university of groningen

faculty of science and engineering

Department of Artificial Intelligence

EXPLORING DEEP REINFORCEMENT LEARNING FOR CONTINUOUS ACTION CONTROL

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

Author
Athanasios Trantas

Supervisors
Dr. Hamidreza Kasaei
Prof. Dr. Bart Verheij

In memory of
Dr. Marco Wiering

Groningen, 27 October 2021

Acknowledgments

This study would not be the same without the contribution of specific people.

First, I would like to thank my first supervisor, Marco Wiering, for his constant support, academic guidance, and mentorship. He made me love the field of Reinforcement Learning and extend my machine learning skills towards autonomous and multi-agent systems. Following his advice I discovered fundamental principles of Artificial Intelligence. The legacy he left behind will be a fountain of inspiration for many years to come.

My second supervisor, Hamidreza Kasaei, also deserves my infinite gratitude. He always have an insightful answer to all my questions, providing significant support and advice through this project. Additionally, he gave me unparalleled knowledge through his robotic courses while extended my research interest towards application of AI into real-world tasks.

I would also like to notably mention, Bart Verheij, who taught me GOOD AI. Special thanks for the constructive discussions and his valuable suggestions at the early stages of this thesis. Plus, for his willing accept to support me finishing this thesis on the most difficult moment.

In addition, this research would have been impossible without the ceaseless support of my parents, Dimitris and Anastasia, all my study years and in every possible way. I am also extremely grateful to my brother John for all his efforts to cheer me up and boost my morale all these years, especially during the Covid-19 wave.

A huge “Thank you!” to my partner Nikolina who stood rock next to me the past 7 years, boosting me to chase my dreams.

Last but not least, I would like to thank the Center for Information and Technology of the University of Groningen for their support when using the Peregrine high performance computer cluster.

Abstract

Reinforcement Learning is a learning framework or toolbox with mathematically proven computational tools and methods to understand, calculate and automate goal-directed learning and decision making. Reinforcement Learning has been involved in some of the most remarkable developments in Artificial Intelligence mostly “Deep Reinforcement Learning”; reinforcement learning with function approximation by deep artificial neural networks. In this thesis, Deep Reinforcement Learning for continuous action control is investigated. In contrast with discrete action spaces, having unlimited actions is scary but challenging in parallel. In this study, action selection derives from a deterministic policy. For this reason, a model-free, off-policy, actor-critic algorithm named Sample Policy Gradient is extended and benchmarked. Specifically, a prioritize buffer is used to store the experience and a regularizing term is added to its objective function. Extensive experimentation is analyzed, using two simulators to test and evaluate the performance of the algorithm. To wrap up this part, a performance study and a qualitative comparison with the state-of-the-art algorithms are presented. Additionally, an important aspect of Reinforcement Learning namely Safe Reinforcement Learning is investigated. To unlock the full potential of Reinforcement Learning and apply it to daily life, it is necessary to embed the agents into the data generation distribution, which is the real-world experience. The agents should have the ability to learn in our world and not only in the simulator. To do so, safety must be ensured and subsequently, the agents need to satisfy constraints. In this study, Safe Reinforcement Learning and more specific Safe Exploration is scrutinized. The Constrained Markov Decision Process framework is built and a novel solution to the constrained optimization problem with transfer learning and function approximation is provided. Finally, the safe agent is tested and evaluated on a third simulator and empirical results are presented.

***Keywords*—**Deep Reinforcement Learning, Safe Reinforcement Learning, Safe Exploration, Continuous Action Control, Sample Policy Gradient.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

A2C	Actor-Critic
CMDP	Constrained Markov Decision Process
DRL	Deep Reinforcement Learning
DL	Deep Learning
DDPG	Deep Deterministic Policy Gradient
MDP	Markov Decision Process
ML	Machine Learning
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SL	Safety Layer
SPG	Sample Policy Gradient
SPGR	Sample Policy Gradient Regularized
SRL	Safe Reinforcement Learning
TD3	Twin Delayed Deep Deterministic Policy Gradient
VPG	Vanilla Policy Gradient

1 Introduction

1.1 Reinforcement Learning

Reward is enough to drive behaviour that exhibits abilities studied in natural and artificial intelligence, including knowledge, learning, perception, social intelligence, language, generalisation and imitation. (Silver et al., 2021)

One of the four learning frameworks along with Supervised Learning (SL), Unsupervised Learning (UL) and Self-Supervised Learning (SSL) (Yann LeCun, 2021), Reinforcement Learning (RL) provides the computational tools and methods to understand, calculate and automate goal-directed learning and decision making. RL can be characterized as a toolbox of methods to create intelligent agents/controllers for almost all kinds of modern computational tasks. One precise definition, given by Sutton & Barto (2018) is “learning what to do–how to map situations to actions–so as to maximize a numerical reward signal”. The notation of this research is adapted to the books of (R. S. Sutton & Barto, 2018), (Wiering & Van Otterlo, 2012) and (Achiam, 2018). Their work is extensive and spans in a wide range of computational topics in Reinforcement Learning, thus provides the best source of numerical tools needed to adequately explain the biggest part of this study’s theory .

