 match or exceed those published by Schulman et al. (2017), one can trust that the code written for this thesis allows for a reliable evaluation.

5.4. Experiments

A total of 47 experiments were conducted to evaluate optimizations listed in chapter 4 and values of hyperparameters. For most experiments, two graphs are generated for each game: One including only the training itself and one adding a trendline of the reference configuration of PPPO for ATARI games. By reference configuration we mean the hyperparameters outlined in chapter 4.7 with all optimizations listed in chapters 4.3–4.6. It uses reward binning instead of reward clipping (cf. chapter 4.1).

As we test each configuration on 5 games, this makes for over 470 graphs. For this reason, only a selection of graphs is shown in this thesis. However, all graphs are made available online including the configuration and raw data used to generate them.[23] When we refer to an experiment, consult the repository unless a figure or table is specified.

Subsequently, we show three different experiments:

1. By evaluating the reference configuration we determine that the implementation written for this thesis achieves the intended performance.

2. Schulman et al. (2017) claim that PPO is robust to hyperparameter choice. Hence, we evaluate the performance of agents when a parameter is severely misconfigured.

3. The effect of the non-disclosed optimizations is shown.

We discuss the findings in chapter 5.5.

5.4.1. Reference Configuration

Figures 13–17 show the reward graphs of three training runs on each of the games chosen for evaluation with the reference configuration. The final scores of the three runs and the scores reported by Schulman et al. (2017) are shown in table 4.

Game	Run 1	Run 2	Run 3	Schulman et al. (2017)
BeamRider	2432.3	2324.5	2246.6	1590.0
Breakout	257.3	252.9	230.4	274.8
Pong	20.9	20.9	20.6	20.7
Seaquest	1756.6	1744.4	928.0	1204.5
SpaceInvaders	1026.1	763.8	670.6	942.5

Table 4: Final scores achieved in training with the reference configuration and final score given by the authors of Proximal Policy Optimization.

[23] The configuration, raw data and images are available at `https://github.com/Aethiles/ppo-results`.

Comparing the shapes of the reward graphs with those given by Schulman et al. (2017) reveals no discernable differences except for the magnitude of score obtained in BeamRider. On BeamRider and Seaquest, the algorithm implemented for this thesis achieves a higher final score than the originally reported results. On Breakout and SpaceInvaders, the obtained score is slightly lower.

Most likely, these differences can be attributed to differing configurations, as Schulman et al. (2017) train with a value function loss coefficient $c_1 = 1.0$ and $K = 3$ epochs instead of $c_1 = 0.5$ and $K = 4$. Training with the original configuration grants even more favorable results (cf. experiment *paper_configuration*), which means that some of the optimization methods outlined in chapter 4 likely were not used for the publication.

We can easily see a lot of noise in the plots for BeamRider, Breakout and SpaceInvaders (cf. figures 13, 14 and 17). This is expected, as our agents follow stochastic policies and may pick suboptimal actions that allow them to explore the state space. Furthermore, the agents still encounter novel states that they have no knowledge on leading to more random actions. Finally, there is a very apparent outlier in the Seaquest runs. The same can be observed in Schulman et al.'s (2017) graphics. We discuss this phenomenon in chapter 5.5.

As the performance of the implementation in this thesis matches or exceeds publicized results, one can reasonably assume that the implementation is suitable for the intended evaluation of hyperparameters and the optimizations.

