university of groningen

faculty of science
and engineering

Towards Delivering a Coherent Self-Contained Explanation of Proximal Policy Optimization

Master's Research Project

Daniel Bick
daniel.bick@live.de

August 15, 2021

Supervisors: Prof. dr. H. Jaeger (Artificial Intelligence, University of Groningen),
dr. M.A. Wiering (Artificial Intelligence, University of Groningen)

Artificial Intelligence
University of Groningen, The Netherlands

Abstract

Reinforcement Learning (RL), and these days particularly Deep Reinforcement Learning (DRL), is concerned with the development, study, and application of algorithms that are designed to accomplish some arbitrary task by learning a decision-making strategy that aims for maximizing a cumulative performance measure. While this class of machine learning algorithms has become increasingly successful on a variety of tasks over the last years, some of the algorithms developed in this field are sub-optimally documented. One example of a DRL algorithm being sub-optimally documented is Proximal Policy Optimization (PPO), which is a so-called model-free policy gradient method (PGM). Since PPO is a state-of-the-art representative of the important class of PGMs, but can hardly be understood from only consulting the paper having introduced it, this report aims for explaining PPO in detail. Thereby, the report shines a light on many concepts generalizing to the wider field of PGMs. Also, a reference implementation of PPO has been developed, which will shortly be introduced and evaluated. Lastly, this report examines the limitations of PPO and quickly touches upon the topic of whether DRL might lead to the emergence of General Artificial Intelligence in the future.

Contents

1 Introduction

Reinforcement Learning (RL) refers to a class of machine learning algorithms which can be trained by repeatedly being told how good or bad their recent behavior in their environment has been. Those algorithms can be trained without (or, depending on the point of view, *minimal* [1]) prior knowledge of the task they may have to accomplish in the future [2]. A sometimes very valuable aspect is that RL algorithms, or *agents*, may be trained without there being a clear definition of the task to be learned by an agent. The only requirements for training an RL agent encompass an environment that the agent can observe and interact with, as well as the numeric so-called *reward signal* telling the agent about the appropriateness of its recent behavior [3]. A more formal definition of RL will be given in the succeeding section. For now it suffices to say that the essence of training an RL agent is to learn some function, called *policy*, which tells the agent which action to perform in any given state the agent encounters in its environment [3]. The policy is trained so as to maximize the sum of rewards that the agent expects to obtain in the future given its choice of action in every state the agent encounters [2, 3].

In the past, various kinds of functions have been used as policies. In some approaches, policies were realized as look-up tables [3], while other approaches used linear functions combined with a set of hand-crafted features that were used to encode information from raw state representations observed by an agent, as reported in [2]. However, these approaches were susceptible to too few or poor hand-crafted feature representations being provided [2].

In 2013, a team at DeepMind successfully demonstrated for the first time that it was possible to successfully train RL agents, whose policies were realized as artificial neural networks (NNs) [2], thereby motivating a lot of research into this direction. Since the NNs commonly being employed in this field are so-called deep NN architectures, the resulting approach to RL was named Deep Reinforcement Learning (DRL). In this paper, the two terms RL and DRL will commonly be used interchangeably, since DRL has become the predominant approach to RL. Note that DRL architectures commonly do not rely on the use of pre-defined, hand-crafted feature representations, but commonly learn themselves to extract useful features from provided raw state representations throughout the course of training using stochastic gradient descent [2].

Since its introduction, a lot of research has been conducted in the field of DRL. Recurring themes in the study of DRL are variations in how agents map from states to actions [4, 2] and whether they learn some explicit representation of their environment [5] or not [6]. Other lines of research are concerned with the question how to balance an agent's curiosity for exploring new states and the agent's tendency to exploit the behavior that the agent has learned already over time [7, 2, 8].

Some of the major achievements in the field of DRL are as follows. In 2013, the authors of [2] demonstrated for the first time that it was possible to train DRL agents on playing Atari games [9] without these agents having any prior knowledge about the games they were trained on. Later, a follow-up publication on this research was published in *Nature* [1]. These agents' policy networks consumed sequences of raw images representing a game's screen over the last few time-steps and produced corresponding Q-values, i.e. estimates of how good each available action would be in a given state. According to [1], the trained agents achieved game playing performance that was comparable to that of trained human players across 49 different Atari games. Note that these agents used Convolutional Neural Networks [10] as their policy networks and that they were trained end-to-end using stochastic gradient descent and learning principles from the field of RL. Partly drawing upon DRL, in 2016 for the first time an agent excelled in the game of Go, beating even human expert players [11]. Later, in 2017, this approach was improved to work even in the absence of human knowledge during training [12] and, in 2018, generalized to also work for other games like chess and shogi [13]. In 2019, a DRL agent was taught to master the game Dota 2 [14].

In spite of all the great progress made in the field of DRL over the last years, some of the papers published in this field suffer from sub-optimally documented methods and/or algorithms they propose or utilize. In order to understand the content of those publications, a substantial amount of prior knowledge from the field of DRL is required.

One example of a paper proposing a sub-optimally documented DRL algorithm is the one proposing the Proximal Policy Optimization (PPO) [6] algorithm. While the PPO algorithm itself is still, even years after its invention, a state-of-the-art DRL algorithm [15], understanding it from the paper proposing it, [6], requires extensive prior knowledge in the field of DRL.

PPO is a so-called policy gradient method (PGM) [6], which means that the agent's policy maps directly from state representations to actions to be performed in an experienced state [4, 16]. PPO's main demarcating feature is its objective function on which stochastic gradient ascent is performed in order to train the policy network being either a NN or RNN [6]. The objective function is special in that it is designed to allow for multiple epochs of weight updates on freshly sampled training data, which has commonly been associated with training instablities in many other policy gradient methods [6]. Thereby, PPO allows for training data being used more efficiently than in many other previous PGM methods [6].

In order to make the powerful PPO algorithm, as well as related methods, more accessible to a wider audience, this report focuses on providing a comprehensible and self-contained explanation of the PPO algorithm and its underlying methodology. When I set out to compile the contents for this paper, the task appeared seemingly simple. While it was clear from the beginning that some background-research had to be conducted to explain PPO in sufficiently more detail than in the original paper, it was not expected how difficult compiling the following contents would eventually turn out to be. While the original paper on PPO largely focuses on some particularities of PPO and how PPO is different from some related DRL algorithms, the aforementioned paper seemingly assumes a reader's complete knowledge about the general working of policy gradient methods (PGMs), thus not making any serious effort in explaining the fundamental procedure upon which PPO rests. Consequently, a lot of my effort had to be spent on understanding the field of PGMs in the first place, before being able to distil a clear picture of how PPO is different from other vanilla PGMs. In this way, large parts about the working of PPO became apparent. Last but not least, some final uncertainties concerning the working of PPO had to be ruled out by consulting a provided reference implementation of PPO[1] offered by OpenAI.

In the following, Section 2 formally introduces Reinforcement Learning, as well as Policy Gradient Methods, and establishes the basic notation used throughout the rest of this report. In Section 3, the PPO algorithm will be presented in minute detail. A custom reference implementation will be introduced in Section 4, followed by Section 5 discussing the algorithm, its shortcomings, potential improvements, and the question whether the methods presented in this paper might lead to the emergence of General Atificial Intelligence at some point in the future. This paper concludes with Section 6.

2 Preliminaries

This section will introduce some of the preliminary knowledge needed for understanding the working of the PPO algorithm. First, Reinforcement Learning (RL) and Deep RL (DRL) will be introduced in Section 2.1. Afterwards, policy gradient methods (PGMs) will be introduced in Section 2.2.

2.1 Reinforcement Learning

In Reinforcement Learning (RL), an algorithm, or *agent*, learns from interactions with its environment how to behave in the given environment in order to maximize some cumulative reward metric [3]. An agent's decision-making strategy, which is also called *policy* and from which the agent's behavior directly follows, defines the way how the agent maps perceived environmental states to actions to be performed in these perceived states [3]. An environmental state, or simply *state*, refers to a representation of the current situation in the environment that an agent finds itself in during a given discrete time step [3]. While distinctions can be made with respect to whether an agent directly perceives raw state representations or only certain (partial) observations thereof [17], this report will not distinguish these two cases, always assuming that an agent has access to state representations fully describing current environmental states. Using its decision-making strategy, i.e. policy, the agent selects in every state it encounters an action to be performed in the given state [3]. Upon executing a selected action in a given state, the agent transitions into the succeeding state and receives some numeric reward indicating how desirable taking the chosen action in the given state was

[1] https://github.com/openai/baselines/tree/master/baselines/ppo1

[3]. Each action may not only affect immediate rewards obtained by the agent, but potentially also future rewards [3]. In this report, it is assumed that episodes, or sequences, of interactions between an agent and its environment, so-called *trajectories*, are always of a finite maximal length T. Training an RL agent involves the repeated application of two successive training steps. During the former training step, the agent is made to interact with its environment for a given number of trajectories. Relevant information, such as transitions from one state to the next, the actions chosen by the agent, as well as the corresponding rewards emitted by the environment, are recorded for the latter training step. In the latter training step, the agent's policy gets updated using information extracted from the data collected in the former training step. The goal of this procedure is to update the agent's policy so as to maximize the cumulative rewards that the agent expects to receive from the environment over time given the agent's choice of action in every environmental state the agent encounters [3].

Formally, an RL agent is situated in an environment $\mathcal{E}$, which the agent perceives via state representations, or *states*, s_t, at discrete time steps t [3]. Each state $s_t \in \mathcal{S}$ is drawn from a possibly infinite set of possible states $\mathcal{S}$ [17], which describe the various environmental situations an agent might find itself in given the agent's environment $\mathcal{E}$ and its means of perceiving said environment. At any given time step t, an agent is assumed to be situated in some perceived state s_t, where it has to choose which action a_t to perform. Upon performing action $a_t \in \mathcal{A}$ in state s_t, the agent transitions into the next state $s_{t+1} \in \mathcal{S}$ and receives some reward $r_t \in \mathbb{R}$ [3, 4]. Here, $\mathcal{A}$ defines the *action space*, i.e. the set of all possible actions a_t, which an agent may choose to perform in a given state s_t. For the sake of simplicity, it is generally assumed that an action space $\mathcal{A}$ remains constant across all possible states $s_t \in \mathcal{S}$. Transitions from one state, s_t, to the next, s_{t+1}, are assumed to happen stochastically and the sampling procedure can mathematically be denoted as $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$ [17]. The stochasticity involved in the sampling of next states is governed by a so-called *transition probability distribution* defined as $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}_{\geq 0}$ [17]. This transition probability distribution determines for each possible state the conditional probability mass or density (depending on the nature of the state space) values of being stochastically sampled as the next state given the current state, s_t, and the action, a_t, chosen by the agent in that state s_t [3]. Rewards r_t are assigned to an agent through some *reward function* defined as $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ [17]. Note that also alternative definitions of reward functions, for example definitions involving stochasticity as presented in [3], are possible. Each initial state, to which an agent is exposed immediately after the initialization of its environment, is stochastically drawn from the state space $\mathcal{S}$ in accordance with a so-called *initial state distribution*, which is defined as $\rho_1 : \mathcal{S} \rightarrow \mathbb{R}_{\geq 0}$ [17] and assigns probability values to the elements in $\mathcal{S}$, indicating how probable it is for any given state to be chosen as an initial state. Concerning the notation of discrete time steps, note that this report always starts counting time steps at $t = 1$. Discrete time steps t in an agent's environment thus range in $[1, 2, ..., T]$. Recall that T denotes the finite maximal length of a given trajectory, and thus the finite maximal number of times that an agent can transition from one state to the next by choosing some action in the former state. Speaking about trajectories, those can more formally be defined as $\tau = (s_1, a_1, ..., s_T, a_T, s_{T+1})$, where state s_{T+1} is only observed as a final response to the T'th action, a_T, chosen in the T'th state, s_T, at time step $t = T$.

The formalism described above yields the setup of a *finite-horizon discounted Markov Decision Process* as long as the cumulative future rewards that the agent expects to receive over time by following its policy are discounted by some discount factor $\gamma \in (0, 1]$ [17]. Dealing with a Markov Decision Process (MDP) has the following implication: All relevant information an agent needs to have access to, in order to make educated future predictions, must be determined by nothing but knowledge of the current state and the possible actions that the agent might take in the current state. Thus, an agent is not required to have memory capabilities concerning the past going beyond its knowledge of the current state. This follows from the following fact. In an MDP, the probability of transitioning into some next state, s_{t+1}, and receiving the corresponding reward, r_t, when transitioning into s_{t+1}, must depend on nothing but the current state s_t and the action a_t taken by the agent in s_t [18, 3]. Environmental states satisfying such a property are said to satisfy the *Markov property* [18, 3].

In order to predict actions, i.e. to map from states to actions, an RL agent employs a so-called *policy*, which implements the agent's decision making strategy and thus determines the agent's behavior. The policy is iteratively updated during training of the agent. The exact methodology, how an agent maps from states to actions, depends on the concrete RL algorithm being employed.

While some policies are instantiated as value functions (consider for example [19]) mapping only indirectly from states to actions [20], other policies learn to directly predict actions a_t for every encountered state s_t [20]. Policy gradient methods (PGMs) fall into the latter category [20, 4, 16]. In the subclass of PGMs exclusively considered in this report, actions are stochastically sampled from some distribution over the action space $\mathcal{A}$, where the sampling is denoted as $a_t \sim \pi_\theta(a_t|s_t)$, which means that action a_t is sampled from $\mathcal{A}$ with the conditional probability $\pi_\theta(a_t|s_t)$. Here, π refers to the agent's policy, which is parameterized by the adjustable, i.e. trainable, parameters θ. The corresponding distribution, with respect to which an action gets sampled, is parameterized as a deterministic function of an observed state s_t and the momentary state of the policy's trainable parameters θ. Since PPO is based on the principles of PGMs, PGMs will be further elaborated on in Section 2.2. For the sake of completeness, note that other subclasses of PGMs exist where actions are determined fully deterministically [20]. However, these methods will not be addressed in this report, since they are not relevant to the main content of this report.

While the value function based RL algorithm described in [19] and policy gradient methods (PGMs), like those described in [6, 16], are *model-free* in that they do not learn to estimate any explicit world model, there are *model-based* RL algorithms where training the policy involves estimating an explicit model of the agent's environment [3, 5].

Regardless of an agent's concrete way of mapping from states to actions using its policy, the goal of every RL agent is to learn a policy which maximizes the expected cumulative discounted future reward, or *expected return*, $\mathbb{E}[R_t]$, which the agent expects to obtain following its policy π [4, 3]. More concretely, the expected return expresses an agent's expectation on how much rewards the agent will accumulate over the course of its current trajectory when currently being in some state s_t and following its policy π, where immediate rewards are weighted more strongly than rewards expected to be received more distant in the future. Formally, as stated in [2], the return R_t, of which the agent seeks to maximize the expectation, can be defined as:

$$R_t = \sum_{k=t}^{T} \gamma^{k-t} r_k, \tag{1}$$

where again T specifies maximal length of a trajectory, t the current time step for which the future reward is evaluated, and γ the discount factor mentioned above.

Being tightly connected to the concept of an expected return, further important concepts in RL are the so-called *state action values*, $Q(s_t, a_t)$, and *state values*, $V(s_t)$, as learned by corresponding *state action value functions* and *state value functions*, respectively. The state action value, or Q-value, specifies the expected expected return R_t of taking action a_t in state s_t under the current policy and is defined as [4]:

$$Q(s_t, a_t) = \mathbb{E}[R_t|s_t, a_t], \tag{2}$$

where $\mathbb{E}$ is the mathematical expectation operator. The state value specifies the expected return R_t of being in some state s_t, following the current policy, and is defined as [4]:

$$V(s_t) = \mathbb{E}[R_t|s_t]. \tag{3}$$

Those value functions are employed, among others, in value-based RL approaches [19], as well as in so-called *Actor-Critic* RL approaches, of which PPO is an instance [6].