1.2 Research Questions

In this thesis we aim to answer the following research questions:

1. How can we increase the performance of deterministic algorithms like SPG?
2. What performance cost must be paid and how hard is it to solve the policy optimization problem?
3. Does bias overestimation hurts the performance and convergence of SPG?
4. How can we achieve Safe RL for continuous action control?
5. Can we adapt SPG algorithm to incorporate safety constraints? If we can, how competitive is its performance?
6. Can we optimize our policies to achieve minimum number of constraint violations? If we can, what performance cost must be paid and how hard is to solve the constrained policy optimization problem?

1.3 Significance of the study

This study provides the basic theoretical foundations of Reinforcement Learning and extends to Deep Reinforcement Learning and Safe Reinforcement Learning, thus yields a compact manual for the interested reader. First, the community can be benefited from the presentation of a new, well-studied optimization Policy Gradient method that achieves comparable to state-of-the-art results. Furthermore, two new additional methods are studied. First, prioritize replay buffer that can be used by all off-policy methods that learn from examples. Second, policy regularization combats the extrapolation error which always exists when sampling from a buffer rather than from the data generating distribution. Adding on that, bias overestimation is studied that emerges when using artificial neural networks for function approximation. Subsequently, the results from our effort to eliminate it are provided.

The community can also find the second part of this research informing and valuable because it explores Safety in the context of Reinforcement Learning and provides a novel solution to the constrained optimization problem. Imagine for a moment an industrial robot arm learning to assemble a product in a factory. Current work in RL allows the agents (robot arm) to explore freely any behaviour during learning as long as the performance increases. Some of these behaviours could cause the robot arm to damage itself or its surroundings. This kind of behaviours are unacceptable and could be avoided by safe exploration. Thus, Safe RL can be considered as a prerequisite in order to deploy RL in real-world applications. In the upcoming years, this technology will inevitably be widely used on daily basis by a lot of people in applications like autonomous cars, robotic assistants, dialogue systems and many more to be invented. Because of the daily basis interactions of these agents with people and other agents, safety must come first and be ensured by the system designer. Following (Dulac-Arnold et al., 2019), many of the challenges the real-world RL comes with are addressed; but the main focus is placed on Safe RL and how it can be evolved to an **anchored baseline** that will guide the development of intelligent agents in the future.

1.4 Thesis Layout

This thesis is structured as follows. In the first chapter we introduce the research topic of this study along with the research questions and its significance for the community. In the second chapter we present the theory of Reinforcement Learning in a compact but descriptive way. In the third chapter we explore policy optimization and discuss some of the algorithms that shaped the field. Moving to the fourth chapter we present our algorithm named Sample Policy Gradient. In the fifth chapter we introduce Safe Reinforcement Learning, discuss safe exploration and constrained RL while also presenting two methods to solve the constrained optimization problem. On the sixth chapter, we describe the experiment setup and the simulators used to evaluate our algorithms. In the seventh chapter we present the results from our experimentation. In the final chapter eight, we conclude this thesis with a short discussion, first by provide answers to the research questions and then add possible future directions for this study.

2 Theory

2.1 Agent-Environment Interface

Simple at first sight, but computationally feasible and trackable, this interface grounds the problem of learning by interacting in order to achieve a goal. The learner and the one who takes decisions is called **agent**. The medium which interacts with, comprising of everything the agent cannot control is considered the **environment**. These two pieces interact constantly, the agent choosing actions and the environment reacting to these actions and presenting new situations to the agent. Furthermore, the environment gives birth to **reward**, special scalar valued signal that the agent's goal is to maximize cumulative over time through its action selection, called **return**. This number tells the agent how good or bad the present environment state is and it will be further explained in the next sections. It should be noted that the terms agent, environment and action are used instead of the engineering terms controller, controlled system (plant) and control signal. An illustration of the interface presented above can be found of Figure 2.1.

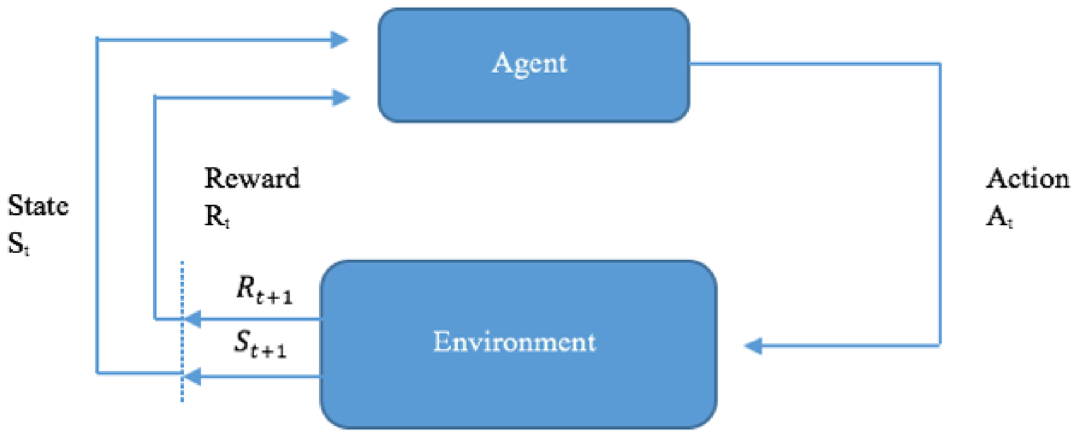


Figure 2.1: The agent-environment interaction in a Markov decision process.