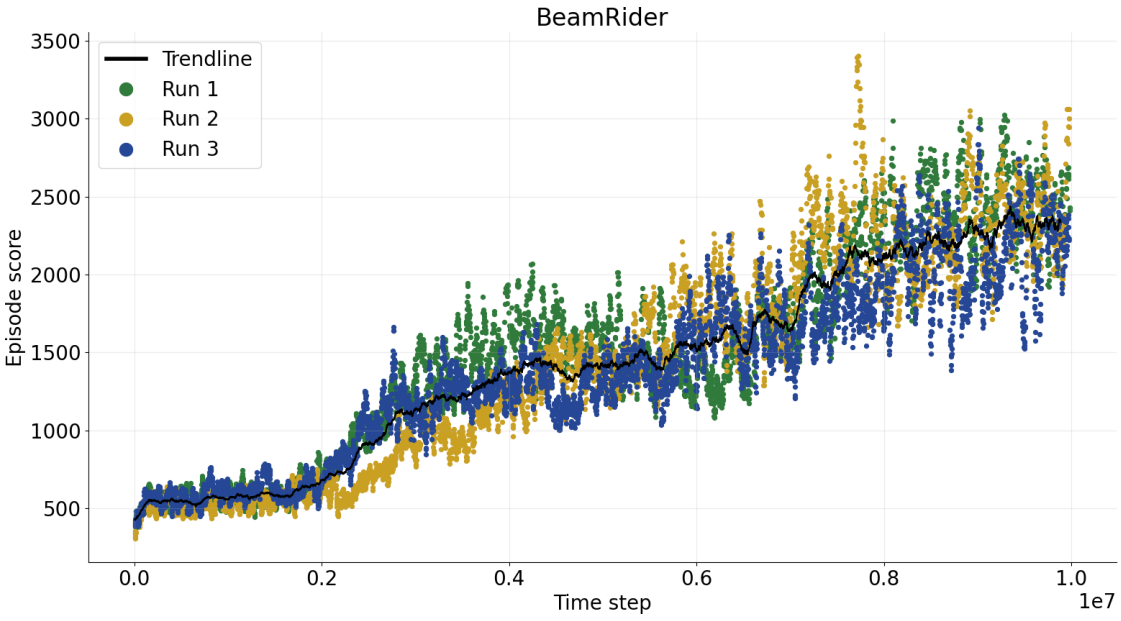

Figure 13: BeamRider with the reference configuration

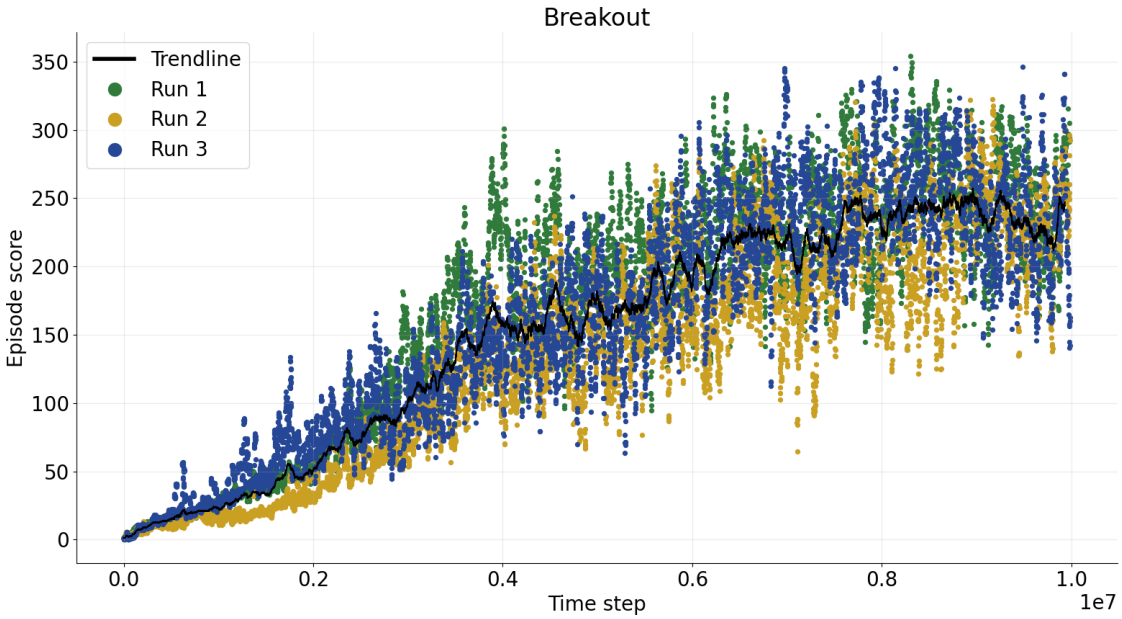

Figure 14: Breakout with the reference configuration

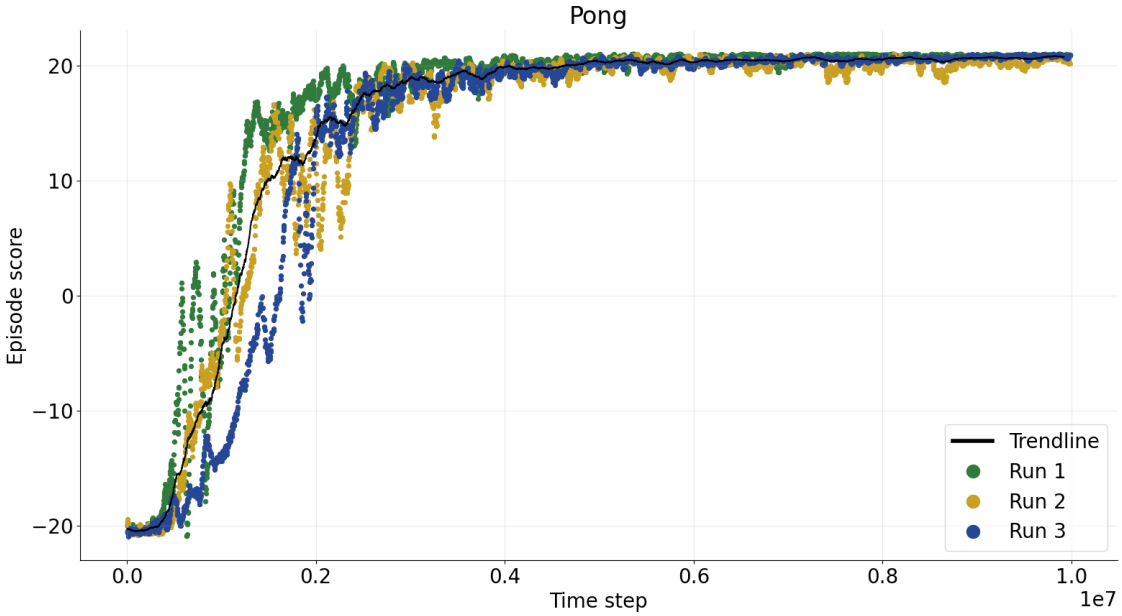

Figure 15: Pong with the reference configuration

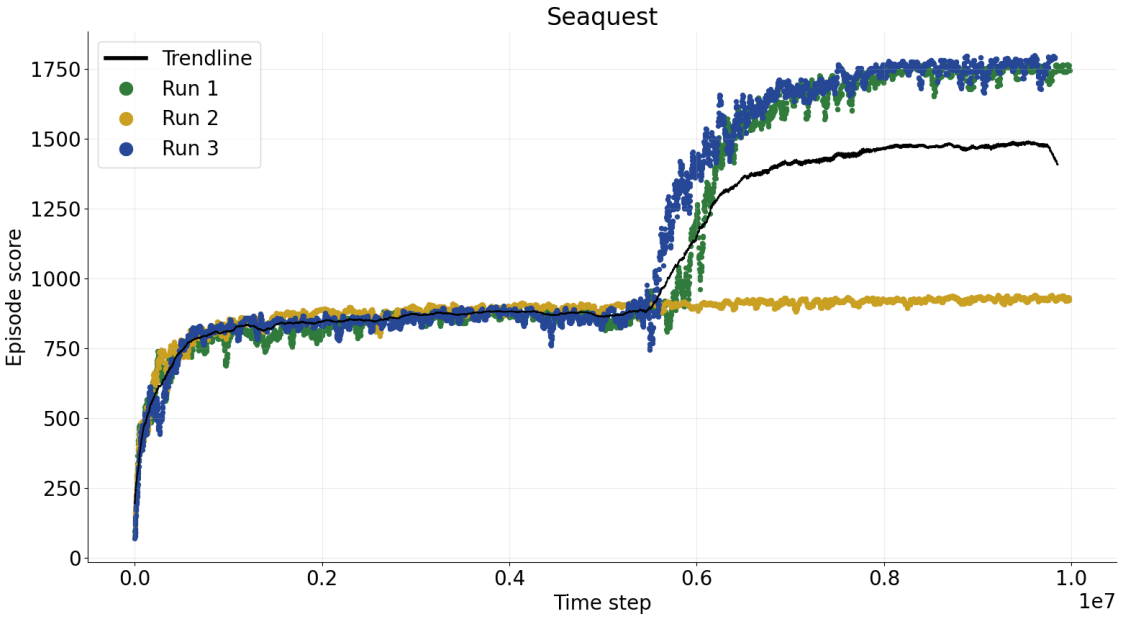

Figure 16: Seaquest with the reference configuration

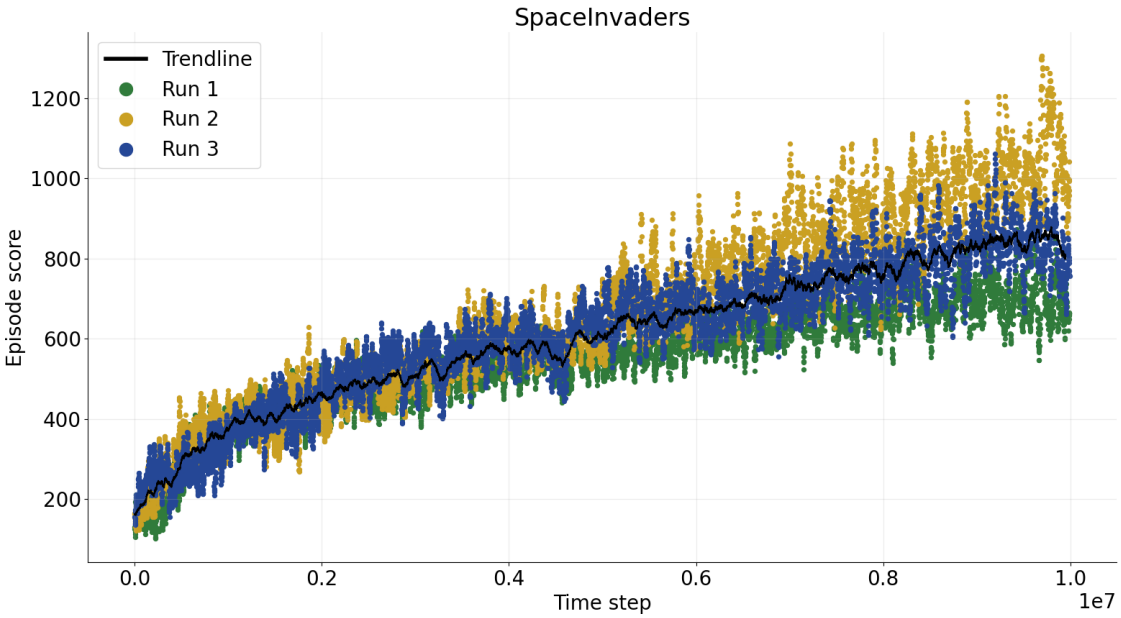

Figure 17: SpaceInvaders with the reference configuration

5.4.2. Stability with Suboptimal Hyperparameter

Schulman et al. (2017) claim that Proximal Policy Optimization is robust to hyperparameter choices. In this experiment, agents were trained with an exploration penalty instead of an exploration bonus. Figures 18–22 display reward graphs for a training with the entropy coefficient $c_2 = -0.01$. The corresponding final scores can be seen below in table 5.

Game	Run 1	Run 2	Run 3	Average of reference
BeamRider	1580.3	1040.9	701.0	2334.5
Breakout	283.81	1.8	1.2	246.9
Pong	20.4	−21.0	−21.0	20.8
Seaquest	1707.6	957.2	907.0	1476.3
SpaceInvaders	854.8	789.0	562.3	820.2

Table 5: Final scores of a training with $c_2 = -0.01$ and the average of the final scores of training runs with the reference configuration.

The results seen with this configuration are the worst among all experiments on BeamRider, Breakout and Pong. However, single runs of Breakout and Pong (cf. figures 19 and 20) still achieve performances comparable to the reference configuration, as highlighted by the magenta trendline. We further note that Seaquest and SpaceInvaders overall are not as affected by this change. This indicates that games pose different challenges and that hyperparameters have different effects depending on the respective game.

In figure 22 a steep drop in performance is visible in the first run after approximately $5.6 \cdot 10^7$ time steps. This performance collapse could be related to the policy making too large gradient steps. However, as policy updates occur within the trust region ϵ, the performance recovers to a degree. If we choose a significantly larger ϵ, the performance collapse may be unrecoverable, as PPO becomes more like simple policy gradient methods.

As some agents still learn, this implies that the algorithm is robust to the specific values chosen for c_2 to a certain extent. We further discuss this topic in chapter 5.5.3.