In the Actor-Critic approach, two functions get trained concurrently. The first function, being the so-called *critic*, learns to approximate some value function [21]. This approximation is often an estimate of the state value function introduced in Equation 3 [4]. The second function, being the so-called *actor*, is a policy, which directly maps from states to actions as explained for PGMs above [21]. In the actor-critic approach, the actor uses the value estimates produced by the critic when updating its trainable parameters [21], as will be explained in more detail in Section 2.2. Training the actor utilizing the value estimates produced by the critic is supposed to reduce the variance contained in the numeric estimates produced by some objective function, based on which the actor gets trained [4, 21]. This reduction of variance, to be achieved when training both an actor and a critic jointly, may lead to faster convergence of an RL algorithm compared to other PGM training procedures not employing a critic when training the actor, i.e. the actual policy [21]. Section 2.2 will provide further information on this.

An important step in the development of the field of RL was the transition from traditional Reinforcement Learning to Deep Reinforcement Learning (DRL). In DRL, functions to be learned by an agent get approximated using artificial Neural Networks (NNs) or Recurrent NNs (RNNs), i.e. function approximators or dynamical system approximators, respectively. When NNs or RNNs are trained to approximate some policy or value estimates, *Stochastic Gradient Descent* (SGD) [22] is used to learn direct mappings from raw state representations to either actions or value estimates, depending on the type of policy or function being approximated [2, 23]. When training DRL agents based on RNNs, naturally also the RNN's hidden state is taken into consideration when predicting the next action or value estimate. More on DRL will be explained in Section 2.2. Also, since DRL has become the predominant approach to RL, from here on this report will treat the two terms RL and DRL synonymously, with both terms jointly referring to DRL. Note that all the formalisms mentioned above apply to both RL and DRL. Only DRL methods will be considered in the remainder of this report.

Another important dimension along which RL approaches get distinguished is whether they are on-policy or off-policy methods [24, 25]. In the former case, a RL agent's policy or value function gets updated using data exclusively generated by the current state of an agent's policy, while off-policy methods may even use training data having been generated using earlier versions of the current policy [24]. Note that Proximal Policy Optimization (PPO) is an on-policy method that tends to stretch the notion of traditional on-policy methods, since it uses the same training data generated by the current state of an agent's policy for performing multiple epochs of weight updates on the freshly sampled training data.

Lastly, it is worth mentioning that one formally distinguishes between RL approaches, where an agent has access either to full state representations or, alternatively, to only certain observations thereof [17]. In the latter case, an RL agent would not face a MDP, but a Partially Observable Markov Decision Process (POMDP) [17]. In order to avoid further complications of the RL framework outlined so far, this report will treat states and their observations as synonymous throughout, thereby only considering the case where an agent's interaction with its environment can be formalized as a MDP.

2.2 Policy Gradient Methods

Policy Gradient Methods (PGMs) are a class of RL algorithms, where actions are either directly and deterministically computed by some function or, alternatively, sampled stochastically with respect to some probability distribution being defined over a given action space and parameterized via some deterministic function. Those deterministic functions are functions of an agent's currently experienced state [20] and their trainable parameters are trained using stochastic gradient ascent (SGA) [26, 20] so as to maximize the expected return $\mathbb{E}[R_t]$ [20, 4]. This report will exclusively focus on PGMs using stochastic policies, where actions are sampled stochastically, as this technique is used in PPO as well.

A popular subclass of PGMs implementing stochastic policies is the REINFORCE [16] family of RL algorithms. REINFORCE algorithms sample actions $a_t \in \mathcal{A}$ stochastically from the action space $\mathcal{A}$ in accordance with a probability distribution computed over the action space $\mathcal{A}$ [16], where each available action receives a certain probability of being selected. Such a distribution is parameterized through a deterministic function of the state s_t the agent currently faces [16]. Specifically, an action a_t is sampled with probability $\pi_\theta(a_t|s_t)$ [16, 20], where s_t denotes the state the agent experiences at the discrete time step t in the current trajectory and π_θ denotes an agent's policy, being characterized by both a set of trainable parameters θ and a type of probability distribution used for sampling a_t. Adopting Mnih et al.'s [4] view on policies, a policy is a mapping from states to actions. In accordance with this view, a policy may be thought of as a processing pipeline required to transform given state representations s_t into corresponding actions a_t. Thus, in REINFORCE algorithms a policy may be said to consist of three parts. The first part is a deterministic function, assumed to be a NN or RNN in the present report, needed to transform a state representation s_t into some parameterization. The second part is a generic probability distribution of a certain type to be parameterized by the output of the former function. The third part is a succeeding random sampler used to stochastically sample action a_t in accordance with the aforementioned distribution from the previous part.

In the following, the fundamentals of training PGMs using the aforementioned type of policies will be explained. Since PPO is an instance of a REINFORCE algorithm, as will be explained in the next section, the following explanations will only be concerned with the explanation of the fundamentals of how PGMs belonging to the subclass of REINFORCE algorithms are trained.

When training an RL agent employing a NN or RNN, the need arises for an objective function with respect to which the NN's or RNN's trainable parameters θ can be updated.

In REINFORCE, as well as in PGMs in general [20], the objective one intends to optimize the policy's trainable parameters θ for is the expected return, $\mathbb{E}[R_t]$ [4]. This quantity is to be maximized by performing gradient ascent on it [4].

Since the true expectation of the return R_t is not available in practice in most cases, the expectation of this value has to be approximated based on a finite number of samples when training an RL agent in practice. In RL, the approximation $\hat{\mathbb{E}}[\cdot]$ of an expected value $\mathbb{E}[\cdot]$ is achieved by averaging the the expression $\cdot$ contained in the expectation operator $\mathbb{E}$ over a so-called minibatch of previously collected training data [6].

The setup described above, combining gradient ascent with the use of minibatches, inevitably leads to the use of SGA as an optimization procedure when training a REINFORCE algorithm whose trainable parameters constitute a NN or RNN. Thereby, one hopes to move the trainable parameters to a location in the parameter space, which is approximately at least a local optimum in maximizing the expected return.

Note, however, that the estimator of the expected return, $\hat{\mathbb{E}}[R_t]$, is treated as a constant when being differentiated with respect to the policy's trainable parameters θ. Therefore, directly computing the gradient estimator $\nabla_\theta \hat{\mathbb{E}}[R_t]$ would yield the trivial zero vector. In order update the trainable parameters into a non-zero direction, the trainable parameters of a REINFORCE algorithm may be updated into the direction $\nabla_\theta \; log \; \pi_\theta(a_t|s_t) \; R_t$, which is an unbiased estimate of the expected return [4, 16]. Thus, one may use the following gradient estimator when training REINFORCE algorithms [4]:

$$g^{PG} = \hat{\mathbb{E}}[\nabla_\theta \; log \; \pi_\theta(a_t|s_t) \; R_t], \tag{4}$$

where $\hat{\mathbb{E}}[\cdot]$ refers to an empirically estimated expectation (again being estimated by averaging over multiple minibatch examples) and $log \; \pi_\theta(a_t|s_t)$ refers to the log probability of selecting action a_t after having observed state s_t under the current policy π_θ. In order to reduce the variance of the policy gradient estimator g^{PG}, one can opt for subtracting a *baseline* estimate b_t from the expected return R_t in Equation 4, leading to the following variance reduced estimator of $\nabla_\theta \mathbb{E}[R_t]$, which is defined as [4]:

$$g^{VR} = \hat{\mathbb{E}}[\nabla_\theta \; log \; \pi_\theta(a_t|s_t) \; (R_t - b_t)]. \tag{5}$$

Note that also $\nabla_\theta \; log \; \pi_\theta(a_t|s_t) \; (R_t - b_t)$ is an unbiased estimate of $\nabla_\theta \mathbb{E}[R_t]$ [4, 16]. In practice, one may choose the the state value function $V(s_t)$ (see Equation 3) as a baseline b_t, i.e. $b_t \approx V(s_t)$ [4]. In such a case, nowadays one would choose to approximate $V(s_t)$ by a NN or RNN. Furthermore, R_t can be seen as an estimate of the state-action value function $Q(s_t, a_t)$ (see Equation 2) associated with taking action a_t in state s_t [4], from which R_t results. Thus, $R_t \approx Q(s_t, a_t)$, which can in practice be estimated from the environmental responses observed after having executed the actions a_t chosen by the policy over the course of some trajectory in the agent's environment. Since $R_t - b_t \approx Q(s_t, a_t) - V(s_t)$, the term $R_t - b_t$ from Equation 5 is nowadays often replaced in the literature by the so-called advantage estimate, or *advantage* in short, being defined as [4]:

$$A_t = Q(s_t, a_t) - V(s_t). \tag{6}$$

Intuitively, A_t expresses how much better or worse it was to perform action a_t in state s_t, of which the quality is measured by $R_t \approx Q(s_t, a_t)$, compared to the value $V(s_t)$ one expected to receive for being in state s_t while following the current policy π_θ. Using the definition of A_t provided in Equation 6, this leads to the following policy gradient estimator nowadays often being used in practice [6]:

$$g^{A} = \hat{\mathbb{E}}[\nabla_\theta \; log \; \pi_\theta(a_t|s_t) \; A_t]. \tag{7}$$

Given this choice of policy gradient estimator, g^A, one can consider REINFORCE agents as actor-critic architectures (as introduced in Section 2.1), where the actor is the policy π_θ and the critic is the state value network V comparing its state value predictions to the state-action values observed after having executed the actions predicted by the actor [4].

The trainable parameters of an REINFORCE agent's policy are then trained by performing SGA on a policy gradient estimator such as g^A [4, 16].

Too large updates of a policy's trainable parameters θ carry the risk of moving the parameter vector away from a location associated with a local maximum in the objective function's performance landscape that the parameter vector would ideally converge to when being repeatedly updated by small steps in the directions of estimated policy gradients. Therefore, too large parameter updates must be avoided when updating a policy's trainable parameters θ.

In the context of PPO training, one speaks of *destructively large policy updates* when referring to updates of the policy's trainable parameters θ that are large enough to move the parameter vector θ away from some local optimum [6]. In PGMs, those destructively large parameter updates may arise from performing multiple *epochs* of parameter updates on the same set of freshly collected training data [6]. These are the kinds of parameter updates that PPO tries to avoid by using a special objective function [6], as will be explained throughout Section 3.

Speaking about epochs, note that in the present report an *epoch* of parameter updates refers to a sequence of parameter updates based on SGA (or SGD), where each training example from the available set of training data has been part of exactly one minibatch based on which one update of the trainable parameters has been performed. Thus, an epoch refers to a sequence of weight updates resulting from a single pass through the entire training data set.

3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a Deep Reinforcement Learning (DRL) algorithm from the class of policy gradient methods (PGMs) as introduced in Section 2.2 [6]. Its training procedure, as well as its input- and output-behavior, largely follow that of standard REINFORCE algorithms. Thus, in order to map from states to actions, a PPO agent uses a stochastic policy as introduced and explained in Section 2.2. More on that will be explained in Section 3.2. In general, while PPO satisfies almost all conditions necessary to be called a REINFORCE algorithm, there is one aspect where PPO does not exactly follow the general definition of a REINFORCE algorithm or a PGM. This difference is due to the fact that PPO does not always, due to a special objective function, update its trainable parameters exactly into the same direction as REINFORCE algorithms or PGMs do. However, since the authors of PPO still call PPO a PGM, it seems justified to likewise call PPO a member of the REINFORCE family of algorithms. Moreover, PPO is an on-policy algorithm (see Section 2.1), since it exclusively uses the most recently collected training data when performing one or multiple epochs of weight updates on its trainable parameters [6]. Also, PPO is a model-free DRL algorithm.

As hinted upon above, the main feature distinguishing PPO from vanilla PGMs, including REINFORCE algorithms, is PPO's particular objective function used for optimizing the algorithm's trainable parameters, i.e. weights and biases. Recall from Section 2.1 that on-policy PGMs, of which PPO is an instance, may be criticized for being too sample-inefficient [6, 24, 20], meaning that these methods commonly use possibly expensive to obtain training data only once for performing updates of their trainable parameters before discarding the data in favor of newer data. PPO aims for improving upon PGMs' sample efficiency by employing an objective function, which is particularly designed to allow for multiple epochs of updates of its trainable parameters based on the same training data, as will be further elaborated on below.

Note that a trust region-based PGM attaining comparable data efficiency and reliable performance, called Trust Region Policy Optimization (TRPO) [27], had been introduced in the past already [6]. However, TRPO suffered from several problems, which PPO tries to provide a solution for. Firstly, PPO is designed to use a computationally cheaper update procedure for its trainable parameters compared to that used in TRPO [6]. Secondly, PPO has been designed to be compatible with techniques like dropout or parameter sharing (during training), while TRPO is not compatible with these techniques [6]. Thirdly, PPO has been designed to be conceptually less complicated than TRPO [6].

Whilst being rather sub-optimally documented in the literature, even years after its invention PPO is still a state-of-the-art DRL algorithm [15]. Therefore, in the following, the PPO algorithm will be explained in thorough detail for the first time to the best of the author's knowledge.

Section 3.1 will give an overview of the PPO algorithm, while Section 3.2 will describe PPO's input-output behavior, i.e. the way how actions are generated in response to observed states. How various target value estimates are computed throughout training will be described in Section 3.3. PPO's main demarcating feature, its objective function, will be explained in minute detail in Section 3.4, followed by an explanation in Section 3.5 of exploration strategies employed by PPO. Section 3.6 will deal with the question how to back-propagate PPO's overall objective function. Finally, PPO's pseudocode is provided in Section 3.7.

3.1 Overview

PPO is a DRL algorithm, which is capable of learning to map state representations, or *states*, onto one or multiple actions to be performed in in every observed state. Consider a PPO agent whose task it is to map an observed state $s_t \in \mathcal{S}$ onto a single action $a_t \in \mathcal{A}$ to be performed in state s_t. If multiple, independent actions were to be performed in parallel by an agent in a given state s_t at time step t, those could be indexed, using superscript (i), as $a_t^{(i)}$. Here, variable $t \in 1, 2, ..., T$ again refers to a discrete time step in a given trajectory of length T (see Section 2.1). $\mathcal{S}$ and $\mathcal{A}$ refer to the state and action spaces (see Section 2.1), respectively. Action spaces may either be continuous or discrete. Upon executing action a_t in state s_t, the agent transitions into the next state state s_{t+1} and receives a reward r_t depending on the agent's choice of action in state s_t [17]. Recall that actions are selected in given states by means of a policy π_θ, where θ denotes policy's trainable parameters. In PPO, θ refers to the trainable parameters, i.e. weights and biases, of an artificial Neural Network (NN) or a Recurrent NN (RNN) [6]. Furthermore, since PPO is a policy gradient method using a stochastic policy, the NN or RNN inside PPO's policy is used to generate the parameterization for some probability distribution with respect to which an action is sampled. Action a_t is selected with probability $\pi_\theta(a_t|s_t)$ in state s_t given the policy's current set of trainable parameters θ [6, 27, 20]. During training, the set of trainable parameters, θ, is repeatedly updated in incremental steps, using stochastic gradient ascent (SGA), in a way such that an approximation of the expected return $\mathbb{E}[R_t]$ (see Equation 1) gets maximized [20].