2.2 Markov Decision Process Framework

In order to solve the RL problem, the need for a mathematical formalization framework of sequential decision making led to Markov Decision Processes (MDPs). In this setting, actions influence not just immediate rewards, but also forthcoming situations or states and having them as a medium, future rewards. Using the definition and notation from (R. S. Sutton & Barto, 2018), an MDP is a 5-tuple $< S, A, R, p, \rho_0 >$ with the below properties:

- S is the set of all valid states,
- A is the set of all valid actions,

- $R : S \times A \times S \longrightarrow \mathbb{R}$ is the reward function with $r_t = R(s_t, a_t, s_{r+1})$,
- $p : S \times R \times S \times A \longrightarrow [0, 1]$ is the transition probability function with $p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}, \forall s, s' \in S, r \in R$. That means that $p(s', r|s, a)$ is the probability of transitioning to state s' if you start from state s and take action a.
- ρ_0 is the starting state distribution.

The function p defines the *dynamics* of the MDP and along with the reward function R they define the *model* of MDP. This system takes its name from following the *Markov property*, that is, transitions only depend on the most recent state and action rather to prior history. The MDP together with the agent give birth to a sequence or *trajectory* with a form like: $(s_0, a_0, s_1, a_1, ...)$

It should be noted that p characterizes a probability distribution for each tuple (s, a) selection:

$$\sum_{s' \in S} \sum_{r \in R} p(s', r|s, a) = 1, \forall s \in S, a \in A(s) \tag{2.1}$$

From the transition probability function p, there can also be computed:

- State-transition probabilities $p : S \times S \times A \longrightarrow [0, 1]$

$$p(s'|s, a) \doteq Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in R} p(s', r|s, a) \tag{2.2}$$

- Expected reward for a (state, action) pair $r : S \times A \longrightarrow \mathbb{R}$

$$r(s, a) \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r|s, a) \tag{2.3}$$

- Expected reward for a (state, action, next-state) triples $r : S \times A \times S \longrightarrow \mathbb{R}$

$$r(s, a, s') \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in R} r \frac{p(s', r|s, a)}{p(s'|s, a)} \tag{2.4}$$

In general, actions can be any decisions we want to learn how to make, and states can be anything we can know that might be useful in making them. Rewards, also, should be computed inside the physical bodies of natural and artificial learning systems, but are considered external to the agent. Adding on that, the agent-environment boundary represents the limit of the agent's *absolute control* not of its knowledge. This definition is sufficient for this part of the research, while on the chapter 5 the Constrained Markov Decision Process (CMDP) framework will be introduced.

2.3 States and Observations

The set of environmental states is defined as the finite set $S = \{s^1, ..., s^N | \; s \in \mathbb{N}\}$ where $|S| = N$. A **state** is a unique placeholder of all important information in a state of the problem that is modeled. From the one side, there is no information about the world which is hidden (unaccessible) from the state.

From the other side, an **observation** o is a partial description of a state, which may withhold information. It is a common practice in deep RL to represent states and observations by higher order real-valued elements $\in \mathbb{R}^d$ i.e vectors, matrices, tensors. Characteristic example would be the state of a robot represented by a 3-tuple (joint angles, velocities, torques) or an RGB matrix of pixel values representing a visual observation.

When the agent is able to observe the whole state of the environment, we say that the environment is *fully observed*. Furthermore, when the agent can only see a partial observation, we say that the environment is *partially observed*. There is a whole branch of RL researching on Partially Observed Markov Decision Processes (POMDP). To illustrate POMDPs, think most card games where we do not see the opponents' cards; game observations are POMDPs, because the current observation could correspond to different cards in the opponents' hands. They are basically MDPs without the Markov property. The agent keeps an internal belief state b that summarizes its experience up to the current state. In order to update the belief state, a state estimator (SE) is being used, based on the last action, the current observation and the previous belief state.

To be more accurate in the notation, it is clear that sometimes the state symbol s is put in places where the actual symbol should be the symbol of observation o. More precisely in cases where we must describe how the agent decides an action. Commonly it is typed that the action is conditioned on the state, when in reality, the action is conditioned on the observation. This is very easy to spot because the agent has no access to the state. Nevertheless, in this work, I will follow the standard conventions for the rest of the chapters.