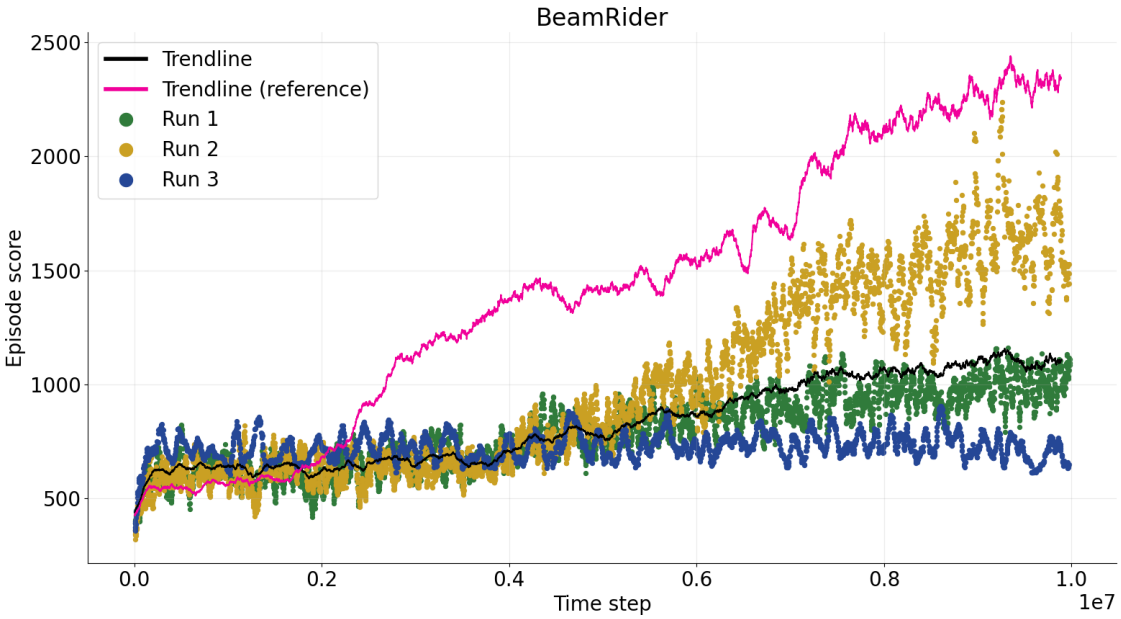

Figure 18: BeamRider with $c_2 = -0.01$

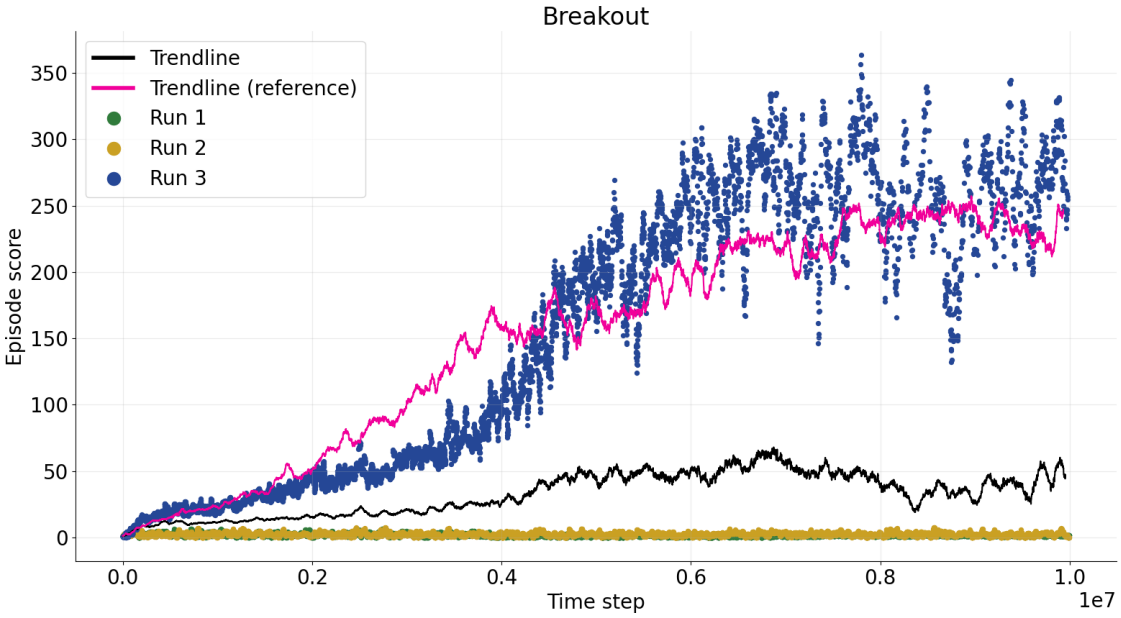

Figure 19: Breakout with $c_2 = -0.01$

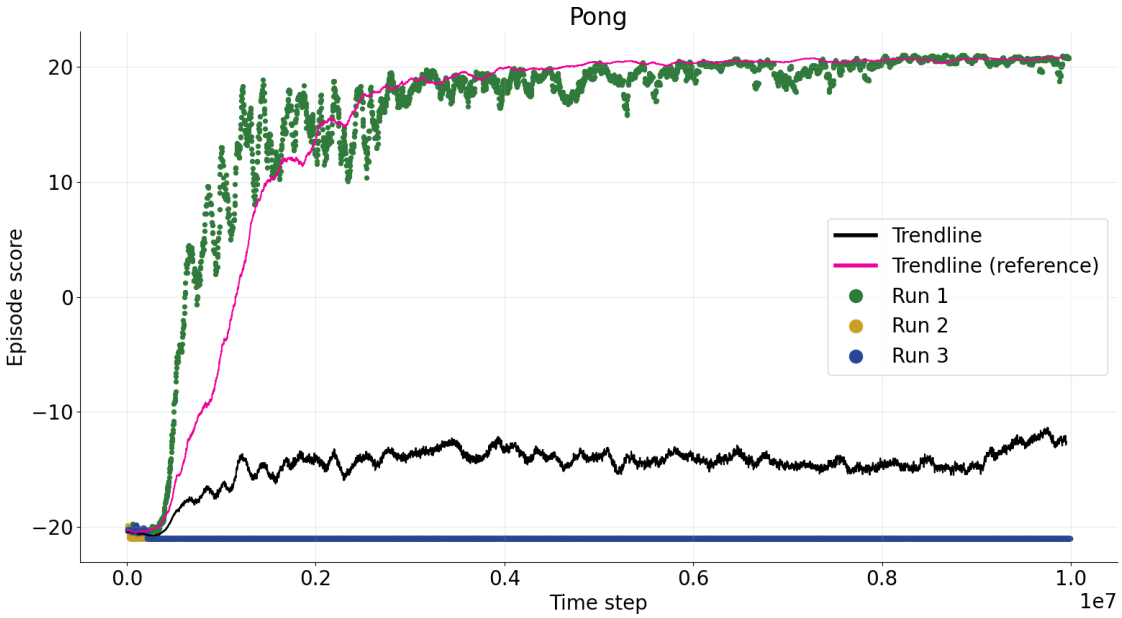

Figure 20: Pong with $c_2 = -0.01$

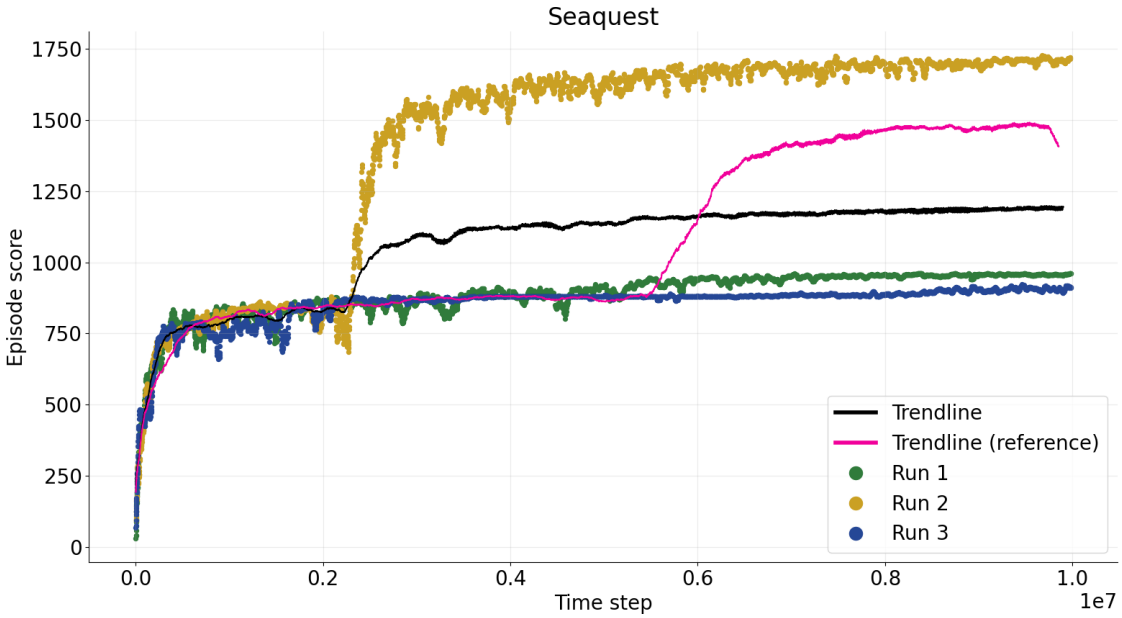

Figure 21: Seaquest with $c_2 = -0.01$

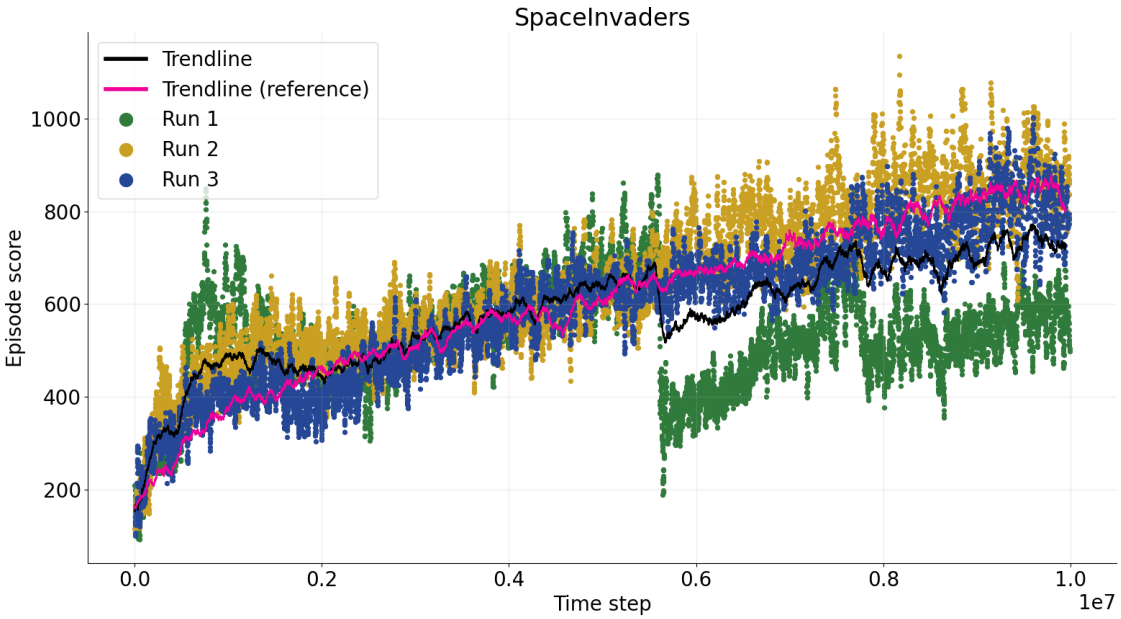

Figure 22: SpaceInvaders with $c_2 = -0.01$

5.4.3. No Optimizations

The following experiment contains no advantage normalization, no reward binning or clipping, no value function loss clipping, no gradient clipping and no orthogonal initialization. The hyperparameters are changed from the reference configuration to match the hyperparameters given by Schulman et al. (2017): $K = 3$ and $c_1 = 1.0$ instead of $K = 4$ and $c_1 = 0.5$. The reward graphs can be seen in figures 23–27; the final scores are shown in table 6.

Game	Run 1	Run 2	Run 3	Schulman et al. (2017)
BeamRider	2618.7	756.2	750.6	1590.0
Breakout	212.4	176.2	126.6	274.8
Pong	20.2	18.5	18.2	20.7
Seaquest	890.2	882.6	873.4	1204.5
SpaceInvaders	559.7	553.0	413.8	942.5

Table 6: Scores of agents trained exactly as described in the Proximal Policy Optimization publication.

Most reward graphs display noticeably lower scores than the reward graphs of the reference configuration. This is also apparent in the final scores—only the agents trained on Pong remain close to the reported performance, but it takes the agents much longer