In more detail, generating an action a_t from a given state s_t using policy π_θ progresses in three consecutive steps in PPO, which are executed in two separate portions of policy π_θ. The first portion of π_θ is the deterministic portion of the policy, which may be denoted as π_{θ_d}, whereas the second portion of π_θ is stochastic and can be denoted as π_{θ_s}. In PPO, an action a_t is stochastically sampled from an agent's continuous or discrete action space $\mathcal{A}$ in accordance with a likewise continuous or discrete probability distribution defined over the PPO agent's action space $\mathcal{A}$, as will be further described in Section 3.2. Such a distribution is generated and sampled from in the stochastic portion of the policy π_θ, i.e. in π_{θ_s}. The corresponding parameterization, here denoted as ϕ_t, for defining said probability distribution is calculated in the deterministic portion of π_θ, i.e. in π_{θ_d}. In practice, π_{θ_d} is a NN or RNN being parameterized by θ. Therefore, the deterministic portion of the policy, π_{θ_d}, may also be referred to as *policy network*. For calculating the aforementioned parameterization ϕ_t, the policy network π_{θ_d} consumes state s_t and computes the aforementioned parameterization ϕ_t in its output layer. If a PPO agent has to perform multiple actions $a_t^{(i)}$ in parallel during each time step t, each action $a_t^{(i)}$ for $i \in \{1, 2, ..., I\}$ is sampled from a respective action space $\mathcal{A}^{(i)}$ with respect to a respective probability distribution $\delta_t^{(i)}$ parameterized by a respective parameterization $\phi_t^{(i)}$. Each parameterization $\phi_t^{(i)}$ is computed by a respective set of output nodes in the policy network's output layer and any possible covariance between actions to be predicted in parallel during a single time step is assumed to be zero, as will be explained in more detail for continuous action spaces in Section 3.2.1. Hence, each action $a_t^{(i)}$ is assumed to be statistically independent of the other actions sampled during the same time step t.

To the level of detail described above, this procedure of sampling actions perfectly follows that described for the REINFORCE family of DRL algorithms [16], as described in Section 2.2. When explaining PPO's procedure for generating actions in more detail in Section 3.2, however, some aspect will be pointed out distinguishing PPO's procedure for sampling actions from that used by the REINFORCE family of algorithms.

While PPO's main objective is to train an agent's (deterministic) policy network π_{θ_d}, training an agent actually involves training two networks concurrently [6], as described above already in the context of how REINFORCE algorithms are trained (see Section 2.2). The first network is the policy network π_{θ_d} itself, while the second network is a state value network V_ω used to reduce the variance contained in the numeric estimates based on which the policy network is trained [4]. Here, ω denotes the trainable parameters of an employed state value network V. Parameter sharing between π_θ and V_ω may apply [6, 4].

Training a PPO agent means repeatedly alternating between the two steps of collecting new training data and then updating both the policy network and state value network based on the freshly sampled training data for multiple epochs [6]. The former of the two aforementioned steps may be referred to as *data collection step* or *training data generation step*, while the latter may be referred to as *update step.*

For the data collection step, N PPO agents in parallel are placed in separate, independent instances of the same type of environment [6]. Then, while all agents are run in parallel, each of the N agents is made to interact with its respective environment for T time steps [6]. It is noteworthy that all N agents use the most up-to-date state of the policy network, which is held fixed during a data collection step [6]. If either of the N parallel agents encounters a terminal state before T time steps have elapsed, the environment instances of all N agents get reinitialized to new stochastically chosen initial states. Then, the agents continue interacting with their environments. This process is repeated until each agent has experienced T time steps during a data collection step [6]. Parallelizing agents' interactions with their environments, as well as the reinitializations of their environments, is done for the sake of efficiency. In this way, not only the interactions of the N agents with their environments can be efficiently parallelized, but also the computations of the target values V_t^{target} and A_t, which will be introduced below, can be largely parallelized using tensor operations. For each experienced state transition, the corresponding state s_t, next state s_{t+1}, action a_t, the corresponding probability of selecting action a_t in s_t given π_θ, denoted as $\pi_{\theta_{old}}(a_t|s_t)$, and the corresponding reward r_t get stored in a so-called *tuple* of the form $o_t = (s_t, a_t, \pi_{\theta_{old}}(a_t|s_t), s_{t+1}, r_t)$ in preparation for the next training step, i.e. the update step of the trainable parameter. Note that it would be more accurate to index each observation tuple o_t by the two additional subscripts n and j, resulting in notation $o_{t,n,j}$. Here, n would indicate the index of the n^{th} parallel agent that has generated a given observation tuple o_t. Subscript j would indicate how many times an environment's discrete time index t has been reset already due to an environment's reinitialization while generating the n^{th} agent's T state transition observations. However, for simplicity of notation, we will generally omit explicitly stating the two subscripts n and j (unless stated otherwise) and only implicitly assume the two subscripts n and j to be known for every tuple o_t.

For each observed state transition, encoded as a tuple of stored data $o_t = (s_t, a_t, \pi_{\theta_{old}}(a_t|s_t), s_{t+1}, r_t)$, two additional target values get computed and appended to the tuple. The first target value to be added per tuple is target state value V_t^{target} associated with state s_t taken from a given observation tuple o_t. The second target value to be added per tuple is the advantage A_t associated with having performed action a_t in state s_t [6, 4]. How V_t^{target} and A_t are concretely computed, using the state value network V_ω, will be described in Sections 3.3.1 and 3.3.2, respectively.

After training data has been collected, in the next step, i.e. the successive update step, the trainable parameters of both the policy network π_{θ_d} and the state value network V_ω get updated using multiple epochs of minibatch training on the freshly collected training data.

The clipped objective function used by PPO to train the policy network has specifically been designed with the intention to avoid destructively large weight updates of the policy network while performing multiple epochs of weight updates using the same freshly collected training data. As indicated earlier, this is meant to increase PPO's data efficiency compared to that of other PGMs. The training data used here are exclusively the observation tuples $o_t = (s_t, a_t, \pi_{\theta_{old}}(a_t|s_t), s_{t+1}, r_t, V_t^{target}, A_t)$ collected during the immediately preceding data collection step. The corresponding objective function, L^{CLIP}, is defined as follows:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[min\left(p_t(\theta)A_t, clip(p_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right], \tag{8}$$

where $\hat{\mathbb{E}}_t[\cdot]$ denotes the empirical expectation operator, which computes the value of its contained function $\cdot$ as the average over a finite set of training examples contained in a minibatch of training examples [6]. Note that each training example is one of the previously collected observation tuples o_t (or, more precisely, $o_{t,n,j}$ using the extended notation as explained above). Unless indicated otherwise, in the context of performing updates of the trainable parameters, i.e. during an update step, subscript t usually denotes the index of a randomly sampled training example (taken from a minibatch of training examples), while it indicates, during a data collection step, the discrete time step inside a given environment during which the information contained in an observation tuple has been observed. Minibatches of training data are sampled at random, without replacement, from the pool of available observation tuples o_t. The term $p_t(\theta)$ in Equation 8 refers to a probability ratio being defined as:

$$p_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \tag{9}$$

where $\pi_{\theta_{old}}(a_t|s_t)$ denotes the probability of action a_t in state s_t during the data collection step (recorded in the observation tuples), while $\pi_\theta(a_t|s_t)$ refers to the probability of a_t in s_t given the most up-to-date state of π_θ [6, 27]. The expression *min* in Equation 8 refers to the mathematical operator that returns the minimum of its two input values and the *clip* operator clips its input value $p_t(\theta)$ to the value range $[1-\epsilon, 1+\epsilon]$, where ϵ is a hyperparameter that has to be chosen by the experimenter. For example, $\epsilon = 0.2$ according to [6].

The meaning of the individual terms contained in the objective function L^{CLIP}, as well as the explanation why this objective function is meant to allow for performing multiple epochs of weight updates based on the same data without experiencing destructively large weight updates, will be explained in Section 3.4.

In theory, the policy network's objective function L^{CLIP} (Equation 8) is optimized, i.e. maximized, using stochastic gradient ascent. In practice, however, commonly SGD is used to minimize $-L^{CLIP}$, thereby treating the objective function to be maximized [6] as a loss function to be minimized. This is because standard deep learning libraries, used to train DRL agents in practice, nowadays only support SGD, but not SGA.

In order to encourage exploratory behavior of the policy network π_{θ_a} during training, an Entropy bonus H can be added to the policy network's objective function L^{CLIP}, as will be further explained in Section 3.5. In theory, this works for both continuous and discrete action spaces $\mathcal{A}$. However, as will be pointed out later, this procedure is ineffective in encouraging exploration in continuous action spaces when strictly following the PPO training procedure. Introducing a weighting factor h for the Entropy bonus, to control the contribution of the Entropy bonus to the overall objective, and adding the weighted Entropy bonus to the clipped objective function results in the objective function $L^{EXPLORE} = L^{CLIP} + hH$, which is then to be maximized using stochastic gradient ascent. Again, when using stochastic gradient descent (as done in practice), $-L^{EXPLORE}$ is to be minimized.

Training the state value network V_ω is done via minimizing the squared error, averaged over multiple training examples contained in a minibatch, between a predicted state value $V_\omega(s_t)$ and the corresponding target state value V_t^{target} (see Section 3.3.1 below) [6]. The corresponding loss function for training the state value network V_ω is therefore defined as:

$$L^V = \hat{\mathbb{E}}_t[(V_\omega(s_t) - V_t^{target})^2], \tag{10}$$

where t refers to the index of some training example. The calculation of V_t^{target} will be described in Section 3.3.1.

Finally, also taking the training of V_ω into consideration, this results in the overall objective function

$$L^{CLIP+H+V} = L^{CLIP} + hH - vL^V \tag{11}$$

to be optimized, i.e. maximized, using stochastic gradient ascent [6]. Just like scalar h, also v is a weighting factor in Equation 11 to be chosen by the experimenter. When using stochastic gradient descent, again $-L^{CLIP+H+V}$ has to be minimized. How this objective function is evaluated and back-propagated will be explained in Section 3.6. Commonly, the Adam optimizer is used to perform SGD to minimize $-L^{CLIP+H+V}$ [6].

3.2 Generation of Actions

PPO is very flexible when it comes to its input-output behavior, i.e. the nature of the inputs and outputs the algorithm can learn to map between. Upon receiving a single state representation s_t as input, the policy generates as output one or multiple continuous or discrete actions to be executed in state s_t, depending on the requirements imposed by the environment. Here, the only restriction that applies is that both the inputs and outputs must be scalar or multi-dimensional real numbers. Unless indicated otherwise, for the sake if simplicity, we will generally assume that only a single a action a_t is generated per time step t, and thus per state representation s_t, by an agent's policy.

As stated in Section 3.1, PPO maps states to actions using a stochastic policy π_θ, i.e. each action a_t is (pseudo-)randomly sampled from the action space $\mathscr{A}$ with respect to some probability distribution δ_t computed over $\mathscr{A}$. Such a probability distribution δ_t is parameterized by a set of parameters denoted ϕ_t. Such a parameterization ϕ_t can be computed either partially or entirely as a function of an input state s_t. In the literature (see [16] or Sections 2 and 3 of the supplementary material associated with [4]) it has been suggested to compute all components of ϕ_t as a function of an input state s_t using NNs or RNNs. This is what is commonly done in REINFORCE algorithms. As an exception to this, in the case of generating continuous actions, the inventors of PPO [6] made use of a technique originally proposed in [27], where parameterization ϕ_t is only partially defined as a function of state representation s_t, while some component of ϕ_t is independent of s_t. This will be further explained in Section 3.2.1.

More concretely, in the most generic way, the generation of actions in PPO can be described as follows. First, a PPO agent receives a state representation s_t. This state representation is passed through the policy network, i.e. the NN or RNN constituting the deterministic portion of the agent's policy π_θ. This results in (at least a part of) a set of parameters ϕ_t being computed in the policy network's output layer. The parameterization ϕ_t may have to be post-processed in order to normalize probability mass estimates or to enforce a non-negative standard deviation. The potentially post-processed set of parameters is then used to parameterize a probability distribution δ_t, which is defined over the agent's action space $\mathscr{A}$. Finally, action a_t is sampled from action space $\mathscr{A}$ in accordance with δ_t by applying a random sampler to δ_t.

This procedure can easily be generalized to the case where multiple, say I, actions $a_t^{(i)}$ (where index $i \in \{1, 2, ..., I\}$) have to be generated and executed concurrently each time step t, i.e. in each state s_t. In this case, the policy network features I sets of output nodes, where each set of output nodes produces (at least a subset of) one set of parameters $\phi_t^{(i)}$ used to parameterize a single probability distribution $\delta_t^{(i)}$. Here, probability distribution $\delta_t^{(i)}$ is defined over action space $\mathscr{A}^{(i)}$. Note that each individual parameterization $\phi_t^{(i)}$ may have to be post-processed as described above. Finally, for each probability distribution $\delta_t^{(i)}$, a single action $a_t^{(i)} \in \mathscr{A}^{(i)}$ is sampled.

In order for a PPO agent to be able to deal with state representations of different nature, the network architecture of the policy network may be varied. If state representations are of visual nature, the policy network's input layers may be convolutional NN layers. Otherwise, they may be fully-connected NN layers.

Sections 3.2.1 and 3.2.2 describe the particular procedures for generating continuous and discrete actions, respectively.

3.2.1 Continuous Action Spaces

In [6], the inventors of PPO propose to use a technique for generating continuous actions, which has previously been proposed in [27]. In the following, this technique will be described.

In PPO, scalar continuous actions are stochastically sampled from one-dimensional Gaussian distributions. Thus, $a_t \sim \mathcal{N}(\mu_t, \sigma_t)$. Such a Gaussian distribution $\mathcal{N}(\mu_t, \sigma_t)$ is denoted δ_t and parameterized by a set of parameters $\phi_t = \{\mu_t, \sigma_t\}$, where μ_t and σ_t are the mean and the standard deviation of the Gaussian distribution δ_t, respectively. The mean μ_t is computed as a function of state representation s_t experienced at time step t in a given trajectory. As proposed in [27], the standard deviation σ_t, controlling the exploratory behavior of an agent during training, is determined independently of state representation s_t.

In more detail, generating a continuous action works as follows. When a PPO agent receives a state representation s_t, s_t is fed through the policy network, i.e. the deterministic portion of the policy. The output node of the policy network computes the mean μ_t, which is used to parameterize a Gaussian distribution δ_t defined over the continuous action space $\mathcal{A}$. The standard deviation σ_t used to parameterize δ_t is a hyperparameter to be chosen by the experimenter before the onset of training and is kept fixed throughout the entire training procedure. Then, a random sampler is applied to the parameterized probability distribution δ_t and action a_t gets sampled by the random sampler with respect to δ_t. More on the choice of the standard deviation hyperparameter σ_t will be explained in Section 4.2.

How to generalize this procedure to multiple action spaces has been explained in the introduction of Section 3.2. Particularly, in this case, I output nodes of the policy network generate I means $\mu_t^{(i)}$ used to parameterize I statistically independent Gaussian distributions $\delta_t^{(i)}$. The I standard deviations used to parameterize the I probability distributions $\delta_t^{(i)}$ are all taken to be the same fixed constant. The remaining procedure is described above. Note, that when predicting multiple continuous actions, one may alternatively and analogously sample an I-dimensional point from a I-dimensional Gaussian distribution featuring an I-dimensional mean vector, consisting of the I outputs generated by the policy network, and a diagonal covariance matrix containing the fixed standard deviation parameter along its diagonal. In this case, the value along the i^{th} dimension of a point sampled from the I-dimensional Gaussian distribution constitutes action $a_t^{(i)}$.