2.4 Actions and Action Spaces

The set of actions is defined as the finite set $A = \{a^1, ..., a^K | \; a \in \mathbb{N} \vee a \in \mathbb{R}^d\}$ where $|A| = K$. An action is used to control the system state and the set of actions that can be applied in some particular state $s \in S$, is denoted $A(s) : A(s) \subseteq A$. Every environment has its own different kind of actions and the space of all valid actions is called the **action space**. Some environments like Chess, Go, Backgammon and Atari have *discrete* action spaces $a \in \mathbb{N}$ with finite number of available moves to the agents.

Agents controlling a robot or a satellite in a real-world scenario, have *continuous* action spaces and the actions are real-valued vectors $a \in \mathbb{R}^d$. This work, will focus on continuous actions and action spaces but all the methods and techniques introduced, can be used for discrete problems with the appropriate discretization of their action space.

2.5 Policies

A **policy** is a rule or the agent's plan through which decides what actions to take. There are two kinds of policies namely *deterministic* and *stochastic*. Now, one should keep in mind from now on, that a policy is the agent's brain and by shaping it through optimization methods explored later into this study, we can achieve approximately intelligent agents with amazing problem solving capabilities. In this case, **parametrized policies** are being used, whose outputs are computable functions that depend on a set of parameters i.e neural network weights which are adjustable. It is important to highlight that deterministic policies map each state to one action only in contrast with stochastic policies where the action is a probability distribution function over states where we sample from in order to act. To bridge all this under the same notation, we denote the extra parameter of the parametrized policy by θ.

- Deterministic Policies: μ
$$a_t = \mu_\theta(s_t) \tag{2.5}$$
- Stochastic Policies: π
$$a_t \sim \pi_\theta(\cdot|s_t) \tag{2.6}$$

2.6 Trajectories

A trajectory τ is a sequence of states and actions created by the MDP and the agent,

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{2.7}$$

Denoting the first state of the world as s_0, randomly sampled by the **start-state distribution** ρ_0: $s_0 \sim \rho_0(\cdot)$.

In the agent-environment interface, state transitions stem from the natural laws of the environment and depend on only the most recent action a_t. State transitions describe what happens to the environment between the state at **time** t, s_t and the state at $t+1, s_{t+1}$. They can be deterministic,

$$s_{t+1} = g(s_t, a_t) \tag{2.8}$$

or stochastic,

$$s_{t+1} \sim p(\cdot|s_t, a_t). \tag{2.9}$$

2.7 The Reward Function

A key piece in Reinforcement Learning, the reward function R depends on the current state of the environment, the action just selected and the next state of the environment:

$$r_t = R(s_t, a_t, s_{t+1}),\ r_t \in \mathbb{R} \tag{2.10}$$

or simplified to current state $r_t = R(s_t)$, or state-action pair $r_t = R(s_t, a_t)$.

2.8 The Return

The goal of the agent is to maximize a notion of cumulative reward over a trajectory with three main extensions, all denoted $R(\tau)$.

- Finite-horizon undiscounted return,

$$R(\tau) = \sum_{t=0}^{T} r_t \tag{2.11}$$

  which is the sum of rewards obtained in a fixed window of steps.

- Infinite-horizon discounted return,

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.12}$$

  which is the sum of all rewards obtained in the long-run, but the rewards that are received in feature steps are discounted according to how far away in time they will be received. The *discount factor* $\gamma \in [0, 1)$ maximizes the discount of rewards obtained later than rewards obtained earlier. Adding to that, when $\gamma = 0$ the agent is called *myopic*, caring only for the immediate rewards. Last, this factor ensures that the infinite-horizon sum of the rewards obtained is finite.

- Average-reward return,

$$R(\tau) = \lim_{h \to \infty} \frac{1}{h} \sum_{t=0}^{h} \gamma^t r_t \tag{2.13}$$

  maximizes the long-run average discounted reward and as $\gamma \to 1$, it is equal to the infinite-horizon discounted reward.

Leaving the mathematically convenient expressions in the side for a bit and realize that in practice is common to set up algorithms to optimize the undiscounted return, but in parallel utilize discount factors for estimating value functions.

2.9 The Reinforcement Learning Problem

In this section, we try to formulate the RL problem without taking into account the return measure (whether infinite-horizon discounted, finite-horizon undiscounted or average-reward) and whatever the selection of policy. The goal of RL is to choose a policy which maximizes **expected return** when the agent acts according to it. But in order to talk about expected return, first must formulate probability distributions over trajectories. For now on, we assume that the policy and the environment transitions are stochastic. In this case, the probability of a $T-$step trajectory is:

$$p(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} p(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{2.14}$$

Next, the expected return for a measure, denoted by $J(\pi)$ is:

$$J(\pi) = \int_{\tau} p(\tau|\pi)R(\tau) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)] \tag{2.15}$$

Finally the central optimization problem in RL can be formulated as:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{2.16}$$

with π^* denote the **optimal policy**.