to achieve strong performances. The results strongly deviate from the results reported by Schulman et al. (2017). Therefore, we can conclude that the optimizations outlined in chapter 4 have strong effects on the course of training as well as on the final performance of trained agents.

This result is not restricted to the original paper configuration—with one exception. As can be seen in the experiment *no_optimizations*, agents perform worse without the optimizations when trained with the reference configuration as well. The notable—and unexpected—exceptions to this are agents trained on BeamRider, which achieve noticeably improved performances. We may reason that this shows that agents trained on BeamRider are more sensitive to hyperparameter choice than they are to the optimizations. Further research is required to make conclusive statements.

However, we can confidently state that advantage normalization, gradient clipping, orthogonal initialization, reward binning and value function loss clipping combined have a pronounced positive effect on the performance of Proximal Policy Optimization in general. We further discuss this matter and the respective individual impact of the optimizations in chapter 5.5.1.

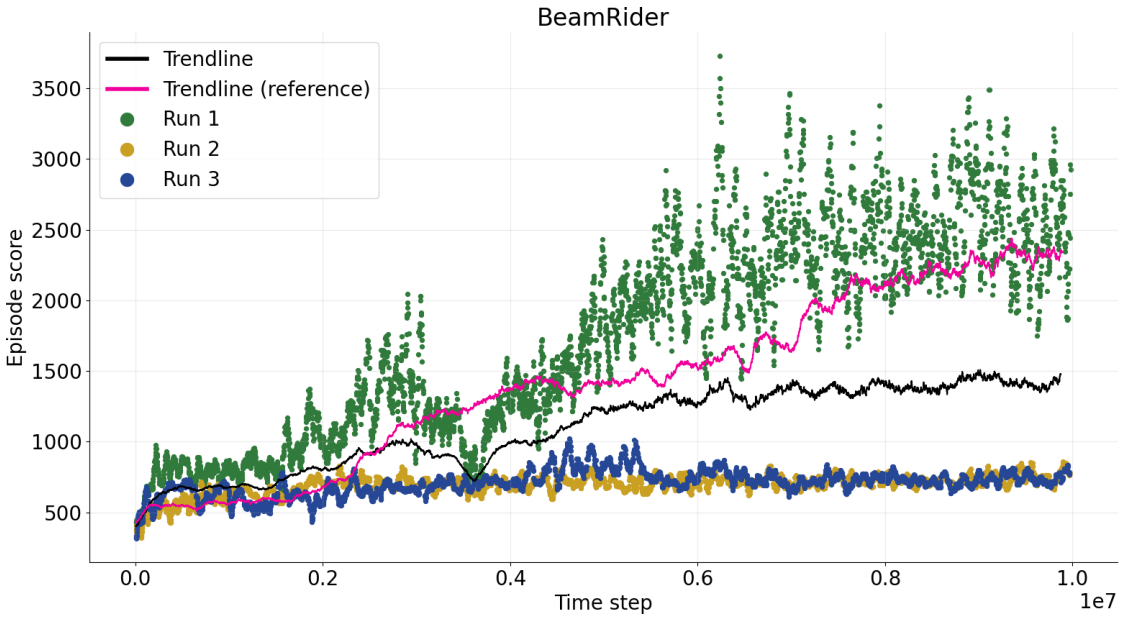

Figure 23: BeamRider trained exactly as described in the Proximal Policy Optimization publication

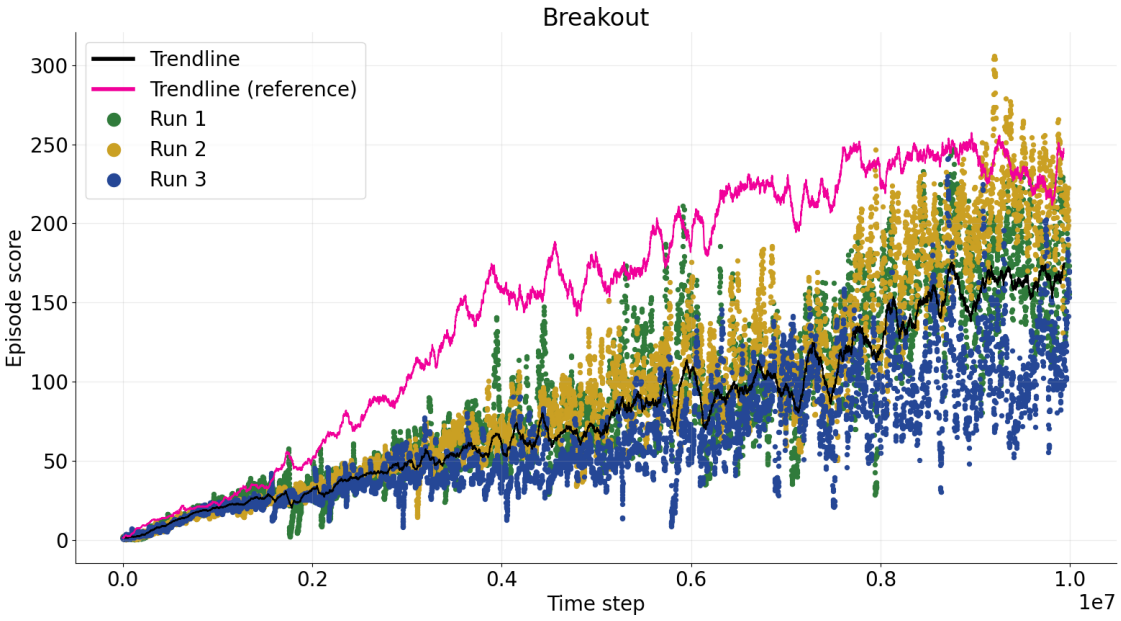

Figure 24: Breakout trained exactly as described in the Proximal Policy Optimization publication