3.2.2 Discrete Action Spaces

In [6], the inventors of PPO propose to use a technique for generating discrete actions, which has previously been proposed in [4]. In the following, this technique will be described.

In PPO, discrete actions are drawn from a corresponding discrete action space defined as $\mathcal{A} = \{1, 2, ..., M\}$. Here, M denotes the number of available options to draw an action a_t from. Each natural number contained in action space $\mathcal{A}$, i.e. 1 through M, is representative one action available in the agent's environment. An action a_t is sampled from action space $\mathcal{A}$ with respect to a Multinomial probability distribution δ_t being defined over $\mathcal{A}$. The probability distribution δ_t assigns each available element in $\mathcal{A}$ a probability of being sampled as action a_t. The probability distribution's parameterization is denoted ϕ_t and computed by the policy network as a function of a received state representation s_t.

More concretely, this works as follows. Upon receiving a state representation s_t, s_t is fed through the policy network. The policy network's output layer contains a set of nodes yielding the unnormalized probability mass estimates for choosing either of the possible actions in $\mathcal{A}$ as a_t. Precisely, the m^{th} output node yields the unnormalized probability of sampling the m^{th} element from action space $\mathcal{A}$ as action a_t. Applying the Softmax [28] function to the outputs generated by the policy network yields a vector ϕ_t of normalized probability mass estimates, where the m^{th} element of vector ϕ_t, $\phi_{t,m}$, specifies the normalized probability of sampling the m^{th} element from $\mathcal{A}$ as a_t given a received state representation s_t. The vector ϕ_t is used to parameterize a Multinomial probability distribution δ_t. Finally, action a_t is obtained by applying a random sampler to probability distribution δ_t, where action a_t is sampled with probability $\phi_{t,a_t} = \pi_\theta(a_t|s_t)$.

How this procedure can be generalized to produce multiple actions $a_t^{(i)}$ concurrently in every state s_t has been described in the introduction of Section 3.2.

3.3 Computation of Target Values

As explained in Section 3.1, training a PPO agent involves computing *target state values* V_t^{target} and *advantage estimates* A_t. The technique used to compute those quantities uses so-called n-step returns [4], as previously presented in [18, 29] according to [4], and is particularly suitable for training RNN-based network architectures [6]. Note that target state values V_t^{target} and advantage estimates A_t are always calculated during the training data generation steps of a PPO agent's training procedure [6]. That means that target values are always computed based on the states of the policy network and the state-value network during the most recent training data generation step [6]. Those states of the policy and the state value network are denoted $\pi_{\theta_{old}}$ and $V_{\omega_{old}}$, respectively, to distinguish them from the corresponding states π_θ and V_ω being repeatedly updated during a successive weight update step.

According to [6], an alternative method for calculating target state values and advantages estimates in the context of training PPO agents, which is not considered here, is presented in [30].

Sections 3.3.1 and 3.3.2 describe for target state values and advantage estimates respectively what these quantities measure and how they are computed.

3.3.1 Target State Values

Following a concept presented in [4], a target state value V_t^{target} denotes the discounted cumulative reward associated with having taken an action a_t in a given state s_t and is computed based on the observations made throughout an experienced trajectory, i.e. a sequence of interactions between an agent and its environment.

In the context of training a PPO agent, target state values V_t^{target} are used in two ways. Firstly, they are used to train the state value network V_ω (see Equation 10). Secondly, they are used to compute advantage estimates A_t for training the policy network (see Section 3.3.2). They are computed as follows.

First, an agent is made to interact with its environment for a given maximal trajectory length, i.e. a maximal number of time steps T (as described in Section 3.1). When a given trajectory ends, target state values are computed for every state s_t experienced during the trajectory according to the following equation:

$$V_t^{target} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{n-1} r_{t+n-1} + \gamma^n V_{\omega_{old}}(s_{t+n}), \tag{12}$$

where V_t^{target} denotes the target state value associated with a given state (or state representation) s_t, r_t is the reward received for having executed action a_t in state s_t, and $\gamma \in (0, 1]$ is the discount factor mentioned earlier. Furthermore, the term $t + n$ denotes the time step at which the trajectory under consideration terminated in the final state s_{t+n}. If the trajectory terminated due to the maximal trajectory length T being reached, $V_{\omega_{old}}(s_{t+n})$ denotes the state value associated with state s_{t+n} as predicted by the state value network. Otherwise, $V_{\omega_{old}}(s_{t+n})$ is set to 0, since this condition indicates that the agent reached a terminal state within its environment from where onward no future rewards could be accumulated any longer. Since target state values are computed during a data collection steps, i.e. before the onset of a training iteration's weight update step, the state of the state value network used to compute V_t^{target} is denoted $V_{\omega_{old}}$ rather than V_ω.

By using the aforementioned way of computing target state values V_t^{target}, each observed reward is used to compute up to T target state values [4]. Thereby, this procedure potentially increases the efficiency of propagating the information contained in each observed reward to the corresponding value estimates being dependent on it [4].

3.3.2 Advantage Estimates

Following [4, 6], an advantage estimate A_t quantifies how much better or worse the observed outcome of choosing a certain action in a given state was compared to the state's estimated value predicted by the state value network. Here, the qualitative outcome of choosing an action a_t in a given state s_t, being compared to a state's predicted value, is measured by the state's target state value V_t^{target} (see Section 3.3.1). Thus, the computation of advantage estimates extends the computation of target state values described above.

The equation for calculating an advantage estimate A_t, associated with having taken action a_t in state s_t as experienced during some trajectory of maximal length T, is given by:

$$A_t = V_t^{target} - V_{\omega_{old}}(s_t), \tag{13}$$

where the target state value V_t^{target} is computed as described in Section 3.3.1 [6, 4] and $V_{\omega_{old}}(s_t)$ refers to the state value, associated with state s_t, predicted by the state value network during the data generation step.

Intuitively, Equation 13 makes sense, since it compares the observed return V_t^{target}, associated with having taken action a_t in state s_t, to the currently estimated value $V_{\omega_{old}}(s_t)$ of state s_t. It is a practical implementation of the more theoretical equation for calculating advantage estimates A_t provided in Equation 6 above.

Note that there is a typo in Equation 10 in [6], which concerns the computation of the advantage estimate A_t in PPO. In that Equation, the term γ^{T-t+1} is meant to be γ^{T-t-1} according to the logic of the equation, since otherwise the reward r_{T-1} gets discounted disproportionately strongly.

3.4 Explanation of Policy Network's Main Objective Function L^{CLIP}

This Subsection will explain in detail the main objective function, L^{CLIP}, which is used to update PPO's policy network using stochstic gradient ascent (SGA). As stated in Equation 8, the objective function L^{CLIP} is defined as $L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min\left(p_t(\theta)A_t, clip(p_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right]$, where $p_t(\theta)$ is a probability ratio being defined as $p_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ [6] and A_t, as introduced in Section 2.2, is an advantage estimate associated with taking an action a_t in state s_t. How to concretely compute A_t has been described in Section 3.3.2 above. Since the topic treated here is concerned with performing updates of an employed NN's or RNN's trainable parameters, subscript t denotes the index of a training example contained in a randomly sampled minibatch again. Again, $\pi_\theta(a_t|s_t)$ refers to the probability of choosing action a_t in state s_t given the most up-to-date state of the policy network, while $\pi_{\theta_{old}}(a_t|s_t)$ refers to the probability of choosing action a_t in state s_t given the state of the policy network during the most recent training data generation step.

This Subsection will be structured as follows. Firstly, a motivation for using L^{CLIP} will be given. Afterwards, the function's constituting components will be explained individually, finally leading up to the description of how the objective function L^{CLIP} overall behaves under different conditions concerning the values of its input arguments. Throughout, the focus of this subsection will not primarily lie on how the objective function behaves in the forward pass, but rather on the more important and more involved topic of how it behaves in the back-propagation pass.

As stated in Section 2.2, nowadays many DRL algorithms from the class of policy gradient methods (PGMs) are trained by performing SGA on the policy gradient estimator $g^A = \hat{\mathbb{E}}[\nabla_\theta \; log \; \pi_\theta(a_t|s_t) \; A_t]$ (see Equation 7). This, however, as indicated earlier, usually does not allow for using training data efficiently [6]. This is because updating the policy repeatedly, i.e. for multiple epochs, based on the same freshly collected training data may lead to destructively large weight updates [6] as the difference between the old state of the policy, used for generating the training data, and the updated state of the policy increases with the number of weight updates performed.

To facilitate more efficient usage of the training data, the main objective function employed by PPO, L^{CLIP}, is particularly designed to allow for multiple epochs of weight updates on the same set of training data. In order to allow for multiple epochs of weight updates on the same data, PPO's main objective function, L^{CLIP}, aims roughly speaking at limiting the extent to which the current state of the policy can be changed compared to the old state used for collecting the training data. More precisely, it aims at limiting to which extent the policy can be changed even further through consecutive weight updates after a rough approximation of the divergence between the updated state of the policy and the old state moves beyond a given threshold value while performing multiple epochs of weight updates on the same training data.

This idea of limiting the impact of weight updates on the state of the policy network had already been explored in Trust Region Policy Optimization (TRPO) [27]. However, one of the downsides associated with using TRPO is its computationally relatively expensive and inflexible procedure used for limiting the extent to which weight updates may change its policy's trainable parameters θ [6]. Therefore, the inventors of PPO aimed at using a more inexpensive to evaluate, more flexible, and conceptually simpler approximation of divergence between the two policy states π_θ and $\pi_{\theta_{old}}$ based on which they could limit the impact of individual weight updates on the policy π_θ. As a consequence, the inventors of PPO opted for directly incorporating a cheap, simple, and flexible to evaluate probability ratio into PPO's main objective function (or *policy gradient estimator*) L^{CLIP}. This probability ratio acts as a measure, or rather rough approximation, of divergence between π_θ and $\pi_{\theta_{old}}$ and is defined as $p_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. According to [6], the use of the probability ratio $p_t(\theta)$ has originally been proposed in [31].

The behavior of the probability ratio $p_t(\theta)$ is as follows:

If an action a_t becomes *more unlikely* under the current policy, π_θ, (in a given environmental state s_t) than it used to be under the old state of the policy, $\pi_{\theta_{old}}$, the probability ratio will vanish towards positive 0, since $\pi_\theta(a_t|s_t)$ shrinks compared to $\pi_{\theta_{old}}(a_t|s_t)$.

When actions become *more likely* under the current state of the policy, π_θ, than they used to be under the old one, $\pi_{\theta_{old}}$, the probability ratio $p_t(\theta)$ will grow and approach positive infinity in the limit, since $\pi_\theta(a_t|s_t)$ grows compared to $\pi_{\theta_{old}}(a_t|s_t)$.

In cases where the probability of choosing some action a_t in a given state s_t is *comparatively similar* under both the current and the old state of the policy, i.e. when the behavior of both states of the policy is similar in terms of the given metric, the probability ratio will evaluate to values close to 1. This is because the two probabilities $\pi_\theta(a_t|s_t)$ and $\pi_{\theta_{old}}(a_t|s_t)$ will have similar values.

Thus, again, the probability ratio $p_t(\theta)$ can be seen as a cheap to evaluate measure, or *rough indicator*, of divergence between the two states π_θ and $\pi_{\theta_{old}}$ of the policy. Note that this divergence measure does not compute any established metric to accurately assess how different the two states of the policy are across all possible actions in all possible states. Instead, it assesses for each training example in a given minibatch how much the behavior of the policy has changed with respect to the training example currently under consideration.

When inspecting L^{CLIP}, one can see that the probability ratio $p_t(\theta)$ is used in two places. The first term the probability ratio is part of is the so-called *unclipped objective* [6], being defined as $p_t(\theta)A_t$. The second term that the probability ratio is part of, $clip(p_t(\theta), 1-\epsilon, 1+\epsilon)A_t$, is referred to as *clipped objective* [6]. Clipping, as done by the clipping operator *clip*, here refers to the act of restricting the range of values, that $p_t(\theta)$ can take, to the interval $[1-\epsilon, 1+\epsilon]$. The minimum operator, *min*, in L^{CLIP} takes the minimum of the unclipped and clipped objective and returns it as the result of evaluating L^{CLIP}.

This design of L^{CLIP} is supposed to have two effects. Firstly, it is supposed to yield a pessimistic, i.e. lower, estimate of the policy's performance [6]. Secondly, it is supposed to avoid destructively large weight updates into the direction increasing the probability of re-selecting some action a_t in a given state s_t [6]. To see how this works, both the clipping operator and the minimum operator have to be explained in more detail.

The first operator to be inspected in more detail is a clipping operator, *clip*, which is part of the aforementioned clipped objective and ensures that its first input argument, the probability ratio $p_t(\theta)$, lies within a specific interval defined by the operator's second and third input arguments, namely $1-\epsilon$ and $1+\epsilon$. Mathematically, the clipping operation is defined as follows:

$$clip(p_t(\theta),\ 1-\epsilon,\ 1+\epsilon) = \begin{cases} 1-\epsilon & \text{if } p_t(\theta) < 1-\epsilon \\ 1+\epsilon & \text{if } p_t(\theta) > 1+\epsilon \\ p_t(\theta) & \quad\text{else} \end{cases} \tag{14}$$

Clipping, i.e. restricting, the range of values that the probability ratio $p_t(\theta)$ can take is supposed to remove the incentive for pushing the probability ratio outside the interval enforced by the clipping operator during repeated updates on the same data [6]. Put differently, due to the clipping operation, the probability ratio $p_t(\theta)$ is supposed to remain within the interval $[1-\epsilon, 1+\epsilon]$ even after multiple epochs of weight updates performed on the same data. Thereby, the goal of avoiding destructively large weight updates is supposed to be achieved.

In order to see why this is supposed to be the case, consider the following. When a clipping operator's input value to be clipped falls outside the interval of admissible input values, i.e. when clipping applies, the partial derivative leading through the clipping operator becomes 0. Particularly, here the partial derivative of the clipping operator's output value with respect to the clipping operator's input value $p_t(\theta)$ becomes 0 if clipping applies [32]. This is due to the fact that the clipping operator's output value is constant when clipping applies. Constants, in turn, evaluate to 0 when performing differentiation on them. Only when clipping does *not* apply, i.e. when $1-\epsilon \leq p_t(\theta) \leq 1+\epsilon$, the partial derivative of the clipping operator's output value with respect to the input value to be clipped is 1 and therefore non-zero (see Equation 20 including the corresponding explanation). Due to the nature of the back-propagation algorithm, only the zero-gradient will result from back-propagation paths leading through operators inside an objective function, whose partial derivatives are 0. Therefore, only the zero-gradient will result from the back-propagation paths leading through the clipping operator when clipping applies during a corresponding forward-pass. Thus, training examples, where the only non-zero gradient component is back-propagated via the clipping operator, will not cause a resulting gradient to point into a direction in parameter space, where the probabilities of some action in a given state will change even more extremely if it has changed enough already for clipping to apply. Thereby, multiple epochs of weight updates may safely be performed on the same training data (at least in theory; a critical assessment of that claim will be provided in Section 5.1).

The second operator of importance in L^{CLIP} to be explained in more detail is the mathematical minimum operator, *min*, which returns the minimum of its two input arguments. Recall that the minimum operator is employed in L^{CLIP} to return the minimum of the unclipped and clipped objective. The partial derivative of the minimum operator's output value with respect to its minimal input value is 1, while the partial derivative of the output value with respect to the other input value is 0 (see Equations 21 and 22). Note that, mathematically speaking, the minimum operator is not differentiable when both its input arguments are equivalent. In practice, however, machine learning packages like PyTorch do select either of two equivalent input values as the minimum input value, thereby causing the minimum operator to be differentiable even when this would mathematically speaking not be the possible.