2.10 Solution Methods

There are two main categories of solution methods, namely **tabular** and **approximate**. The first category can be used in relative small state and action spaces where it is possible to store and use the values on a lookup table. These values are updated as the RL algorithm progresses and these methods usually converge to the optimal value functions. Nevertheless, when the state and action spaces are big e.g in the continuous domain, these methods are computationally infeasible because they pose a huge memory bottleneck for storing and updating all the values. On top of that, more time and data are required to fill the tables accurately. Thus, in this work, we focus on approximate solution methods, using neural networks as function approximators. Approximate solution methods assume the existence of similar states and try to *generalize* over big state space by efficient sampling in order to train the NNs. Generalization from examples is well studied in the context of supervised learning, providing us with a lot of methods to use and combine for the RL task at hand. Nevertheless, new issues arise in the RL context such as bootstrapping, nonstationarity and delayed targets. These and other issues will be treated in more detail in the next sections.

2.11 Neural Networks

Neural Networks (NNs) are among the most powerful known function approximators with a great variety of applications that shaped the modern Machine Learning. Neural networks are inspired from the human brain structure and more specific its *plasticity*, i.e the ability to learn from experience, to adapt and to survive in ever-changing environments. The need to represent the electro-chemical processes in great detail in order to represent the biology as close as possible led to *firing rate* models. Here it is useful to include the term *action potential* or *spike* to illustrate that when a spike is generated, the neuron is said to fire. The implicit assumption in firing rate models is that most of the information in neural processing can be found in the mean activity and frequency of spikes of the neurons. A simple mathematical description of the mean neural activity S_i of neuron i, which receives input from a set $\mathbb{J}$ of neurons $j \neq i$, is the below.

$$S_i = h(x_i) \text{ with } x_i = \sum_{j \in \mathbb{J}} w_{ij} S_j \tag{2.17}$$

The parameter $w_{ij} \in \mathbb{R}$ represents the strength of the synapse connecting the neuron $j \in \mathbb{J}$ with neuron i and h is an activation function. Positive $w_{ij} > 0$ increase the *local potential* x_i for an active neuron j $(S_j > 0)$, while negative $w_{ij} < 0$ contribute negative to the weighted sum.

2.11.1 Activation Functions

Various activation functions are considered in the literature and more specific in the context of feed-forward neural networks, the most used one can be found on Table 2.1. A very important class is the sigmoidal functions such that

$$h(x) = \frac{1}{2}\Big(1 + \tanh\big[\delta(x-\theta)\big]\Big) \tag{2.18}$$

where the threshold parameter θ localizes the steepest increase of activity and the gain parameter δ quantifies the slope.

ReLU	$h(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$
Softmax	$h(\mathbf{x})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{j_i}}$ for $i = 1....,N$ and $\mathbf{x} = (x_1, ..., x_K) \in \mathbb{R}^N$
tanh	$h(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$
Softplus	$h(x) = ln(1 + e^x)$
Swish	$h(x) = \frac{x}{1+e^{-x}}$

Table 2.1: Commonly used differentiable and unbounded (except tanh) activation functions.

2.11.2 McCulloch Pitts Neurons

A very popular model because of its simplicity and similarity to boolean concepts (McCulloch & Pitts, 1943). For infinite slope $\delta \to \infty$, the sigmoidal activation become step function and equation 2.18 yields

$$h(x) = \text{sign}(x - \theta) = \begin{cases} +1 & \text{if } x \geq \theta \\ -1 & \text{if } x < \theta \end{cases} \tag{2.19}$$

This is a symmetrized version of a binary activation function where only two states are considered; the model neuron is at rest $S = -1$ or it fires at maximum frequency $S = +1$.

2.11.3 The Perceptron

Pioneered by Frank Rosenblatt (Rosenblatt, 1958), the Perceptron is the building block from which to construct more powerful systems. According to professor Manfred Opper "*The perceptron is the hydrogen atom of neural network research*". The term perceptron refers to a single layer, binary classifier with real-valued inputs $\mathbf{x} \in \mathbb{R}^N$ which are connected to a single binary output unit of the McCulloch Pitts type with activation $S(\mathbf{x}) \in \{-1, +1\}$. The feedback of a perceptron with weight vector $\mathbf{w}$ is obtained by applying a threshold operation to the weighted sum of inputs:

$$S_{\mathbf{w},\delta}(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} - \delta) = \pm 1 \tag{2.20}$$

where δ and $\mathbf{w} \in \mathbb{R}^N$ parametrize the specific input/output relation and can be viewed as the simplest neural network. Now, we define a set P of input/output pairs $\mathbb{D} = \{\xi^\mu, S_T^\mu\}_{\mu=1}^P$ linearly separable, meaning that at least one weight vector $\mathbf{w}$ exists with $S_{\mathbf{w}}^\mu = \text{sing}(\mathbf{w} \cdot \xi^\mu) = S_T^\mu, \forall \mu = 1, 2, ..., P$ and the labels in $\mathbb{D}$ are $S_T = \pm 1$ with T denoting the *training* target. Finally, the weight vector for time steps $t = 1, 2, ..., \tau$ is bound to have the form

$$\mathbf{w}(\tau) = \frac{1}{N} \sum_{\mu=1}^{P} x^\mu(\tau) \xi^\mu S_T^\mu \tag{2.21}$$

and is a linear combination of the vectors $\xi^\mu \in \mathbb{D}$ and the so-called embedding strengths $x^\mu(\tau) \in \mathbb{R}$ quantify their specific contributions. Equation 2.21 constitutes a realization of *Hebbian learning* [1], that is, the change of a component w_j of the weight vector is proportional to the "pre-synaptic" input ξ_j^μ and the "post-synaptic" output S_T^μ.