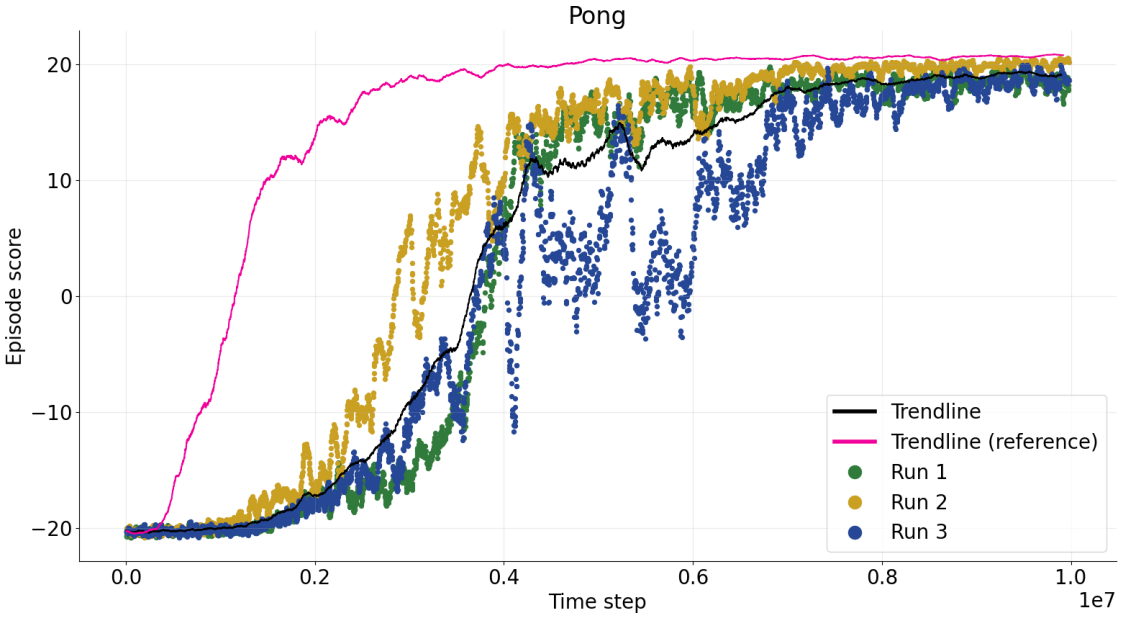

Figure 25: Pong trained exactly as described in the Proximal Policy Optimization publication

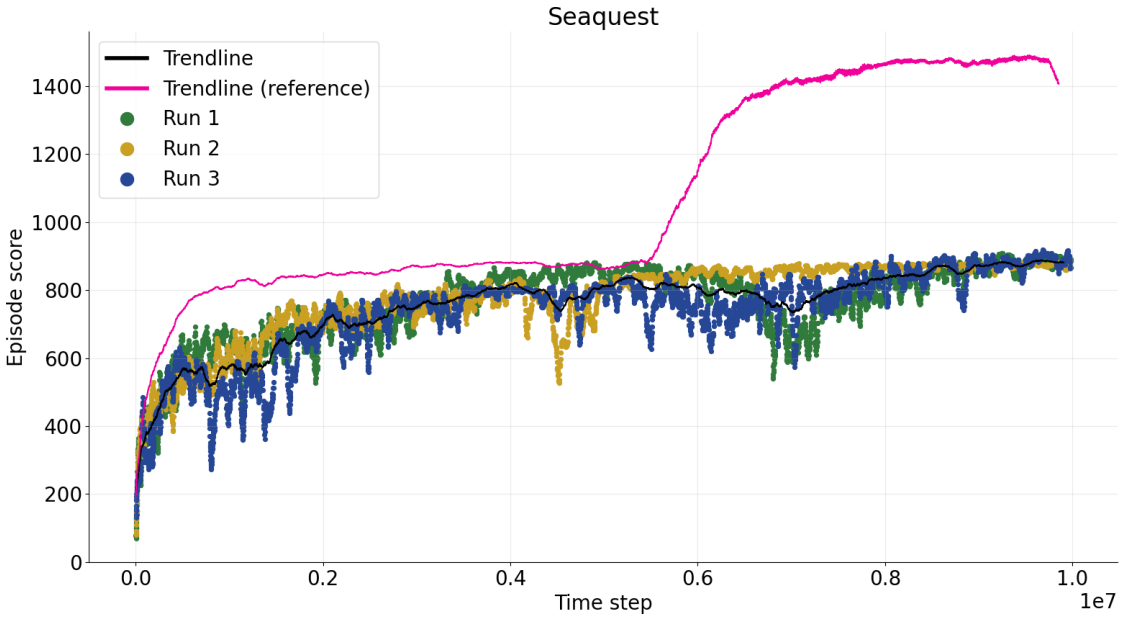

Figure 26: Seaquest trained exactly as described in the Proximal Policy Optimization publication

Figure 27: SpaceInvaders trained exactly as described in the Proximal Policy Optimization publication

5.5. Discussion

Three aspects of the experiments must be discussed. Firstly, the impact of the non-disclosed optimizations shall be evaluated. Secondly, the prevalence of outliers and the reliability of reward graphs as proposed by Schulman et al. (2017) and shown in this thesis are reviewed. Finally, we discuss the robustness of PPO to hyperparameter choices. Remember that all experiments are available online at `https://github.com/Aethiles/ppo-results`.

5.5.1. Non-disclosed Optimizations

Since the performance of agents trained with Proximal Policy Optimization as described in the publication is notably worse than the performance achieved with the reference configuration, it begs the question if all optimizations are required and what their individual impact is. In order to answer this question, agents were trained twice for each optimization. Once with only the specific optimization enabled and once with all other optimizations enabled.

Advantage normalization. Disabling advantage normalization can be seen in experiment *no_advantage_normalization*. Without advantage normalization the performance is improved on BeamRider as well as Seaquest. However, agents trained on Breakout, Pong and SpaceInvaders perform worse. If advantage optimization is the only optimization used (cf. experiment *only_advantage_normalization*), agents perform worse than agents trained without any optimization on all games but SpaceInvaders. Thus, advantage normalization appears to have little impact on the performance at best.

Gradient clipping. Gradient clipping is disabled in experiment *no_gradient_clipping*. We note that the performance on Breakout is particularly improved with a final score of roughly 400 achieved by all three agents without gradient clipping. However, some reward graphs on other games appear less stable, notably one of the training runs on BeamRider, Pong and Seaquest each. If only gradient clipping is enabled (as seen in experiment *only_gradient_clipping*), the performance on Breakout is even worse than having no optimization enabled at all. However, the performance of agents trained on BeamRider and SpaceInvaders is improved.

As gradient clipping restricts the magnitude of change of the neural network's weights, it is another device that keeps the new policy $\pi_{\boldsymbol{\theta}}$ close to the old policy $\pi_{\boldsymbol{\theta}_{\text{old}}}$. If the new policy deviates a little too far, we could see more noise, as the policy will be slightly less

optimal. However, clipping the losses with ϵ still keeps them near the old policy, so we see no total performance collapse.

Orthogonal initialization. Experiment *no_orthogonal_initialization* contains reward graphs of trainings conducted without the very optimization. Although orthogonal initialization has no notable impact on Breakout and SpaceInvaders, it is an important optimization for BeamRider, Pong and Seaquest. Performances on the latter games are worse when the neural network is not initialized with an orthogonal initialization scheme. On all three, the final score is reduced and any learning happens slower. Enabling only orthogonal initialization leads to improved results over no optimizations on Breakout, Pong, Seaquest and SpaceInvaders (cf. experiment *only_orthogonal_initialization*). Only the performance on BeamRider is diminished.

Most notably, agents learn Pong much faster when the neural network is initialized with an orthogonal initialization scheme. The results also indicate that agents cannot achieve meaningful results on Seaquest without orthogonal initialization.

Reward binning. Training runs without any form of reward binning or clipping can be seen in experiment *no_reward_binning*. The only game that is not affected by this change is Pong, as the set of rewards in Pong is $-1, 0, 1$ already. Furthermore, in Pong no player will score multiple times within $k = 4$ frames (cf. chapter 4.1).

On the remaining games, rewards can achieve much larger magnitudes so reward binning or clipping has a notable effect on the rewards. Experiment *reward_clipping* shows a distinct drop in performance on all games but Pong. This is echoed in the reward graphs and final scores with all agents performing much worse once rewards are no longer subject to reward binning. As a consequence, rewards should be binned and not clipped.