Given all the background knowledge stated above, below it will be explained case-wise how PPO's main objective function L^{CLIP} behaves when systematically varying its input arguments' values. Readers only being interested in a short summary of the explanations presented below may skip to Table 1 summarizing the contents presented in the remainder of this subsection.

In cases where clipping does not apply, i.e. in cases where the probability ratio lies within the interval $[1-\epsilon, 1+\epsilon]$, neither the clipping operator nor the minimum operator impact the computation of the gradient. Instead, the gradient associated with a training example where clipping does not apply will point into a direction locally maximizing the unclipped objective $p_t(\theta)A_t$. This is irrespective of whether the advantage estimate A_t, introduced in Section 2.2, is positive or negative.

Next, consider the cases where the probability ratio $p_t(\theta)$ is lower than the threshold value $1-\epsilon$, i.e. $p_t(\theta) < 1-\epsilon$. In that case, clipping applies and the behavior of L^{CLIP} depends on whether the advantage estimate A_t is positive or negative.

If the advantage estimate is positive, i.e. $A_t > 0$, the minimum operator will receive as its input arguments a relatively small positive value for the unclipped objective, $p_t(\theta)A_t$, and a larger positive value for the clipped objective, $clip(p_t(\theta), 1-\epsilon, 1+\epsilon)A_t$. In this case, where $A_t > 0$ and $p_t(\theta) < 1-\epsilon$, the minimum operator will be dominated by the smaller unclipped objective $p_t(\theta)A_t$. Intuitively, this means the following. If the probability of selecting some action a_t in a state s_t has decreased during the previous weight updates, as indicated by $p_t(\theta) < 1-\epsilon$, but choosing a_t in s_t was better than expected, as indicated by $A_t > 0$, then the gradient will point into the direction maximizing $p_t(\theta)A_t$. Thus, the training example in question will try to influence the gradient, which is computed over a minibatch of training examples, in such a way that action a_t becomes more likely in state s_t again.

If the advantage estimate is negative, i.e. $A_t < 0$, while $p_t(\theta) < 1-\epsilon$, then the behavior of L^{CLIP} changes. In this case, multiplying a negative advantage estimate A_t by a small positive value for $p_t(\theta)$, being smaller than $1-\epsilon$, will result in a negative value of less magnitude (in the negative direction) than is obtained by multiplying the negative value A_t by the corresponding clipped probability ratio evaluating to at least $1-\epsilon$. In this case, where $A_t < 0$ and $p_t(\theta) < 1-\epsilon$, the minimum operator will return the clipped objective, evaluating to the negative value $(1-\epsilon)A_t$. Thus, clipping applies when $A_t < 0$ while $p_t(\theta) < 1-\epsilon$. Since clipping applies, the gradient associated with a training example satisfying the aforementioned conditions will be the zero-gradient. As a consequence, a corresponding training example will not encourage the gradient, which is computed over an entire minibatch of training examples, to point into a direction making the probability of selecting action a_t in state s_t, being associated with a negative advantage A_t, even more unlikely if it has become unlikely enough already for the probability ratio to drop below $1-\epsilon$. If this was not the case, destructively large weight updates could result due to the increasingly larger divergence between the two states of the policy indicated by the probability ratio considerably diverging from value 1 already.

Now, consider the two cases where $p_t(\theta) > 1 + \epsilon$. Also here, the behavior of L^{CLIP} depends on whether the advantage estimate A_t is positive or negative.

Consider the case where $A_t > 0$ while $p_t(\theta) > 1 + \epsilon$. In this case, the probability of choosing an action a_t associated with a positive advantage estimate A_t in state s_t has become considerably larger already under the current policy than it used to be under the old state of the policy. This is indicated by the condition $p_t(\theta) > 1 + \epsilon$. In this case, clipping applies and the minimum operator will return the clipped objective as its minimal input value. Thus, since the overall objective value is clipped, only the zero-gradient will result from a training example where $A_t > 0$ while $p_t(\theta) > 1 + \epsilon$. Also in this case, destructively large weight updates are supposed to be prevented through the resulting zero-gradient, as explained above already.

Lastly, consider the case where the advantage estimate is negative, i.e. $A_t < 0$, while $p_t(\theta) > 1 + \epsilon$. In such a case, the probability of selecting an action a_t in a state s_t has become considerably larger under the current state of the policy than it used to be under the old state, while choosing action a_t in state s_t led to a worse outcome than expected, as indicated by the negative advantage estimate A_t. Here, the clipped objective will evaluate to the negative value $(1 + \epsilon)A_t$, while the unclipped objective will evaluate to a negative value $p_t(\theta)A_t$ of magnitude larger than $(1 + \epsilon)A_t$. Consequently, the minimum operator will return the unclipped objective, $p_t(\theta)A_t$, being of larger magnitude in the negative direction. Therefore, a training example satisfying the conditions $A_t < 0$ and $p_t(\theta) > 1 + \epsilon$ will be associated with a non-zero gradient pointing into the direction maximizing the negative value $p_t(\theta)A_t$. This is to rigorously correct the behavior of the policy in the case that an action has become more likely in a given state, potentially as a byproduct of updating the policy on other training examples, even though past experience has indicated that the chosen action was worse than expected in the given state in terms of the experienced advantage estimate. Here, no means of preventing destructively large weight updates applies. Also, in this way, the objective function L^{CLIP} aims at yielding a pessimistic estimate of the policy's performance. Drastic contributions to the objective value are only admissible if they make the valuation of the objective worse, but are clipped, i.e. bounded, when they would lead to an improvement of the objective value [6].

$p_t(\theta) \ > 0$	A_t	**Return Value of** *min*	**Objective is Clipped**	**Sign of Objective**	**Gradient**
$p_t(\theta) \in [1-\epsilon, 1+\epsilon]$	$+$	$p_t(\theta)A_t$	no	$+$	✓
$p_t(\theta) \in [1-\epsilon, 1+\epsilon]$	$-$	$p_t(\theta)A_t$	no	$-$	✓
$p_t(\theta) < 1-\epsilon$	$+$	$p_t(\theta)A_t$	no	$+$	✓
$p_t(\theta) < 1-\epsilon$	$-$	$(1-\epsilon)A_t$	yes	$-$	**0**
$p_t(\theta) > 1+\epsilon$	$+$	$(1+\epsilon)A_t$	yes	$+$	**0**
$p_t(\theta) > 1+\epsilon$	$-$	$p_t(\theta)A_t$	no	$-$	✓

Table 1: Table summarizing the behavior of PPO's objective function L^{CLIP} for all non-trivial cases, where both $p_t(\theta)$ and A_t are unequal zero. The first column indicates the value of the probability ratio $p_t(\theta)$, while the second column indicates whether the advantage estimate A_t is positive $(+)$ or negative $(-)$ for a given training example (indexed by subscript t) taken from a minibatch of training examples. The third column indicates the output of L^{CLIP}, i.e. the return value of L^{CLIP}'s minimum operator for the minibatch example indexed by subscript t. The fourth column indicates whether this term, i.e. the output of L^{CLIP}, is a clipped term (yes) or not (no). The fifth column indicates whether the sign of the value returned by L^{CLIP} is positive $(+)$ or negative $(-)$. The last column indicates whether the gradient resulting from back-propagating L^{CLIP} aims at maximizing the value returned by L^{CLIP} (✓) or whether only the trivial zero-gradient (**0**) results.

3.5 Exploration Strategies

In DRL, the *exploration-exploitation dilemma* refers to the problem of balancing how much a learning agent explores its environment by taking novel actions in the states it encounters and how much the agent chooses to exploit the knowledge it has gained throughout training thus far already [7]. If the agent explores the different effects that different actions have in given states, the agent might encounter more valuable behaviors than it has previously found [7]. On the other hand, if the agent over-explores its environment, this might lead to slow convergence to an optimal policy, even though over-exploring agents might still possibly converge to some local optimum in terms of an agent's learned policy [33]. Thus, good exploration-exploitation trade-off strategies are very important in order to ensure that the agent will eventually converge, in reasonable time, on some optimal policy after having sufficiently explored its state-action space in search for the most valuable actions in given states.

In PPO (and more generally in all REINFORCE [16] algorithms), exploration is naturally incorporated into the learning procedure by means of the stochastic policy π_θ, which stochastically samples actions instead of computing them solely deterministically as a function of given states. In the following, Sections 3.5.1 and 3.5.2 will describe how the exploratory behavior of a PPO agent's stochastic policy is regulated for continuous and discrete action spaces, respectively.

3.5.1 Exploration in Continuous Action Spaces

Recall from Section 3.2.1 that continuous actions are sampled from Gaussian distributions in PPO. Each Gaussian is parameterized by a mean and a standard deviation. Here, the only way of adjusting the expected spread of values to be sampled around the mean of a Gaussian distribution is to adjust the Gaussian's standard deviation. However, as stated in Section 3.2.1, in PPO the standard deviation of Gaussians is fixed throughout the training procedure. Thus, in practice there is no way of adjusting the exploratory behavior of a PPO agent except for adjusting the fixed standard deviation parameter manually before the start of the training procedure.

An alternative procedure, treating a Gaussian's standard deviation as a trainable parameter to be adjusted throughout training by means of stochastic gradient descent, is presented in [16].

3.5.2 Exploration in Discrete Action Spaces

In the case of discrete action spaces, an *Entropy bonus H* can be added to the policy network's overall objective function in order to enhance a PPO agent's exploratory behavior. Recall from Section 3.2.2 that, in the case of discrete action spaces, an action $a_t \in \mathscr{A}$ is sampled from a discrete action space $\mathscr{A}$ with respect to a Multinomial probability distribution parameterized by a probability vector ϕ_t assigning a probability of being sampled as action a_t to each of the elements in $\mathscr{A}$. In such a situation, maximal exploration is achieved when assigning equal probability to each of the elements contained in $\mathscr{A}$. An Entropy bonus rewards an agent's tendency to produce probability estimates over the action space $\mathscr{A}$ which make all available actions equally likely in a given state, thereby leading to many distinct actions being explored in the states encountered throughout training.

Using the notation introduced in this report, the Entropy of a Multinomial distribution parameterized for a single training example (taken from a minibatch of training examples) is defined as:

$$H^{full}(\phi_t, q_t, n) = -log(n!) - n \sum_{m=1}^{M} \phi_{t,m} \; log(\phi_{t,m}) + \sum_{m=1}^{M} \sum_{q_{t,m}=0}^{n} \binom{n}{q_{t,m}} \phi_{t,m}^{q_{t,m}} (1-\phi_{t,m})^{n-q_{t,m}} \; log(q_{t,m}!). \quad (15)$$

Recall that ϕ_t denotes a vector of normalized probability estimates (associated with a single training example taken from a minibatch of training examples), where the vector's m^{th} element, $\phi_{t,m}$, denotes the probability of selecting the action space's m^{th} element as action a_t in a given state s_t. The vector q_t contains the counts of how many times each element contained in action space $\mathscr{A} = \{1, 2, ..., M\}$ has been sampled in a given state s_t as action a_t. This vector's m^{th} element, $q_{t,m}$, denotes how many times the m^{th} element from $\mathscr{A}$ has been sampled in state s_t. Since exactly one action a_t is sampled in every state s_t, always exactly one element of q_t will be 1, while all other elements will be 0. The variable n is computed as the sum over vector q_t, and thus counts how many actions are sampled in total in a given state. Therefore, always $n = 1$. The term $\binom{n}{q_{t,m}}$ is the so-called binomial coefficient, being computed as $\binom{n}{q_{t,m}} = \frac{n!}{q_{t,m}!(n-q_{t,m})!}$. Given that always $n = 1$ and $q_{t,m} \in \{0, 1\}$, it can be shown that the binomial coefficient will always evaluate to 1 in the cases considered here. Likewise, here the term $\sum_{m=1}^{M} \sum_{q_{t,m}=0}^{n} \binom{n}{q_{t,m}} \phi_{t,m}^{q_{t,m}} (1 - \phi_{t,m})^{n-q_{t,m}}\ log(q_{t,m}!)$ will always evaluate to 0, since $q_{t,m} \in \{0, 1\}$, so that always $log(q_{t,m}!) = log(1) = 0$. Furthermore, since always $n = 1$, also $-log(n!)$ will always evaluate to 0. Thus, in the context of computing the Entropy bonus for a Multinomial distribution used to sample a single action a_t per state s_t, Equation 15 can be simplified to:

$$H^{Shannon}(\phi_t) = -\sum_{m=1}^{M} \phi_{t,m}\ log\ \phi_{t,m}, \tag{16}$$

which corresponds to the definition of the so-called *Shannon Entropy.* The evaluation of Equation 16, yielding the Entropy bonus for a single training example, is then averaged over a minibatch of training examples. This gives rise to the complete equation used for computing the Entropy bonus when training a PPO agent on a discrete action space. It looks as follows:

$$H(\phi) = \hat{\mathbb{E}}_t[-\sum_{m=1}^{M} \phi_{t,m}\ log\ \phi_{t,m}]. \tag{17}$$

Here, $\hat{\mathbb{E}}$ denotes the empirical expectation again and ϕ denotes a minibatch of normalized probability vectors ϕ_t. M refers to the number of elements, i.e. possible actions, in action space $\mathcal{A}$ and t is again used to denote the index of a training example taken from a minibatch of training examples.

3.6 Back-Propagation of Overall Objective Function

Recall that during the training of a PPO agent, both a stochastic policy network and a state value network get trained, with the latter being used to reduce the variance contained in the numeric estimates based on which the former gets trained [4]. Throughout Section 3, we have seen so far how the objective function $L^{CLIP+H+V} = L^{CLIP} + hH - vL^V$, presented in Equation 11 and being used to train a PPO agent in its entirety, decomposes. While the clipped objective function L^{CLIP} (see Equation 8) is used to train the policy network, the quadratic loss L^V (Equation 10) is used to train the state value network. In order to encourage exploration in the case of discrete action spaces, an Entropy bonus H (Equation 17) can be added to the overall objective.

Next, we will consider how these separate terms are back-propagated by working out the partial derivatives of the terms contained in the overall objective function with respect to the outputs produced by the policy network and the state value network. To make the following more practically oriented, we will consider the optimization procedure as one where a *loss function* has to be minimized. Thus, we will consider the computation of partial derivatives from a perspective where $-L^{CLIP+H+V}$ has to be minimized using stochastic gradient *descent.*

Concretely, the remainder of this subsection is structured as follows. In Section 3.6.1, it will be shown how to compute the partial derivative of the negative clipped objective function, $-L^{CLIP}$, with respect to the probability value $\pi_\theta(a_t|s_t)$ serving as input to the computation of $-L^{CLIP}$. This yields the definition of the partial derivative $\frac{\partial -L^{CLIP}}{\partial \pi_\theta(a_t|s_t)}$. The result obtained from the aforementioned derivation applies to both continuous and discrete action spaces, since the procedure is identical in both cases. Subsection 3.6.2 will show how to compute the partial derivative of $\pi_\theta(a_t|s_t)$ with respect to the output μ_t computed by the policy network in the case of continuous action spaces, yielding the definition of $\frac{\partial \pi_\theta(a_t|s_t)}{\partial \mu_t}$. Applying the chain rule for differentiation to $\frac{\partial -L^{CLIP}}{\partial \pi_\theta(a_t|s_t)}$ and $\frac{\partial \pi_\theta(a_t|s_t)}{\partial \mu_t}$ yields $\frac{\partial -L^{CLIP}}{\partial \mu_t}$ for continuous action spaces. Subsection 3.6.3 will show how to compute the partial derivative of $\pi_\theta(a_t|s_t)$ with respect to the outputs $\phi_{t,m}$ computed by the policy network in the case of discrete action spaces, yielding the definition of the partial derivatives $\frac{\partial \pi_\theta(a_t|s_t)}{\partial \phi_{t,m}}$. Applying the chain rule to $\frac{\partial -L^{CLIP}}{\partial \pi_\theta(a_t|s_t)}$ and $\frac{\partial \pi_\theta(a_t|s_t)}{\partial \phi_{t,m}}$ yields $\frac{\partial -L^{CLIP}}{\partial \phi_{t,m}}$ for discrete action spaces. Subsection 3.6.4 will show how to compute the partial derivative of L^V with respect to the output of the state value network. Subsection 3.6.5 shows how to compute the partial derivative of the negative Entropy bonus, $-H$, with respect to the normalized outputs produced by the policy network in the case of discrete action spaces.