In order to train the perceptron, we assume a regression problem and to evaluate the performance for one training example we define the simple quadratic deviation objective function J by omitting the subscripts from eq. 2.20:

$$J(S, \hat{S}) = \frac{1}{2}(S - \hat{S})^2 \tag{2.22}$$

that computes how far the output S is from the target $\hat{S}$. The weights initialization is done *tabula rasa*, i.e $\mathbf{w}(0) = 0$. For one training example, we need to compute the partial derivative of J with respect to each weight w_i:

$$\frac{\partial J(S, \hat{S})}{\partial w_i} = -(S - \hat{S}) x_i \tag{2.23}$$

To minimize the error we adjust the weights and update w_i with learning rate α following the below formula

$$w_i \leftarrow w_i - \alpha \frac{\partial J(S, \hat{S})}{\partial w_i} \tag{2.24}$$

[1] From Donald Hebb's book *The Organization of Behaviour.* A theory attempting to explain synaptic plasticity and can be summarize as "cells that fire together wire together".

2.11.4 Multi-layer Perceptron

A limitation of the original Perceptron unit is its inability to solve non linearly separable problems like XOR. To overcome this issue all we have to do is add more units layer-wise to create a Multi-Layer Perceptron (MLP) like in Figure 2.2. The figure suggests a convergent architecture with decreasing number of hidden units per layer towards the output. This network is strictly feedforward; the state of a particular hidden unit depends directly and only on the nodes in the previous layer. All the basic theory discussed for the perceptron is extendable to the multi-layer perceptron.

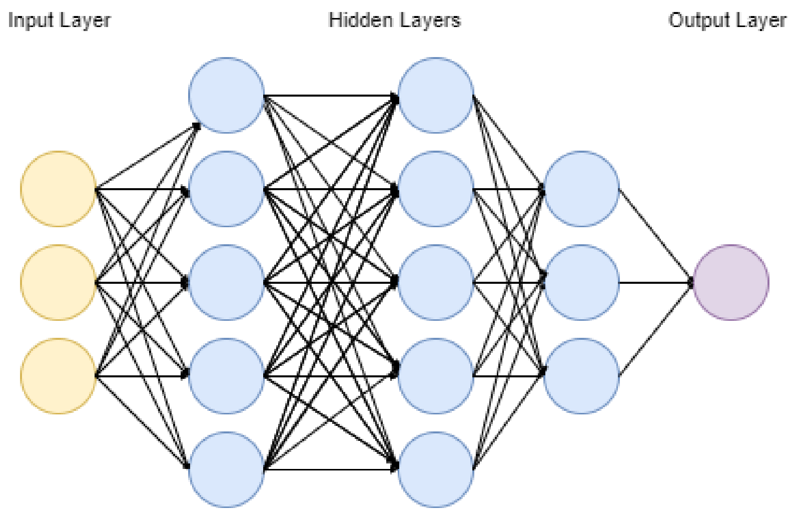


Figure 2.2: An MLP with 3 input units, 3 hidden layers with 5,5 and 3 units respectively and 1 output unit.

The *Universal Approximation Theorem* has various incarnations in the literature but a early and important formulation from (Cybenko, 1989) states that a relative simple network with a single hidden layer of sigmoidal units and a linear output is a universal function approximator. The precise form of the input/output relation is determined by the network architecture and includes its connectivity and the activation functions. In theory whatsoever non-linear function could be an activation function and the most commonly used can be found on Table 2.1. In particular, the rectifier or ReLU is heavily used because of its convenient derivative properties and its simplicity. Generally, we employ the feedforward networks in the context of nonlinear input/output relations with their output interpreted as a real-valued function

$$\sigma : \mathbb{R}^N \to \mathbb{R} \tag{2.25}$$

The goal is to derive the underlying structure of the data by approximation of unknown functions, given an input/output pair. The classification scheme can be realized by additional binary threshold operations on the output σ or by appropriate *binning* when there are multiple classes.

2.11.5 Back-propagation and Stochastic Gradient Descent

Feedforward layer neural networks with differentiable activation functions have the key property of their output being a differentiable function of the inputs. This makes the error measure as defined in equation 2.22 be differentiable with respect to the adaptive weight parameters $\mathbf{w} \in \mathbb{R}^L$ for every layer weight L in the network. The gradient of the error function J is then defined as

$$\nabla_{\mathbf{w}} J(S,\hat{S}) = \left[\frac{\partial J(S,\hat{S})}{\partial w_1}, \frac{\partial J(S,\hat{S})}{\partial w_2}, ..., \frac{\partial J(S,\hat{S})}{\partial w_L}\right]^T \tag{2.26}$$

Back-propagation or backprop (Rumelhart et al., 1986) is an algorithm that computes the chain rule with a specific order of operation that is highly efficient and allows the information from the error to flow backwards through the network to compute the gradient.