As expected, agents trained with only reward binning achieve notably improved results than agents trained without optimizations (cf. experiment *only_reward_binning*). The unexpected exception to this observation is Pong, which sees diminished results for no apparent reason.

Value function loss clipping. Agents trained without value function loss clipping show notably improved performances on Breakout but perform slightly worse on Pong as shown in the experiment *no_value_function_loss_clipping*. With only value function loss clipping enabled, the performance of agents trained on BeamRider is worse than those trained without optimizations and those trained with the reference configuration

(cf. experiment *only_value_function_loss_clipping*). On the other hand, agents trained on Breakout and SpaceInvaders perform better than agents trained without optimizations.

Conclusion. The conducted experiments show that the optimizations overall have a notable positive effect on the performance of agents. However, no general statements on the impact of an optimization can be made, as the influence of an optimization depends on the respective game an agent is trained on. For example, orthogonal initialization is shown to be crucial for agents learning Pong or Seaquest whilst the positive impact on Breakout and SpaceInvaders is notable but less pronounced.

Furthermore, optimizations likely interact with each other, which further complicates general statements based on the experiments. This can be seen with value function loss clipping on Breakout. Both the experiment with all optimizations but value function loss clipping and the experiment with only value function loss clipping show improved performances over the reference configuration and the no optimizations experiment, respectively.

Most notably, agents trained on BeamRider benefit from the optimizations when trained with the paper configuration of $K = 3$ and $c_1 = 1.0$, whereas they achieve subpar results with the reference configuration. This indicates that hyperparameter choices revolving around the value function may have a larger impact than the optimizations.

Overall, the optimizations have a positive effect on the algorithm with advantage normalization having the least impact. Hence, we must criticize that they were not disclosed by Schulman et al. (2017). Even though they are not directly related to reinforcement learning, they should be disclosed to ensure reproducibility and a transparent discussion on the performance of Proximal Policy Optimization algorithms.

5.5.2. Outliers

Figures 16 and 18 to 23 contain obvious outliers, some of which perform a lot better than other runs in that game, whereas others perform a lot worse. Among all experiments conducted for this thesis, about 35% of the graphs generated contain an obvious outlier. Similar inconsistency can be seen in the plots published by Schulman et al. (2017) for a variety of games.

With merely three runs per configuration used in evaluation, it is hard to tell if this behavior is representative. For example, we cannot determine if a single good performance must be attributed to luck or if in fact roughly a third of all trained agents achieve such a performance.

At the time of publication of Proximal Policy Optimization in 2017 OpenAI Gym contained 49 ATARI games. As of writing this thesis, this number has grown to 57 games. It seems unlikely that each of these games provides a unique challenge. Therefore, instead of training on all the games available in OpenAI Gym one could train on a specific selection of games chosen by experts to be as diverse as possible. If this subset is half or a third the size of the original set of games, we can easily perform more training runs on each game without requiring more time to conduct our experiments.

Even if no subset can be selected, performing more runs on each configuration might be a worthwhile effort. By increasing the number of training runs, we achieve more robust results. Thus, it becomes easier to discern if a run's performance is representative of the configuration under test. Moreover, we can compute more reliable trendlines that could allow for easier comparisons of configurations of the same algorithm or of entirely different algorithms.

5.5.3. Robustness to Hyperparameter Choices

Schulman et al. (2017) state that Proximal Policy Optimization is robust in terms of hyperparameter choice. The results of the experiments support this statement in general. Huge deviation from the published hyperparameters is required to impede PPO such that no or little learning occurs. This can be seen in the experiment shown in chapter 5.4.2 and with BeamRider in experiment *epsilon_0_5*. When hyperparameters remain remotely close to the original values, the performance may deviate, but agents still learn, for example when $\gamma = 0.95$ or $\lambda = 0.9$ (see experiments *gamma_0_95* or *lambda_0_9*). Therefore, it can be concluded that the reference configuration is a sensible choice for learning ATARI video games.

That does not mean that the reference configuration is the optimal configuration for any game:

- Agents perform much better on Breakout when ϵ is not annealed over the course of the training (cf. experiment *no_epsilon_annealing*).
- When the minibatch size is decreased from 256 to 64, agents learn Pong a lot faster (cf. experiment *16_minibatches*).
- Both the reward graph and the final score of Seaquest are notably improved when the learn rate α is not annealed over the course of the training (cf. experiment *no_alpha_annealing*).

However, performing such changes often leads to worse performances on at least one other game. Hence, it is hard to conclude which configuration is superior and a larger set of games may be required to draw extensive conclusions.

6. Conclusion and Future Work

6.1. Conclusion

In this thesis we introduced the necessary concepts of reinforcement learning to discuss *Proximal Policy Optimization*. Among these are an agent and an environment the agent observes as well as the Markov decision process consisting of states, actions, rewards. In addition, the Markov decision process contains a transition function for the environment the agent interacts with and a policy guiding the agent's behavior. We built upon these concepts to define the value function and the advantage function, which allow us to examine the performance of an agent following a specific policy.

In order to improve the policy, we introduced parameterization and policy gradients, which enable us to optimize a policy via stochastic gradient descent. As the environments in this thesis are specific ATARI 2600 video games, we lack full knowledge of the environments' dynamics. Thusly, we must estimate gradients by sampling training data from interaction with the environments. Simple policy gradient estimators have been shown to be unreliable, resulting in performance collapses and suboptimal solutions (Kakade & Langford, 2002).

Hence, advanced learning algorithms are required. One of these algorithms is *Proximal Policy Optimization*, which uses parallel environments to generate diverse training data (Schulman et al., 2017). An agent learns on this data for multiple epochs instead of just once. To ensure that the performance of a policy does not collapse after optimizing, Proximal Policy Optimization intends to keep the new policy close to the old policy by clipping losses that deviate too much.

However, multiple non-disclosed optimizations to the algorithm can be found in two implementations of the algorithm that are provided by the authors of the original publication (OpenAI Inc., 2017a). We examined these optimizations and popular hyperparameter choices on five ATARI 2600 games: BeamRider, Breakout, Pong, Seaquest and SpaceInvaders. The results of the experiments show that most of the optimizations are crucial to achieve good results on these ATARI games, although their specific impact depends on the game an agent is trained on. This finding is shared by Ilyas et al. (2018) and Engstrom et al. (2019), who examined Proximal Policy Optimization on robotics environments. Furthermore, we found that noticeable outliers are present in approximately 35% of the experiments.

Therefore, we may call for two improvements: Firstly, all optimization choices pertaining to the algorithm—even those not purely related to reinforcement learning—must be published. In order to ensure proper reproducibility and comparability, methods used

for the portrayal of results, such as the smoothing of graphs, must be described as well. Secondly, instead of testing each configuration thrice on every game available, performing more runs on an expert-picked set of diverse games grants more reliable results. With more training runs, it becomes easier to discern if a striking performance is to be expected or simply a result of chance.

6.2. Future Work

Based on the work conducted for this thesis, multiple venues for future work present themselves—in the following three possible topics are highlighted. First, we could work on improved evaluation methods that allow for easier comparisons of algorithms across publications. This includes increasing the number of training runs executed and potentially selecting a diverse subset of games, so robust trendlines may be created and shared. Furthermore, we may examine advanced regression methods and evaluate if these methods can be used to create more suitable baselines. Lastly, this also includes devising a proper means of determining the noise of a reward graph, for example by displaying a standard deviation or confidence intervals around the trendline.

Second, the code written for this thesis can be adjusted to support other benchmarking environments. Most importantly, by supporting robotics and control tasks, we could reproduce the findings of Ilyas et al. (2018) and Engstrom et al. (2019). Moreover, we could also reproduce the findings of modified Proximal Policy Optimization algorithms such as Truly Proximal Policy Optimization (Wang et al., 2019). A number of improvements to Proximal Policy Optimization have been suggested by researchers, so a survey comparing these might yield further insight into which modifications are suitable and which may not be as promising.

Finally, Proximal Policy Optimization can be combined with further learning techniques. In particular, curiosity-driven learning as detailed by Pathak et al. (2017) is an interesting technique to combine with PPO. Instead of having an agent explore by simply picking random actions, the agent attempts to predict and learn transitions to successor states. The agent is then rewarded if it cannot predict the successor state, which incentivizes it to observe novel states more often. The combination of this intrinsic reward with the extrinsic reward returned by the environment is an interesting topic for further research.