To keep computations more tractable, Sections 3.6.1 through 3.6.5 will consider how to compute the aforementioned partial derivatives for only a single training example (taken from a minibatch of training examples) at a time. Furthermore, in the following, the logarithm operator *log* denotes the natural logarithm.

3.6.1 Back-Propagation of L^{CLIP}

This Subsection will show how to compute the partial derivative of the loss function $-L^{CLIP}$, i.e. $-1 * L^{CLIP}$, with respect to the probability value $\pi_\theta(a_t|s_t)$ serving as input to the evaluation of the aforementioned loss function.

Recall that L_t^{CLIP}, here being shown as the loss function associated with a single training example identified by subscript (i.e. index) t, is defined as follows:

$$L_t^{CLIP}(\theta) = min\left(p_t(\theta)A_t, clip(p_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right), \tag{18}$$

where $p_t(\theta)$ denotes the probability ratio being defined as $p_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$.

Equation 18 can be differentiated with respect to $\pi_\theta(a_t|s_t)$ as follows:

$$\frac{\partial - L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} = \frac{\partial - L_t^{CLIP}}{\partial L_t^{CLIP}} \left(\frac{\partial L_t^{CLIP}}{\partial p_t(\theta)A_t} \frac{\partial p_t(\theta)A_t}{\partial p_t(\theta)} + \frac{\partial L_t^{CLIP}}{\partial clip(p_t(\theta))A_t} \frac{\partial clip(p_t(\theta))A_t}{\partial clip(p_t(\theta))} \frac{\partial clip(p_t(\theta))}{\partial p_t(\theta)} \right) \frac{\partial p_t(\theta)}{\partial \pi_\theta(a_t|s_t)} \tag{19}$$

That is:

$$\begin{aligned} \frac{\partial - L_t^{CLIP}}{\partial \pi_\theta(a_t|s_t)} = -1 * \Bigg(& \begin{cases} 1 & \text{if } p_t(\theta)A_t \leq clip(p_t(\theta), 1-\epsilon, 1+\epsilon)A_t \\ 0 & \text{else} \end{cases} * A_t + \\ & \begin{cases} 1 & \text{if } clip(p_t(\theta), 1-\epsilon, 1+\epsilon)A_t < p_t(\theta)A_t \\ 0 & \text{else} \end{cases} * A_t * \\ & \begin{cases} 1 & \text{if } 1-\epsilon \leqslant p_t(\theta) \leqslant 1+\epsilon \\ 0 & \text{else} \end{cases} \Bigg) * \frac{1}{\pi_{\theta_{old}}(a_t|s_t)} \end{aligned} \tag{20}$$

Note that the definitions of Equations 19 and 20 imply that the partial derivatives of the minimum operator *min* with respect to its two input arguments are defined as:

$$\frac{\partial min(x,y)}{\partial x} = \begin{cases} 1 & \text{if } x \leqslant y \\ 0 & \text{else} \end{cases} \tag{21}$$

and

$$\frac{\partial min(x,y)}{\partial y} = \begin{cases} 1 & \text{if } y < x \\ 0 & \text{else} \end{cases} \tag{22}$$

Also, Equations 19 and 20 imply that the partial derivative of the clipping operator *clip* with respect to its input argument to be clipped is defined as:

$$\frac{\partial clip(x,a,b)}{\partial x} = \begin{cases} 1 & \text{if } a \leqslant x \leqslant b \\ 0 & \text{else} \end{cases} \tag{23}$$

Strictly mathematically speaking, the minimum operator is not differentialbe when its inputs x and y are equivalent. Likewise, mathematically speaking, the clipping operator is not differentiable when its first input argument x is equivalent to the inputs a or b defining the boundaries where clipping applies. However, to keep those operators differentiable in situations where their partial derivatives would not be defined otherwise, deep learning software packages like PyTorch make use of the partial derivatives provided in Equations 21 through 23 rather than using those being strictly mathematically correct.

3.6.2 Continuing Back-Propagation of L^{CLIP} in Continuous Action Spaces

In the following, it will be shown how to compute the partial derivative of the probability $\pi_\theta(a_t|s_t)$ with respect to the mean parameter μ_t computed by the policy network in the case of continuous action spaces.

Before stating the definition of $\frac{\partial \pi_\theta(a_t|s_t)}{\partial \mu_t}$, first note how the probability value $\pi_\theta(a_t|s_t)$ is obtained via a forward-pass through the policy network. To obtain $\pi_\theta(a_t|s_t)$, the policy network has to be evaluated on a given state s_t, where s_t is provided by a training example, indexed by subscript t, taken from a minibatch of training examples. Thereby, the mean parameter μ_t is computed, which is then used to parameterize a Gaussian distribution. Modern deep learning software packages like PyTorch or TensorFlow support obtaining the *log-probability* $log\ \pi_\theta(a_t|s_t)$ of selecting a given action a_t in state s_t by evaluating the Gaussian's probability density function (PDF) $g(a_t) = \frac{1}{\sigma_t\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{a_t-\mu_t}{\sigma_t})^2}$, to which the logarithm log has to be applied, on value a_t. Note that action a_t is also provided by the given training example and denotes the action that has been taken in state s_t during the previous training data generation step. Inserting $log\ \pi_\theta(a_t|s_t)$, i.e. $log\ g(a_t)$, into the exponential function exp yields the probability $\pi_\theta(a_t|s_t)$.

Notice from the above that computing the partial derivative $\frac{\partial \pi_\theta(a_t|s_t)}{\partial \mu_t}$ involves differentiating a Gaussian's PDF, to which the logarithm has been applied, with respect to mean parameter μ_t. The corresponding partial derivative $\frac{\partial log\ g(a_t)}{\partial \mu_t}$, which has previously been derived in [16], is defined as follows:

$$\frac{\partial log\ g(a_t)}{\partial \mu_t} = \frac{\partial log\ \pi_\theta(a_t|s_t)}{\partial \mu_t} = \frac{a_t - \mu_t}{\sigma_t^2}. \tag{24}$$

Using Equation 24, $\frac{\partial \pi_\theta(a_t|s_t)}{\partial \mu_t}$ is defined as follows:

$$\frac{\partial \pi_\theta(a_t|s_t)}{\partial \mu_t} = \frac{\partial \pi_\theta(a_t|s_t)}{\partial log\ \pi_\theta(a_t|s_t)} \frac{\partial log\ \pi_\theta(a_t|s_t)}{\partial \mu_t}, \tag{25}$$

where $\frac{\partial \pi_\theta(a_t|s_t)}{\partial log\ \pi_\theta(a_t|s_t)} = exp(log\ \pi_\theta(a_t|s_t))$. More concretely:

$$\frac{\partial \pi_\theta(a_t|s_t)}{\partial \mu_t} = exp(log\ \pi_\theta(a_t|s_t)) * \frac{a_t - \mu_t}{\sigma_t^2}. \tag{26}$$

3.6.3 Continuing Back-Propagation of L^{CLIP} in Discrete Spaces

In the following, it will be shown how to compute the partial derivatives of the probability $\pi_\theta(a_t|s_t)$ with respect to the normalized probability mass estimates $\phi_{t,m}$ computed by the policy network in the case of discrete action spaces.

Before stating the definition of the partial derivatives, first note how the probability value $\pi_\theta(a_t|s_t)$ is obtained via a forward-pass through the policy network. To obtain $\pi_\theta(a_t|s_t)$, the policy network has to be evaluated on a given state s_t provided by a training example, indexed by subscript t, taken from a minibatch of training examples. Thereby, the probability mass estimates $\phi_{t,1}$ through $\phi_{t,M}$ are computed, which are then used to parameterize a Multinomial distribution. Modern deep learning software packages like PyTorch and TensorFlow support obtaining the *log-probability* $log\ \pi_\theta(a_t|s_t)$ of selecting a given action a_t in state s_t by evaluating the Multinomial probability distribution's probability mass function (PMF), to which the logarithm log has to be applied, on value a_t. Here, a_t refers to the action taken in state s_t during the previous training data generation step. The probability $\pi_\theta(a_t|s_t)$ is obtained from $log\ \pi_\theta(a_t|s_t)$ by inserting $log\ \pi_\theta(a_t|s_t)$ into the exponential function exp. Then, one has to compute the set of partial derivatives of the probability $\pi_\theta(a_t|s_t)$ with respect to the probability mass estimates $\phi_{t,1}$ through $\phi_{t,M}$ computed in the forward pass. This involves computing the partial derivative of the Multinomial distribution's PMF with respect to the probability mass estimates $\phi_{t,1}$ through $\phi_{t,M}$. Continuing to use the notation introduced in Section 3.2.2, the PMF of a Multinomial distribution is defined as follows:

$$g(\cdot) = \frac{n!}{q_1! \cdots q_M!}\phi_{t,1}^{x_1} \cdots \phi_{t,M}^{x_M}, \tag{27}$$

where n denotes the total number of actions sampled in each state s_t. This is always 1 here, since only a single action a_t is sampled in each state s_t. Furthermore, q_1 through q_M denote how often each of the M

elements in the discrete action space $\mathscr{A}$ has been sampled in state s_t. The probabilities $\phi_{t,1}$ through $\phi_{t,M}$ indicate the probability of choosing the M elements contained in action space $\mathscr{A}$ respectively. Let ϕ_{t,a_t} be the probability of choosing action a_t in state s_t and let $q_{a_t} = 1$ indicate that action a_t has been sampled during the previous training data generation step in state s_t, while *all remaining* q_1 through q_M are 0, since neither of the other actions has been sampled in state s_t.

Simplifying Equation 27 based on the observations stated above yields:

$$g(\cdot) = \phi_{t,1}^{x_1} \cdots \phi_{t,a_t}^{x_{a_t}} \cdots \phi_{t,M}^{x_M} = \phi_{t,1}^{0} \cdots \phi_{t,a_t}^{1} \cdots \phi_{t,M}^{0} = 1 \cdots \phi_{t,a_t} \cdots 1 = \phi_{t,a_t}, \tag{28}$$

since $\frac{n!}{x_1! \cdots x_{a_t}! \cdots x_M!}$ becomes $\frac{1!}{0! \cdots 1! \cdots 0!} = 1$ and all q_1 through q_M, except for $q_{a_t} = 1$, are 0.

Applying the logarithm to Equation 28 yields:

$$log\ g(\cdot) = log(\phi_{t,a_t}). \tag{29}$$

Note that $\phi_{t,a_t} = \pi_\theta(a_t|s_t)$ and, given the simplifications done to Equation 27, that $log\ g(\cdot) = log\ \pi_\theta(a_t|s_t)$.

Differentiating $log\ g(\cdot)$ from Equation 29 with respect to the probability mass estimate of sampling the previously sampled action a_t results in:

$$\frac{\partial log\ g(\cdot)}{\partial \phi_{t,a_t}} = \frac{1}{\phi_{t,a_t}}. \tag{30}$$

Differentiating $log\ g(\cdot)$ with respect to the probability of sampling any alternative value from action space $\mathscr{A}$, which has not been sampled in state s_t during the training data generation step, results in:

$$\frac{\partial log\ g(\cdot)}{\partial \phi_{t,m \neq a_t}} = 0, \tag{31}$$

for all actions $m \in \mathscr{A}$, where $m \neq a_t$.

Finally, this results in the following partial derivatives:

$$\frac{\partial \pi_\theta(a_t|s_t)}{\partial \phi_{t,a_t}} = \frac{\partial \pi_\theta(a_t|s_t)}{\partial log\ \pi_\theta(a_t|s_t)} \frac{\partial log\ \pi_\theta(a_t|s_t)}{\partial \phi_{t,a_t}} = exp(log\ \pi_\theta(a_t|s_t)) * \frac{1}{\phi_{t,a_t}} \tag{32}$$

and

$$\frac{\partial \pi_\theta(a_t|s_t)}{\partial \phi_{t,m \neq a_t}} = \frac{\partial \pi_\theta(a_t|s_t)}{\partial log\ \pi_\theta(a_t|s_t)} \frac{\partial log\ \pi_\theta(a_t|s_t)}{\partial \phi_{t,m \neq a_t}} = exp(log\ \pi_\theta(a_t|s_t)) * 0 = 0 \tag{33}$$

3.6.4 Back-Propagation of state-value Network's Objective Function

The quadratic loss function, used to train the state value network on a a single training example indexed by subscript t is defined as $L_t^V = (V_\omega(s_t) - V_t^{target})^2$. The partial derivative of the quadratic loss function L_t^V with respect to the output $V_\omega(s_t)$ generated by the state value network is defined as follows:

$$\frac{\partial L_t^V}{\partial V_\omega(s_t)} = \frac{\partial (V_\omega(s_t) - V_t^{target})^2}{\partial V_\omega(s_t)} = 2 * (V_\omega(s_t) - V_t^{target}). \tag{34}$$

3.6.5 Back-Propagation Entropy Bonus in Discrete Action Spaces

In the following, it will be shown how to compute the partial derivative of the Entropy bonus in the case of discrete action spaces with respect to the outputs generated by the policy network. Again, the computations will be shown for a single training example, indexed by subscript t, at a time. Also, the Entropy bonus will be treated as a loss term to be minimized again.

Recall that computing the Entropy bonus for a Multinomial probability distribution in the case of discrete action spaces was done by evaluating the Shannon Entropy, stated in Equation 16, on the probability mass estimates $\phi_{t,1}$ through $\phi_{t,M}$ produced by the policy network for a given state s_t. How to compute $\phi_{t,1}$ through $\phi_{t,M}$ has been explained in Section 3.6.3. Next, it will be shown how to compute the partial derivatives of the negative Shannon Entropy, $-H_t$, with respect to the probability mass estimates $\phi_{t,1}$ through $\phi_{t,M}$. First, recall that the negative Shannon Entropy is defined as follows:

$$-H_t = \sum_{m=1}^{M} \phi_{t,m} \ log \ \phi_{t,m} = \phi_{t,1} \ log \ \phi_{t,1} + \cdots + \phi_{t,M} \ log \ \phi_{t,M}. \tag{35}$$

Differentiating Equation 35 with respect to the normalized output generated by the policy network's m^{th} output node yields:

$$\frac{\partial - H_t}{\partial \phi_{t,m}} = log \ \phi_{t,m} + 1. \tag{36}$$

3.7 Pseudocode

Algorithm 1 shows the pseudocode of the overall PPO algorithm.

4 Reference Implementation

To facilitate the understanding of the PPO algorithm, which has only been explained theoretically so far throughout this paper, a reference implementation has been produced. The reference implementation can be found at https://github.com/Bick95/PPO. The main objective while writing the code has been to deliver an easy to understand, but consequently less rigorously efficient and competitive implementation. In the following, the provided reference implementation will be introduced (Section 4.1) and a corresponding evaluation thereof will be presented (Section 4.2).