The most used optimization algorithm in Machine Learning and especially in Deep Learning is Stochastic Gradient Descent (SGD) (Robbins & Monro, 1951) and its variants. Their aim is to minimize the error function J in order to update the weight parameters. To do that, SGD approximates the error in gradient by computing the loss over a random sampled subset of data. This is possible because we can obtain an unbiased estimate of the gradient by taking the average gradient on a minibatch of m examples drawn i.i.d [2] from the data-generating distribution. An important parameter of SGD is the learning rate that controls the step sizes of the steepest descend in the error landscape. If it is too large, divergent behaviour can be exhibited while a common practice is to decay its value linearly while training. In this research, we mainly use the robust Adam optimizer (Kingma & Ba, 2014) which is derived from “adaptive moments”. The key addition is the one of momentum directly as an estimate of the first-order moment of the gradient, while also includes bias corrections to the estimates of both first and second order moments to account for their initialization.

2.12 Value Functions

The value of a state, or state-action pair, gives the expected return if you start in that state or state-action pair and then act following a particular policy forever after. A value function represents an estimate of how good it is for the agent to be in a certain state. Following the same syllogism, how good is to perform a certain action in that state. The notion of how good is expressed in terms of an *optimality criterion*, i.e in terms of the expected return. In this study we are interested for the **expected infinite-horizon discounted return**, subsequently omitting the time as an argument. This alleviates the extra notation for the terminal states. There are four main families of value functions.

- **On-Policy Value Function** $V^\pi(s)$

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] \tag{2.27}$$

[2] independent and identically distributed

which computes the expected return if the start point is state s and always act according to policy π.

- **On-Policy Action-Value Function** $Q^{\pi}(s,a)$

$$Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi}\left[R(\tau)\,|s_0 = s, a_0 = a\right], \;\; Q : S \times A \rightarrow \mathbb{R} \tag{2.28}$$

which computes the expected return if the start point is state s, takes an arbitrary action a (which may be just noise and not sampled from the policy) and always act according to policy π.

2.13 Optimal Polices & Value Functions

In the context of learning, *optimality* can be characterized in terms of what the final result of learning might be. According to (Wiering & Van Otterlo, 2012) the first concern is about the agent's ability to achieve *optimal performance* in principle. Many algorithms provide proofs of convergence and optimal behaviour, nevertheless many of them do not. Then a question rises, how can we know that reaching a global optimum, a local optimum or even an oscillation between performances is ensured? In this research we will answer this question by performing experiments in various environments taking into consideration the performance-reward trade-off and highlight them with appropriate figures.

The second concern is about the speed of converging to an optimal solution. It is easy to distinguish between learning methods by the number of interactions needed or how much computation is required per interaction. Closely related, what would be the performance after a certain period of time? For example in supervised learning the optimality criterion is commonly defined in terms of *predictive accuracy* which is different from optimality in the MDP setting. Next, it is important to know how much experimentation is necessary, or even the premises of an allowed range, for reaching optimal behaviour. This syllogism, leads to a big dilemma, known as Exploration-Exploitation, described adequately in chapter 2.17. Furthermore, the notion of Exploration will be extended to Safe Exploration, later on chapter 5.2.

The final concern is about the reward that is not obtained by the learning policy as it may be in an optimal policy. This is usually called the *regret* of a policy and it will not concern this study.

An agent that learns an optimal policy has done very well, but in typical practical scenarios, this rarely happens. In the continuous action domain, we are interested in this study, *optimal policies* can be generated only with extreme computational cost. To relax this computational cost, we will inevitably settle for approximations. The challenge here is to achieve useful approximations, that is why we use Artificial Neural Networks (ANN), the best non-linear function approximators. Adding depth to these neural networks (more layers and techniques), we can start talking about Deep Neural Networks (DNN). Having Deep Learning (Goodfellow et al., 2016) as the guideline of how to train these DNN, trying to solve the RL problem now, subsequently leads to **Deep Reinforcement Learning** (DRL).

Here, it is important to highlight (Watkins & Dayan, 1992) work on Q-learning algorithm for estimating Q^*, placed action-value functions into the epicenter of RL, and commonly these functions are called Q-functions.

- **Optimal Value Function** $V^*(s)$

$$V^*(s) = \max_{\pi} \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_0 = s] \tag{2.29}$$

  which computes the expected return if the start point is state s and always act according to the *optimal* policy.

- **Optimal Action-Value Function** $Q^*(s, a)$

$$Q^*(s, a) = \max_{\pi} \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_0 = s, a_0 = a] \tag{2.30}$$

  which computes the expected return if the start point is state s, takes arbitrary action a, and then always act according to the *optimal* policy.