List of Mathematical Symbols and Definitions

$\doteq$	defined to be	
$\sum_{s',r}$	shorthand for $\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}}$	
$\#M$	cardinality of the set M	
$\propto$	proportional to	
$a, s \sim \pi$	a, s observed by following π	
$\mathbb{E}$	expected value	
$\mathbb{E}_{a,s\sim\pi}$	expected value with observed a, s	
t	discrete time step	(chapters 2.2.1 and 2.2.4)
T	final time step, horizon	(chapter 2.2.4)
s, s'	states	(chapter 2.2.2)
a	action	(chapter 2.2.2)
r	reward	(chapter 2.2.2)
S_t	state at time t	(chapter 2.2.2)
A_t	action at time t	(chapter 2.2.2)
R_{t+1}	reward at time $t+1$	(chapter 2.2.2)
$\mathcal{S}$	set of states	(chapter 2.2.2)
$\mathcal{A}$	set of actions	(chapter 2.2.2)
$\mathcal{R}$	set of rewards	(chapter 2.2.2)
$p(s', r \mid s, a)$	dynamics function	(chapter 2.2.3, eq. 1)
$\pi(a \mid s)$	policy	(chapter 2.2.3, eq. 2)
α	learn rate	(chapter 2.5.2)
γ	discount factor	(chapter 2.3.2)
λ	variance tuning factor	(chapter 3.2.1)
ϵ	clipping parameter	(chapter 3.3.2)
G_t	return	(chapter 2.3.1, eq. 3)
G_t^λ	λ-return	(chapter 3.2.2, eq. 22)
$v_\pi(s)$	value function	(chapter 2.3.2, eq. 4)
$a_\pi(s, a)$	advantage function	(chapter 2.3.3, eq. 5)
δ_t	advantage estimator	(chapter 2.3.3, eq. 7)
$\delta_t^{\mathrm{GAE}(\gamma,\lambda)}$	Generalized Advantage Estimation	(chapter 3.2.1, eq. 21)

$\boldsymbol{\omega}$	parameter vector	(chapter 2.4
$\hat{v}_\pi(s, \boldsymbol{\omega})$	parameterized value function	(chapter 2.4)
$\hat{a}_\pi(s, a, \boldsymbol{\omega})$	parameterized advantage function	(chapter 2.4, eq. 9)
$\mu(s)$	stationary distribution of states	(chapter 2.4)
$\overline{\text{VE}}(\boldsymbol{\omega})$	mean squared value error	(chapter 2.4, eq. 8)
$\boldsymbol{\theta}$	parameter vector	(chapter 2.5.1)
$\pi(a \mid s, \boldsymbol{\theta})$	parameterized policy	(chapter 2.5.1)
$J(\boldsymbol{\theta})$	fitness function	(chapter 2.5.2)
$\hat{g}$	gradient estimator	(chapter 2.5.2)
$\rho_t(\boldsymbol{\theta})$	likelihood ratio	(chapter 3.3.2, eq. 23)
$\text{clip}(\rho_t(\boldsymbol{\theta}), \epsilon)$	clipping function	(chapter 3.3.2, eq. 25)
$\text{clip}_v(\boldsymbol{\omega}, \boldsymbol{\omega}_{\text{old}}, \epsilon, S_t)$	value clipping function	(chapter 4.5, eq. 31)
$\mathcal{L}(\boldsymbol{\theta})$	loss	(chapter 3, eq. 17)
$\mathcal{L}^{\text{CLIP}}(\boldsymbol{\theta})$	PPO clipped loss	(chapter 3.3.2, eq. 26)
$\mathcal{L}^{\text{VF}}(\boldsymbol{\omega})$	value function loss	(chapter 3.3.3, eq. 27
$\mathcal{L}^{\text{VFCLIP}}(\boldsymbol{\omega})$	clipped value function loss	(chapter 4.5, eq. 32)
c_1	value function loss coefficient	(chapter 3.3.4)
c_2	entropy bonus coefficient	(chapter 3.3.4)
S	entropy bonus	(chapter 3.3.4)
$\mathcal{L}^{\text{CLIP+VFCLIP+}S}(\boldsymbol{\omega})$	shared parameters PPO loss	(chapter 3.3.4, eq. 28)
τ	rollout	(chapter 3.4)
ϕ	environment adaptation	(chapter 4.1)
k	number of frames skipped	(chapter 4.1)
N	number of parallel actors	(chapter 3.4)
I	number of training iterations	(chapter 3.4)
K	number of epochs	(chapter 3.4)
M	minibatch size	(chapter 3.4)

Glossary

action

chosen by the agent to interact with the environment. Denoted a or A_t.

actor

denotes an agent that is run in parallel in multiple instances of the same environment.

advantage

commonly used to denote a single advantage estimation or value of the advantage function.

advantage function

determines if an action yields higher or lower reward than the expected behavior would.

agent

the acting and learning entity.

ALE

see *Arcade Learning Environment*.

Arcade Learning Environment

a framework for benchmarking reinforcement learning algorithms on ATARI 2600 games.

dynamics

a probability distribution that determines the environment's behavior. Denoted p.

entropy bonus

a component of the loss that encourages exploration.

environment

the world surrounding the agent.

episode

the sequence from an initial state to a terminal state. In this thesis an episode usually means one attempt at a game.

episode reward graph

a graph that plots time steps on the x axis and the cumulated rewards of terminated episodes on the y axis.

exploitation

denotes choosing optimal actions to strengthen the knowledge of these actions.

exploration

denotes choosing suboptimal actions that the agent has little or no knowledge of yet.

fitness function

a metric that shall be optimized. Denoted J.

GAE

see *Generalized Advantage Estimation*.

Generalized Advantage Estimation

a sophisticated advantage estimator that allows the tuning of bias versus variance.

gradient ascent/descent

a method for maximizing/minimizing multidimensional optimization problems.

Gym

see *OpenAI Gym*.

horizon

the final time step of an episode.

hyperparameter

a configuration parameter that is not learned by the agent.

learn rate

controls the step size in gradient ascent or descent.

loss

the objective of minimization or maximization.

Markov decision process

a stochastic process containing states, actions and rewards as well as probability distributions determining the behavior of the agent and the environment.

OpenAI Gym

a framework that includes several benchmark environments, e.g., robotics tasks and the Arcade Learning Environment.

policy

a probability distribution that determines the agent's behavior. Denoted π.

PPO

see *Proximal Policy Optimization*.

Proximal Policy Optimization

a class of deep reinforcement learning algorithms. The specific version used in this thesis is called PPO clip.

return

a sum of rewards seen after a given time step.

reward

observed by the agent following an action. The agent is trained to maximize the observed rewards. Denoted r or R_{t+1}.

rollout

a sequence of states, actions, rewards, action probabilities and values generated from interaction of the agent with the environment. Denoted τ.

state

describes the environment. Denoted s, s' or S_t.

surrogate objective

an objective that approximates the true objective, e.g., by posing a lower bound on improvement.

tensor

a multi-dimensional vector (or array, in computer science terms).

terminal state

an absorbing state that the agent cannot leave. Transitioning to a terminal state marks the end of an episode.

trajectory

a sequence of states, actions and rewards generated from interaction of the agent with the environment.

trust region

a region around the old policy that should allow for safe policy updates, so the performance of the new policy does not collapse.

References

Anand, A. S., Zhao, G., Roth, H., & Seyfarth, A. (2019). A deep reinforcement learning based approach towards generating human walking behavior with a neuromuscular model. *2019 IEEE-RAS 19th International Conference on Humanoid Robots (Humanoids)*, 537–543.

Baird, L. C. (1993). Advantage updating. *Wright Laboratory Technical Report*.

Bellemare, M., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, *47*, 253–279.

Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., ... Zhang, S. (2019). Dota 2 with large scale deep reinforcement learning. *arXiv*, *abs/1912.06680v1*.

Brockman, G., Cheung, V., Petterson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *arXiv*, *abs/1606.01540v1*.

Chen, S. (2018). *Comparing deep reinforcement learning methods for engineering applications* (Master's thesis, Otto-von-Guericke-Universität Magdeburg). Retrieved 2020-05-29, from `https://www.ci.ovgu.de/is_media/Master+und+Bachelor_Arbeiten/MasterThesis_ShengnanChen_2018-p-4774.pdf`

Dai, T., Dubois, M., Arulkumaran, K., Campbell, J., Bass, C., Billot, B., ... Bharath, A. A. (2019). Deep reinforcement learning for subpixel neural tracking. *Medical Imaging with Deep Learning*.

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., & Madry, A. (2019). Implementation matters in deep policy gradients: A case study on ppo and trpo. *International Conference on Learning Representations*.

Gaudet, B., Linares, R., & Furfaro, R. (2020). Deep reinforcement learning for six degree-of-freedom planetary landing. *Advances in Space Research*, *65*, 1723–1741.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press. (ISBN: 9780262035613)

Grinstead, C. M., & Snell, J. L. (1997). *Introduction to probability*. American Mathematical Society. (ISBN: 9780821807491)

Güldenring, R. (2019). *Applying deep reinforcement learning in the navigation of mobile robots in static and dynamic environments* (Master's thesis, Universität Hamburg). Retrieved 2020-05-29, from `https://tams.informatik.uni-hamburg.de/publications/2019/MSc_Ronja_Gueldenring.pdf`

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. *arXiv*, *abs/1710.02298v1*.