4.1 Description of Reference Implementation

When designing the provided reference implementation, two competing objectives had to be balanced. As indicated above, the main objective while writing the code has been to deliver an easy to read implementation to facilitate the reader's understanding of the PPO algorithm. The second objective, central to the development of the reference implementation, has been the implementation's ease of use, which involved a lot of added complexity to make it easy for an user to customize a PPO agent's training procedure. To balance those two aspects, the reference implementation has been designed in a very modular way. This makes it easier to observe the implementation of isolated parts of the overall DRL algorithm, while, at the same time, making it easy to exchange modules in order to adapt the implementation to different learning conditions. A strategic design choice has been to keep the code immediately concerned with training a PPO agent as clean and concise as possible, while outsourcing a lot of the involved complexity into separate parts of the code.

The produced reference implementation has been developed in Python using the PyTorch library. The implementation can be applied to the popular OpenAI Gym environments, of which a large variety can readily be found on OpenAI's website[2] or in OpenAI's corresponding GitHub repository[3]. Note that the implementation currently only supports the generation of a single action per time step. In the following, the design of the implementation will be described.

At the heart of the implementation lies the class *ProximalPolicyOptimization*, which is the class implementing the actual PPO agent. This class allows for training and evaluating an agent's policy featuring a policy network as well as a corresponding state value network.

[2] https://gym.openai.com/

[3] https://github.com/openai/gym

Algorithm 1 Proximal Policy Optimization (PPO) using Stochastic Gradient Descent (SGD)

Input: N = Number of parallel agents collecting training data, T = Maximal trajectory length, performance criterion or maximal number of training iterations, weighting factors v and h

```
$\pi_\theta \leftarrow newPolicyNet()$
$V_\omega \leftarrow newStateValueNetwork()$        ▷ Possibly parameter sharing with $\pi_\theta$
env $\leftarrow newEnvironment()$
optimizer $\leftarrow newOptimizer(\pi_\theta, V_\omega)$
number_minibatches = $\left\lfloor \frac{N*T}{minibatch\_size} \right\rfloor$        ▷ Compute number of minibatches per epoch
while performance criterion not reached or maximal number of iterations not reached do
    train_data $\leftarrow$ []
    // Training data collection step. Ideally to be parallelized:
    for actor = 1, 2, ..., N do
        train_data $\leftarrow$ []
        $s_{t=1} \leftarrow$ env.$randomlyInitialize()$        ▷ Reset environment to a random initial state
        // Let agent interact with its environment
        // for T time steps & collect training data:
        for $t$ = 1, 2, ..., T do
            $a_t \leftarrow \pi_\theta.generate\_action(s_t)$
            $\pi_{\theta_{old}}(a_t|s_t) \leftarrow \pi_\theta.distribution.get\_probability(a_t)$
            $s_{t+1}, r_t \leftarrow env.step(a_t)$        ▷ Advance simulation one time step
            train_data $\leftarrow$ train_data + $tuple(s_t, a_t, r_t, \pi_{\theta_{old}}(a_t|s_t))$
        // Use training data to augment each collected tuple
        // of training data stored in train_data:
        for $t$ = 1, 2, ..., T do
            $V_t^{target} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V_\omega(s_T)$
            $A_t = V_t^{target} - V_\omega(s_t)$
            train_data[$t$] $\leftarrow$ train_data[$t$] + $tuple(A_t, V_t^{target})$
    $optimizer.resetGradients(\pi_\theta, V_\omega)$
    // Update trainable parameters $\theta$ and $\omega$ for $K$ epochs:
    for epoch = 1, 2, ..., K do
        train_data $\leftarrow randomizeOrder($train_data$)$
        for mini_idx = 1, 2, ..., number_minibatches do
            M $\leftarrow getNextMinibatchWithoutReplacement($train_data, mini_idx$)$
            for example e $\in$ M do
                $s_t, a_t, r_t, \pi_{\theta_{old}}(a_t|s_t), A_t, V_t^{target} \leftarrow unpack(e)$
                _ $\leftarrow \pi_\theta.generate\_action(s_t)$        ▷ Parameterize policy's probabilit distribution
                $\pi_\theta(a_t|s_t) \leftarrow \pi_\theta.distribution.get\_probability(s_t)$
                $p_t(\theta) \leftarrow \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$
                $\phi_t \leftarrow \pi_{\theta_{stoch}}.get\_parameterization()$        ▷ To be computed in case of discrete action space
            $L^{CLIP} = \frac{1}{|M|} \sum_{t \in \{1,2,...,|M|\}} min(p_t(\theta)A_t, clip(p_t(\theta), 1-\epsilon, 1+\epsilon)A_t)$
            $L^V = \frac{1}{|M|} \sum_{t \in \{1,2,...,|M|\}} (V_\omega(s_t) - V_t^{target})^2$
            $H = -\frac{1}{|M|} \sum_{t \in \{1,2,...,|M|\}} \phi_t \ log \ \phi_t$        ▷ To be computed in case of discrete action space
            $L^{CLIP+V+H} \leftarrow -L^{CLIP} + v * L^V - h * H$
            optimizer.$backpropagate(\pi_\theta, V_\omega, L^{CLIP+V+H})$
            optimizer.$updateTrainableParameters(\pi_\theta, V_\omega)$
return $\pi_\theta$
```

Policy networks are implemented via the class *Policy* and feature three internal modules forming a processing pipeline. The first part is the *input module* of the policy network, the second part is the *output module* of the policy network, and the third part is an implementation of a Gaussian or Multinomial probability distribution provided by PyTorch. This processing pipeline of the three aforementioned parts works as follows. The input module consumes a state representation and transforms it into some intermediate representation. The resulting intermediate representation is then transformed into a parameterization for a probability distribution using the output module. The probability distribution provided by PyTorch is then parameterized using the parameterization produced by the output module. Actions are then drawn stochastically from the probability distribution. Alternatively, the entropy may be computed for a given probability distribution or the log-probability of a given action may be obtained after a probability distribution has been parameterized. In the following, these three parts will be explained in more detail.

The input module may be of type, i.e. class, *InCNN* or *InMLP*, where the former implements a convolutional neural network (CNN) architecture and the latter implements a feed-forward neural network (NN) architecture. Both modules may be customized to some degree. Having two different options for instantiating an input module lets the user apply the same PPO agent, only using different input modules inside an agent's *Policy*, to different observation spaces. When using a CNN, the agent may be applied to environments involving visual state representations. If the observation space is non-visual, an input module consisting of a feed-forward NN, as is the case when using an input module of class *InMLP*, may be more suitable. The provided reference implementation may infer an input module's required network architecture automatically from a provided Gym environment instance.

While there are multiple classes of input modules, there is only one class of output modules. This class is called *OutMLP* and implements a feed-forward NN. The number of output nodes inside an output module is automatically chosen in accordance with the requirements imposed by an environment.

Generally, the implementation currently only supports generating a single action a_t per state s_t. When facing an environment featuring a continuous action space, the output module computes a single mean μ_t to parameterize a one-dimensional Gaussian probability distributions (called *Normal* in the reference implementation). The Gaussian's standard deviation is a hyperparameter to be chosen by the experimenter. It may either be set to a fixed value (as is usually the case in PPO) or (linearly or exponentially) annealed between two values or trained using SGD. If the standard deviation is trainable, the policy network does not only compute a mean μ_t to parameterize a given Gaussian distribution, but also a second value taken to be a so-called *log standard deviation*, $log(\sigma_t)$. The log standard deviation is then transformed into a regular standard deviation σ_t by applying the exponential function to the log standard deviation, i.e. $\sigma_t = exp(log(\sigma_t))$. This is done to enforce non-negative standard deviations. The resulting standard deviation is then used to parameterize a given Gaussian distribution. How the procedure for training standard deviation works in detail, including back-propagation-paths, is explained in [16]. For Gym environments featuring discrete action spaces, a Multinomial probability distribution (called *Categorical* in the reference implementation) is parameterized given the outputs generated by the output module. In this case, an output module has as many output nodes as there are categories, i.e. elements, in the corresponding action space $\mathcal{A}$.

Since training a PPO agent involves both training a policy network and a state-value network, also a state-value network class has been implemented, which is called *ValueNet* in the reference implementation. A state-value network consists of both an input- and an output module, where the input module may be shared with the policy network, and the output module predicts a state's value given the intermediate representation produced by the input module.

To increase the training efficiency, the provided reference implementation makes use of vectorized Gym environments. Vectorization here refers to stacking multiple parallel Gym environments of the same kind together and letting multiple PPO agents interact with these environments in parallel. To be more precise, each agent interacts with its private instance of a Gym environment. This speeds up the training data generation step, since multiple agents can generate training data for the next update step in parallel rather than sequentially.

Note that some Gym environments, e.g. the implementations of the Atari 2600 environments[4], do not provide Markovian state representations by default. For those cases, functionality has been implemented to stack multiple consecutive non-Markovian state representations together in order to artificially build Markovian state-representations from the non-Markovian ones directly provided by these Gym environments. Note that state representations are called *observations* in the reference implementation.

Moreover, PyTorch's auto-differentiation capability is used to perform back-propagation.

Also, the *ProximalPolicyOptimization* class provides basic evaluation capabilities. These are two-fold. For a user to gain a subjective impression of the performance of a trained agent, the *ProximalPolicyOptimization* class allows for letting a trained agent act in its environment while visually displaying, i.e. rendering, the environment. Alternatively, for a more objective analysis, basic quantitative analysis of the learning outcome of a PPO agent is supported in that the implementation allows for collecting and saving basic statistics concerning the training progress and corresponding evaluations. This will be demonstrated in the following subsection.

Furthermore, for convenience, configuration ("config") files may be used to specify in which configuration to train a PPO agent. Numerous hyperparameter settings may be controlled by adapting some example configuration files or adding new ones. Moreover, paths may be specified for saving or loading an agent's trained network architectures. This may be useful for saving trained agents and visually inspecting their performance at a later time point. Note also that both policy- and state-value networks may be saved to possibly continue training later.

Inline-comments have been added to the reference implementation to facilitate better understanding of the produced code. Again, the reference implementation can be found at https://github.com/Bick95/PPO.

4.2 Evaluation of Reference Implementation

In the following, the provided reference implementation will be evaluated. For this, PPO agents have been trained on two different OpenAI Gym environments. Below, the quantitative analysis of the training outcomes will be shown. Also a short discussion of the observed results will be provided.

In a first task, PPO agents have been trained on the OpenAI Gym MountainCarContinuous-v0 environment featuring a continuous action space as well as a continuous observation space. In this environment, an agent steers a car, placed between two surrounding mountains, to drive forth and back. The agent's goal is to control the car in such a way that the car gains enough momentum to be able to reach the flag at the top of one mountain. In this environment, agents have been trained for 3 million state transitions. In one training condition, the standard deviation of the Gaussian, from which actions are sampled, has been fixed. In a second training condition, the standard deviation has been a trainable parameter, being trained using SGD (as explained in Section 4.1). The configuration files, specifying exactly how the agents have been trained and tested on the given task, using either a fixed or trainable standard deviation, can be found on GitHub. In both training conditions, i.e. fixed versus trainable standard deviation, both a stochastic and a deterministic evaluation have been performed. During a stochastic evaluation, actions a_t are stochastically sampled from a Gaussian distribution, while actions are selected using a standard deviation of 0 during a deterministic evaluation. Furthermore, the results reported below have been obtained by training ten independent agents per training condition and averaging the ten independent results per testing condition.

For each testing condition, two metrics have been measured after the end of the training procedure. One metric is the total reward accumulated over 10,000 time steps and the second reward is the total number of restarts of an agent's environment during the 10,000 aforementioned time steps performed during the final evaluation. Here, a higher number of restarts indicates that an agent has achieved its goal more frequently. The accumulated reward increases every time that the car controlled by the agent reaches its goal, while it is decreased as a function of the energy consumed by the car. Tables 2 and 3 show the total number of restarts and the total accumulated rewards (both averaged over ten independent test runs), respectively.

[4] https://gym.openai.com/envs/atari

		Evaluation	
		Stochastic	**Deterministic**
Standard Deviation	**Fixed**	52.5 (21.28)	46.1 (25.79)
	Trainable	43.3 (21.69)	36.6 (20.53)

Table 2: Total number of restarts of an agent's environment within 10,000 time steps performed during the final evaluation. The results have been averaged over 10 randomly initialized test runs. Measurements have been taken in the OpenAI Gym MountainCarContinuous-v0 environment featuring both a continuous action and state space. The corresponding training and testing configuration files for *fixed* and *trainable* standard deviations can be found *here* and *here*, respectively.

First, consider the total number of restarts under the four different testing conditions presented in Table 2. When only considering stochastic evaluations, no statistically significant difference can be found between the two training conditions, i.e. when comparing the results obtained using a fixed standard deviation to those obtained using a trainable standard deviation. Likewise, when only considering deterministic evaluations, also no statistically significant difference can be found between the two training conditions.

When fixing the standard deviation and comparing the total number of restarts between the stochastic and deterministic evaluation, also no statistically significance is found. Moreover, no statistically significant difference is found when training the standard deviation and comparing the total number of restarts between the stochastic and deterministic evaluation.

When performing the same set of comparisons, as explained above for the total number of restarts, to the total accumulated rewards, as shown in Table 3 below, the same results, i.e. no statistically significant differences, are found.

In conclusion, this means the following. While there appear to be relevant differences in the quality of the learning outcome depending on whether the standard deviation is fixed or trained, those differences are not statistically significant at the 0.05 level. Also, there appear to be relevant differences in the learning outcome depending on whether the final evaluation is performed stochastically or deterministically. However, also those differences are not statistically significant at the 0.05 level.

While the results provided above suggest that there is no measurable advantage of training the standard deviation as opposed to keeping it constant, as is commonly done in PPO, there may still be reasons to train the standard deviation. Note that the fixed standard deviation value, used in the experiments presented above, has been obtained by inspecting to which value the standard deviation converged during one training run while treating the standard deviation as a trainable parameter. Prior trial and error search of appropriate settings of the fixed standard deviation parameter had not been successful. The manually tested values led to over-exploration or under-exploration. Thus, in cases where choosing a standard deviation parameter is a challenging task, treating the standard deviation as a trainable parameter might reveal appropriate choices for a fixed standard deviation value.

		Evaluation	
		Stochastic	**Deterministic**
Standard Deviation	**Fixed**	3609.84 (2360.32)	3899.43 (2783.13)
	Trainable	-4.61 (1.46)	-23703.79 (83220.94)

Table 3: Total number of accumulated rewards received from an agent's environment within 10,000 time steps performed during the final evaluation. The results have been averaged over 10 randomly initialized test runs. Measurements have been taken in the OpenAI Gym MountainCarContinuous-v0 environment featuring both a continuous action and state space. The corresponding training and testing configuration files for *fixed* and *trainable* standard deviations can be found *here* and *here*, respectively.

In a second task, PPO agents have been trained on the OpenAI Gym CartPole-v0 environment featuring a discrete action space and a continuous observation space. The goal of an agent in this environment is to command a cart to move horizontally to the left or right, such that a pole, placed vertically on top of the cart, remains balanced without falling over to the left or right. In this environment, rewards are emitted for every time step that the pole remains balanced without falling over. Since environments get immediately restarted as soon as an agent fails on the given task, only the total number of restarts serves as a sensible metric to assess an agent's learning outcome in this environment. Here, a lower number of restarts indicates that the agent has successfully managed balancing the pole for longer periods of time before an environment had to be restarted. Ten randomly initialized agents have been trained on this task for 200,000 time steps each. The aforementioned metric, i.e. the total number of restarts, has been measured during a stochastic and a deterministic evaluation (for 10,000 time steps each) per trained agent. During a stochastic evaluation, actions have been sampled stochastically, while always the action associated with the highest probability mass in a given state has been chosen in the deterministic evaluation condition. The measurements reported below have been computed as the average over the ten independent test runs per testing condition. The whole training and testing configuration can be found in the corresponding configuration file on GitHub. The results are as follows.