Hunter, D. R., & Lange, K. (2004). A tutorial on mm algorithms. *The American Stastistician*, *58*, 30–37.

Ilyas, A., Engstrom, L., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., & Madry, A. (2018). Are deep policy gradient algorithms truly policy gradient algorithms? *arXiv*, *abs/1811.02553v3*.

Ilyas, A., Engstrom, L., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., & Madry, A. (2020). A closer look at deep policy gradients. *International Conference on Machine Learning*.

Jayasiri, V. (n.d.). *Proximal policy optimization algorithms - ppo in pytorch.* Retrieved 2020-05-24, from `https://blog.varunajayasiri.com/ml/ppo_pytorch.html`

Kakade, S., & Langford, J. (2002). Approximately optimal approximate reinforcement learning. *International Conference on Machine Learning*, *19*.

Kostrikov, I. (2018). *pytorch-a2c-ppo-acktr.* Retrieved 2020-05-08, from `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`

Liu, H., Socher, R., & Xiong, C. (2019). Taming maml: Efficient unbiased meta-reinforcement learning. *International Conference on Machine Learning*.

Miller, D., Englander, J. A., & Linares, R. (2019). Interplanetary low-thrust design using proximal policy optimization. *AAS/AIAA Astrodynamics Specialist Conference*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., ... Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *Computing Research Repository*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *Advances in Neural Information Processing Systems*. (Workshop paper)

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.

OpenAI Inc. (2017a). *Openai baselines.* Retrieved 2020-03-26, from `https://github.com/openai/baselines`

OpenAI Inc. (2017b). *Proximal policy optimization.* Retrieved 2020-04-30, from `https://openai.com/blog/openai-baselines-ppo`

OpenAI Inc. (2018). *Openai spinning up.* Retrieved 2020-05-04, from `https://spinningup.openai.com/en/latest/algorithms/trpo.html#background`

Pathak, D., Agrawal, P., Efros, A. A., & Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. *International Conference on Machine Learning*, *34*.

Peng, X. B., Abbeel, P., Levine, S., & van de Panne, M. (2018). Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics*, *37*(4), 143:1–143:14.

PyTorch Contributors. (2019). *Pytorch 1.5.0 documentation.* Retrieved 2020-05-26, from `https://pytorch.org/docs/stable/notes/randomness.html`

Saxe, A., McClelland, J., & Ganguli, S. (2014). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *International Conference on Learning Representations*.

Schulman, J., Levine, S., Moritz, P., Jordan, M., & Abbeel, P. (2015). Trust region policy optimization. *International Conference on Machine Learning*, *37*, 1889–1897.

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. *International Conference on Learning Representations*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv*, *abs/1707.06347v2*.

Seita, D. (2017). *Going deeper into reinforcement learning: Fundamentals of policy gradients.* Retrieved 2020-05-04, from `https://danieltakeshi.github.io/2017/03/28/going-deeper-into-reinforcement-learning-fundamentals-of-policy-gradients`

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Driessche, G., ... Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*, 484-489.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). MIT Press. (ISBN: 9780262039246)

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems*, *12*.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., ... Silver, D. (2019). Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, *575*, 350–354.

Wang, Y., He, H., & Tan, X. (2019). Truly proximal policy optimization. *Uncertainty in Artificial Intelligence*, *35*.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*, 229–256.

A. List of Experiments

Table 8 contains an overview of all experiments and a short summary of the observation. The corresponding graphs are available at `https://github.com/Aethiles/ppo-results`. If a game is not mentioned in the observations column, the graphs show no conclusive deviation from the reference graphs. In the *only_<name>* experiments, the reference trendline is replaced with a trendline of the *no_optimizations* experiment.

Experiment	Observations
3 epochs[24]	slightly improved on Breakout and SpaceInvaders
4 actors	worse on BeamRider; improved on Seaquest
16 actors	improved on BeamRider; worse on Pong and Seaquest
16 actors, Minibatch size 128	improved on BeamRider, Pong, Seaquest; worse on SpaceInvaders
Minibatch size 32	notably worse on BeamRider; worse on Breakout and SpaceInvaders; improved on Seaquest; faster learning on Pong but less stable after $4 \cdot 10^6$ time steps
Minibatch size 64	notably worse on BeamRider; improved on Pong and Seaquest
Minibatch size 128	notably worse on BeamRider with three very different runs; improved on Seaquest
Minibatch size 32, ϵ not annealed	notably worse on BeamRider and SpaceInvaders; improved on Pong and Seaquest
Minibatch size 64, ϵ not annealed	worse on BeamRider; improved on Breakout, Pong and Seaquest
Minibatch size 128, ϵ not annealed	improved on Breakout, Pong and Seaquest
α and ϵ not annealed	worse on BeamRider; noisier on Pong; improved on Seaquest
α not annealed	worse on BeamRider; notably improved on Seaquest[25]
ϵ not annealed	less stable on BeamRider; notably improved on Breakout
$\alpha = 2.5 \cdot 10^{-3}$	no learning on BeamRider and two runs of Breakout; notably worse on one run of Breakout, Seaquest and SpaceInvaders
$\alpha = 2.5 \cdot 10^{-5}$	notably worse to no learning on all games

[24] A common choice to reduce training time.
[25] As Seaquest often features prominent outliers, it cannot be concluded that this configuration is better for Seaquest.

Experiment	Observations
$\epsilon = 0.05$	improved on Seaquest; worse on SpaceInvaders with three very different runs
$\epsilon = 0.2$	improved on Breakout; noisier and worse on Pong
$\epsilon = 0.5$	worse on all games; noisier on Pong and possibly on Breakout
$\gamma = 0.9$	notably worse on all games
$\gamma = 0.95$	notably worse on BeamRider, Seaquest and SpaceInvaders
$\lambda = 0.9$	improved on Breakout
$\lambda = 0.99$	worse on BeamRider, Breakout and SpaceInvaders
$c_1 = 0.2$	no learning on Breakout; improved on Seaquest; performance collapse in one run of SpaceInvaders
$c_1 = 1$	improved on BeamRider and Seaquest
$c_1 = 1$, $\epsilon = 0.2$	worse on BeamRider; slightly improved on Breakout
$c_2 = -0.01$	notably worse to no learning on all games but SpaceInvaders; outliers may still achieve reference performance
$c_2 = 0$	notably worse and noisier on Pong, worse on Seaquest
$c_2 = 0.1$	worse on BeamRider and Pong; notably worse on Breakout; improved on Seaquest
paper configuration	improved on BeamRider, Breakout and Seaquest
no optimizations, paper configuration	notably worse on all games
no optimizations	notably worse on all games but BeamRider; notably improved on BeamRider[26]
no advantage normalization	improved on BeamRider and Seaquest; slightly worse on Breakout and Pong; slightly improved on SpaceInvaders
only advantage normalization	worse than no optimizations on all games but SpaceInvaders
no gradient clipping	worse on BeamRider; notably improved on Breakout and Seaquest
only gradient clipping	improved on all games but BeamRider compared to no optimizations; slightly worse than no optimizations on BeamRider
no orthogonal initialization	worse on BeamRider, Pong and Seaquest

[26] This result is unexpected and merits further investigation.

Experiment	Observations
only orthogonal initialization	better than no optimizations on all games but BeamRider; worse than no optimizations on BeamRider
no reward binning	notably worse on all games but Pong
only reward binning	notably better than no optimizations on all games but Pong; worse than no optimizations on Pong[27]
no value function loss clipping	notably improved on Breakout; slightly worse on Pong
only value function loss clipping	worse than no optimizations on BeamRider; better than no optimizations on Breakout and SpaceInvaders
reward clipping	improved on BeamRider; worse on Breakout, Seaquest and SpaceInvaders
no input scaling	no learning on BeamRider; better on Seaquest; worse on SpaceInvaders
adam epsilon[28] 10^{-8}	slightly improved on Seaquest
conv3 32 filters[29]	slightly worse on BeamRider; slightly improved on Seaquest
small net[30]	worse on all games but BeamRider

Table 8: This table contains a list of all experiments and a short summary of the observations.

[27] This result is unexpected because reward binning should have no effect on the reward signal of Pong. Further research is required.
[28] A parameter for numerical stability.
[29] Architectural change by Kostrikov (2018).
[30] An architecture that has only two convolutional layers, the first with 16 filters and the second with 32 (Mnih et al., 2013).

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen und ist noch nicht veröffentlicht worden. Alle zu deren Beurteilung verwendeten (auch elektronischen) Kopien dieser Arbeit haben genau den gleichen Inhalt wie dieses gedruckte Exemplar. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Nordwalde, den 07. Juni 2020