During the stochastic evaluation, 78.0 restarts have been observed on average (with a standard deviation of 18.35). During the deterministic evaluation, 53.7 restarts have been observed on average (with a standard deviation of 5.08). These findings are significantly different (p-value: $0.00078 < 0.05$).

This means that, in at least one of two tasks considered in this report, using a deterministic policy during the evaluation of an agent has led to significantly better evaluation results than performing the evaluation on the same policy run stochastically.

The aforementioned observation is important for the following reason. Contemporary research in the field of DRL sometimes mainly focuses on comparing the learning speed of agents [4] or the average scores obtained during training of an agent [6]. These are metrics being computed based on the performance of policies run stochastically. However, in real life applications, there may be situations where running a policy stochastically may result in catastrophic errors, such that in some occasions policies might have to be run deterministically after the end of the training phase. For example, consider the physical damage that may arise from a robot performing surgery on a patient based on a stochastic policy. The results presented above seem to suggest that assessing the quality of the learning outcome of a DRL algorithm purely based on the results obtained during the evaluation of a stochastic policy might not accurately reflect the results that would be obtained when running a resulting policy deterministically. Given the above considerations, it might be a valuable contribution to the field of DRL in the future if researchers come to focus more strongly on the differences between running policies in a stochastic and a deterministic mode after the end of training.

5 Considerations and Discussion of PPO

Throughout this report, the PPO algorithm has been presented and explained in a lot of detail. By now, the reader is assumed to know in detail how the algorithm works. Furthermore, the Introduction (see Section 1) and later Sections listed some reasons for using PPO, thereby justifying the importance of giving a detailed, thorough explanation of the algorithm for the first time in this report. Some of these reasons for using PPO were its comparatively high data efficiency, its ability to cope with various kinds of action spaces, and its robust learning performance [6]. However, since this report aims at providing a neutral view on PPO and the field of DRL in general, this section will address some critical considerations concerning PPO and related methods. Also, the field of DRL will be considered from a broader perspective.

Particularly, the first subsection, Section 5.1, will consider reasons for why PPO might not always be the most suitable DRL algorithm. Critical aspects associated with PPO will be discussed.

The second subsection, Section 5.2, will be concerned with some macro-level considerations, addressing the question whether using PPO or comparable methods might lead to the emergence of General Artificial Intelligence (GAI) at some point in the future.

5.1 Critical Considerations concerning PPO

The following will address some of the limitations of the PPO algorithm.

In the original paper, the inventors of PPO argue that PPO's main objective function, L^{CLIP}, is designed to prevent excessively large weight (i.e. trainable parameter) updates from happening. This is because L^{CLIP} is supposed to remove the incentive for moving the probability ratio $p_t(\theta)$ outside a certain interval within a single weight update step consisting of multiple epochs of weight updates. That this procedure will not entirely prevent destructively large weight updates from happening is obvious already from the fact that there is no hard constraint enforcing this condition. The authors of [15] analyzed the effectiveness of L^{CLIP} in preventing weight updates, which would effectively move the probability ratio outside the interval $[1-\epsilon, 1+\epsilon]$, from happening. They found that L^{CLIP} had at least some effect in restricting the evolution of $p_t(\theta)$, but generally failed to contain $p_t(\theta)$ strictly inside the interval $[1-\epsilon, 1+\epsilon]$. As a possible solution to this problem, the authors of [15] proposed a variant of PPO, which they call *Trust Region-based PPO with Rollback* (TR-PPO-RB). According to [15], TR-PPO-RB exhibits better learning performance and higher sample efficiency across many learning tasks compared to vanilla PPO.

Speaking about data efficiency, it must be mentioned that PPO's sample efficiency is comparatively low. On the one hand, PPO's sample efficiency is indeed higher than that of many other policy gradient methods (PGMs). This is because PPO allows for multiple epochs of weight updates using the same freshly sampled training data, whereas many other PGMs may perform only a single epoch of weight updates on the obtained training data [6]. On the other hand, PGMs, serving as a means of reference here and commonly being on-policy methods [25], are generally associated with lower data efficiency than off-policy methods [25], which makes the whole aforementioned comparison between PPO and other PGMs look less spectacular. This makes PPO a less-optimal choice when facing learning tasks, where training data is expensive or difficult to obtain. In such situations, more sample efficient DRL algorithms might be more suitable to use.

Also, it must be mentioned that PPO, being an on-policy method, is only applicable to learning tasks being on-policy compatible.

Another no-trivial aspect about PPO is hyperparameter tuning. In many cases, PPO performs reportedly well without performing much parameter tuning [15]. However, in cases where parameter tuning is still required, this task is non-trivial, since there is no intuitive way of determining whether, for example, larger or smaller values for the hyperparameter ϵ would improve a PPO agent's learning performance as well as possibly its sample efficiency. Likewise, there is no way of determining a suitable number of epochs (of weight updates) per weight update step in advance. Those values possibly have to be fine-tuned using the expensive method of parameter sweeping when the training outcome is worse than expected or desired.

A related issue concerns the setting of the standard deviation hyperparameter σ when sampling actions from continuous action spaces. As reported above (see Section 3.2.1), the inventors of PPO proposed to set the standard deviation to a fixed value. The question arises what justifies setting the standard deviation to some fixed value. This question arises because there is no reason provided for this particular choice of determining the standard deviation hyperparameter. A problem related to having a fixed standard deviation is that a standard deviation of a certain value might be a very small or very large value, depending on the action space at hand. If the standard deviation is fixed to a comparatively small value, this might hinder exploration of the state-action space. If the standard deviation is fixed to a comparatively large value, this might lead to over-exploration of the state-action space, thus slowing down convergence of the policy. A possible alternative to the proposed way of fixing a Gaussian's standard deviation treats the standard deviation as a trainable parameter [16], as explained in Section 4.1. While this way of training the standard deviation has been motivated in the literature [16], the evaluation of the reference implementation, provided in Section 4.2, failed to demonstrate a measurable advantage of training the standard deviation as opposed to keeping it constant, as is commonly done in PPO. However, as reported in Section 4.2, training the standard deviation might still reveal appropriate choices for a fixed standard deviation value.

Another aspect to be mentioned about PPO concerns the circumstance that PPO is a model-free DRL algorithm. Recall that *model-free* refers to a DRL agent not learning an explicit model of the environment it is situated in, while model-based DRL algorithms learn some form of explicit representation of the environment surrounding them [5]. As argued in [5], model-based DRL algorithms stand out due to their enhanced capability of transferring generalized knowledge about their environment (encoded in their world model) between tasks. Also, model-based DRL algorithms may perform planning by simulating potential future outcomes of their present and future decision making and may even learn to some extent from offline datasets [5]. PPO, on the contrary, being a model-free DRL algorithm, naturally lacks these features.

5.2 RL in the Context of Artificial Intelligence (AI)

In the following, the question will be touched upon whether RL may be a suitable means of developing General Artificial Intelligence (GAI) in the future.

In a recent paper, Silver et al. [34] argue that all there is needed in order to form intelligent behavior in an agent is a reward metric which is to be maximized through some learning procedure implemented by an agent. More specifically, their hypothesis, called *Reward-is-Enough*, states that [34]:

Hypothesis 1 (Reward-is-Enough): *Intelligence, and its associated abilities, can be understood as subserving the maximisation of reward by an agent acting in its environment.*

The idea behind the *Reward-is-Enough* hypothesis is as follows. If an agent is presented to some environment in which it has to learn to act in a way such that a given cumulative reward metric gets maximize, the agent will ultimately discover increasingly complex behaviors in order to achieve the goal of maximizing the given reward metric. Thereby, on the long run, an agent will develop enough sophisticated abilities to be eventually considered intelligent. This hypothesis is relevant to the topic considered here, since Silver et al. [34] argue that the reward maximization task discussed in their paper is perfectly compatible with the concept of RL. Thus, according [34], RL may possibly give rise to the emergence of GAI in the future.

Silver et al. [34] also argue that their hypothesis might even give an appropriate account of the emergence of natural intelligence present in animals, including the human kind. The approach of trying to explain the emergence of natural intelligence as a by-product of solving a singular problem, namely that of evolutionary pressure, has already been adopted in early work on Artificial Intelligence [35] and in the field of Evolutionary Psychology [36]. Thus, there seems to be a lot of support for believing in this hypothesis as a likely solution to the questions of how and why intelligence has ultimately arisen in nature.

However, when using the *Reward-is-Enough* hypothesis to explain both the emergence of natural intelligence and how to arrive at GAI (e.g. through RL) in the future, one must beware of a subtle difference between those two cases. When talking about the evolution of natural intelligence, one considers the evolutionary process happening to an entire population of individuals, but not the evolution of intelligence within an individual. When considering the future emergence of GAI in the context of the *Reward-is-Enough* hypothesis, however, one concerns oneself with the question of how GAI may emerge within a single artificial agent having a possibly infinite life span. This difference is due to the following reason. There is some evidence suggesting that human intelligence is dependent on biological factors [37, 38]. Thus, if the level of general intelligence is prescribed to a biological being, e.g. by a so-called factor g [37], natural intelligence cannot be *caused* by reward maximization within an individual at the same time. Note that this subtle difference, which is also acknowledged by Silver et al. [34], is of importance in that it indicates that the success of natural evolution in forming natural intelligence cannot be seen as direct support for the plausibility of the *Reward-is-Enough* hypothesis in the context of developing GAI. Instead, the truth of this broad hypothesis will ultimately have to be demonstrated by providing a proof of concept, i.e. an example demonstrating the practical working of this hypothesis.

Especially since RL is said to be a suitable means of testing the *Reward-is-Enough* hypothesis [34], providing a practical implementation demonstrating the truth of this hypothesis is of particular importance, since no case is publicly known yet in which GAI has emerged from training RL agents in spite of the large body of corporate and academic research that has been conducted on the field of RL already.

In the following, I would like to point out a few more theoretical considerations challenging the idea that GAI may arise from RL.

First of all, recall that contemporary research on RL draws upon (recurrent) neural network architectures to implement agents' decision making strategies. So far, it is an open question whether the model capacity of network architectures that can possibly be trained on today's available hardware is sufficiently large to accommodate decision making strategies causing truly intelligent behavior. Also, ultimately all vanilla (recurrent) neural network architectures can be seen as singular, sequential streams of information processing. On the contrary, biological brains (being the only known source of true intelligence yet) are composed of up to billions of neurons [38], which have often been observed to form clusters being associated with dedicated cognitive abilities [38], working in parallel. From this point of view, one may ask whether utilizing sequential streams of information processing only, not drawing upon distributed system architectures allowing for truly parallel information processing as is happening in biological brains as well, is an appropriate approach to seek the emergence of GAI. Also, as argued in the literature [35], evolutionary processes often fail to deliver the most efficient solutions to certain problems. Thus, approaching the problem of developing GAI from an engineering perspective, rather than from an evolutionary perspective, might potentially lead to more efficient solutions eventually.

6 Conclusion

This report began by giving a short introduction into the field of Reinforcement Learning (RL) and Deep Reinforcement Learning (DRL), particularly focusing on policy gradient methods (PGMs) and the class of REINFORCE algorithms. Then, Proximal Policy Optimization (PPO), largely following the principles of REINFORCE, has been introduced, pointing out the poor documentation of this algorithm. Acknowledging the importance of PPO, however, this report continued explaining the PPO algorithm in minute detail. Afterwards, an easy to comprehend reference implementation of PPO has been introduced and assessed. Finally, some critical remarks have been made made about the design of PPO and its restricted applicability. Also, the question whether RL may lead to the emergence of General Artificial Intelligence in the future has been addressed. Given the undisputed importance of RL, this report concludes by once again pointing out the importance of delivering adequate documentation of RL algorithms to be introduced in the future. Acknowledging the amount of future work that is still to be done in advancing the field of RL, researchers working in this field to the present date ought present their findings and proposed methods in a way that can be well understood not only by other experts with life-long experience in their field, but also by tomorrow's scientists just about to dive into the broad and exciting field of RL.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

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

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* MIT press, 2018.

[4] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, pp. 1928–1937, 2016.

[5] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, "Mastering atari with discrete world models," *arXiv preprint arXiv:2010.02193*, 2020.

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

[7] L. Rejeb, Z. Guessoum, and R. M'Hallah, "The exploration-exploitation dilemma for adaptive agents," in *Proceedings of the Fifth European Workshop on Adaptive Agents and Multi-Agent Systems*, Citeseer, 2005.

[8] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, *et al.*, "Noisy networks for exploration," *arXiv preprint arXiv:1706.10295*, 2017.

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

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, pp. 1097–1105, 2012.

[11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[12] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[13] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[14] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[15] Y. Wang, H. He, and X. Tan, "Truly proximal policy optimization," in *Uncertainty in Artificial Intelligence*, pp. 113–122, PMLR, 2020.

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

[17] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *International Conference on Machine Learning*, pp. 1329–1338, PMLR, 2016.

[18] C. J. C. H. Watkins, *Learning from Delayed Rewards.* PhD thesis, 1989.

[19] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[20] J. Peters and J. A. Bagnell, *Policy Gradient Methods*, pp. 1–4. Boston, MA: Springer US, 2016.

[21] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems*, pp. 1008–1014, Citeseer, 2000.

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

[23] J. Ba, V. Mnih, and K. Kavukcuoglu, "Multiple object recognition with visual attention," *arXiv preprint arXiv:1412.7755*, 2014.

[24] S. Gu, T. Lillicrap, Z. Ghahramani, R. E. Turner, B. Schölkopf, and S. Levine, "Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning," *arXiv preprint arXiv:1706.00387*, 2017.

[25] J. P. Hanna and P. Stone, "Towards a data efficient off-policy policy gradient.," in *AAAI Spring Symposia*, 2018.

[26] S. Paternain, J. A. Bazerque, A. Small, and A. Ribeiro, "Stochastic policy gradient ascent in reproducing kernel hilbert spaces," *IEEE Transactions on Automatic Control*, vol. 66, no. 8, pp. 3429–3444, 2021.

[27] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning*, pp. 1889–1897, PMLR, 2015.

[28] J. S. Bridle, "Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters," in *Advances in Neural Information Processing Systems*, pp. 211–217, 1990.

[29] J. Peng and R. J. Williams, "Incremental multi-step q-learning," in *Machine Learning Proceedings 1994*, pp. 226–232, Elsevier, 1994.

[30] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[31] S. Kakade and J. Langford, "Approximately optimal approximate reinforcement learning," in *In Proc. 19th International Conference on Machine Learning*, Citeseer, 2002.

[32] Z. Liu, B. Wu, W. Luo, X. Yang, W. Liu, and K.-T. Cheng, "Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 722–737, 2018.

[33] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[34] D. Silver, S. Singh, D. Precup, and R. S. Sutton, "Reward is enough," *Artificial Intelligence*, vol. 299, p. 103535, 2021.

[35] R. Davis, "What are intelligence? and why? 1996 aaai presidential address," *AI Magazine*, vol. 19, no. 1, pp. 91–91, 1998.

[36] L. Cosmides and J. Tooby, "Evolutionary psychology: A primer," 1997.

[37] T. J. Bouchard, "Genes, evolution and intelligence," *Behavior Genetics*, vol. 44, no. 6, pp. 549–577, 2014.

[38] J. W. Kalat, *Biological Psychology.* Boston, MA: Cengage Learning, 2019.