The optimal policy in state s will select whichever action maximizes the expected return from starting in s. This results into obtain optimal action $a^*(s)$.

$$a^*(s) = \arg\max_{a} Q^*(s, a) \tag{2.31}$$

There may be multiple actions that maximize $\mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)]$, in which case, all of them are optimal and the optimal policy can randomly select any of them. Still, there is always an optimal policy that deterministically selects an action.

2.14 Bellman Equations

Value functions fundamentally satisfy certain recursive properties and for any policy π and any state s, the (Bellman, 1957) equations for the on-policy value functions adapted to our formalization are:

$$V^\pi(s) = \mathop{\mathbb{E}}_{a \sim \pi, s' \sim p}[r(s, a) + \gamma V^\pi(s')] \tag{2.32}$$

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{s' \sim p}\Big[r(s, a) + \gamma \mathop{\mathbb{E}}_{a' \sim \pi}[Q^\pi(s', a')]\Big] \tag{2.33}$$

where $s' \sim p \doteq s' \sim p(\cdot|s, a)$ indicating that the next state s' or (s_{t+1}) is sampled from the environment. Also $a \sim \pi \doteq a \sim \pi(\cdot|s)$ indicating the action a is sampled under policy π and $a' \sim \pi \doteq a' \sim \pi(\cdot|s')$ indicating the next action a' is sampled under the policy's π next state s'. For the optimal value functions the Bellman equations are:

$$V^*(s) = \mathop{max}_{a} \mathop{\mathbb{E}}_{s' \sim p}[r(s, a) + \gamma V^*(s')] \tag{2.34}$$

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s'\sim p}\Big[r(s,a) + \gamma \underset{a'}{max}[Q^*(s',a')]\Big] \tag{2.35}$$

A crucial difference between the above equations is the presence of the max over actions, meaning that whenever the agent gets to pick its action, in order to act optimally, it has to pick the action that leads to the highest value.

2.15 Advantage Functions

In RL, it is common to seek a measure of how much better one action is than others on average. For this reason, we need to know the relative **advantage** of that action. For this purpose, the advantage function $A^\pi(s,a)$ of a policy π describes how much better it is to take the specific action a in state s, over randomly selecting an action according to $\pi(\cdot|s)$, having in mind that we act according to policy π afterwards. We will describe in more detail the advantage function later in this study when we will talk about policy gradient algorithms. For now, it is sufficient to present the mathematical definition:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{2.36}$$

2.16 Taxonomy of RL methods

This section will present an informative laconic description of the modern deep RL algorithms taxonomy, with a disclaimer of the authors' perspective.

2.16.1 Model-free

These methods do not build a model of the environment of the reward, rather directly connect observations to actions (or values related to actions). The agent takes current observations, makes some computations on them and then results the action that must be performed next. More specific, a model-free method is one that can be applied both when there is a mathematical model and when there is not. A limiting factor is that the learning signal consists only of scalar rewards, not taking into account the information contained in state transition tuples.

2.16.2 Model-based

These methods try to predict what the next observation and/or reward will be. Based on this prediction, the agent tries to choose the best possible action to take, very often making such predictions multiple times to look more and more steps into the future. An example would be deterministic environments e.g board games with strict rules. When the model is trained, it can be used to choose actions, either by backpropagation of the reward gradients into a policy or by planning directly through the model. In the later case, a very successful method is **model-predictive control** (MPC). In this family of methods, a new action plan is generated at each time step while the first action of the plan is executed before the planning begins from scratch.

2.16.3 Off-policy

In this family belong methods with the ability to learn historical data (obtained by a recorded human demonstration or by a previous version of the agent or just seen by the same agent several episodes ago). These methods requires a kind of memory or buffer to store the computed values. An advantage is that most of the computation is done off-line before the control process starts at time 0.

2.16.4 On-policy

In this family belong methods where most of the computation is performed just after the current state s_t becomes known. Thus, training is done on samples generated by the current policy. These methods are typically faster than off-policy but require much more fresh data from the environment, which is usually costly.

2.17 Exploration-Exploitation Dilemma

The trade-off or dilemma, whether to obtain new knowledge or use the already acquired one to improve the performance, is a natural one. People face this choice all the time e.g should I go to an already known bar for drink or try this new fancy club? Should I study a new field or keep working in my domain of expertise? There are no universal answers to these questions. Subsequently, it is also a main challenge across DRL methods and techniques this balance between *exploitation* of a known proven to be good policy and the *exploration* of the environment to ensure that the current policy is indeed the optimal one. As discussed before on 2.13, *optimality* is difficult and requires relaxations in order to be approximated. Thus, it is a design choice one is called to make when building an end-to-end DRL system. It is apparent from (R. R. Sutton, 2019) that leveraging of computation resources is the only way that will inevitably yield better results, rather than trying to build human-knowledge-based agents. Searching and learning are two other key components that require careful balance for a successful implementation. The *bitter lesson* Richard S. Sutton talked about is based on historical observations and the notion that AI agents should discover like we can, rather contain what we have discovered. Breakthrough progress arrives eventually by a different approach which is based on scaling computation by search and learning.

3 Policy Optimization

In this part, we will present the mathematical foundations considering the case of a stochastic, *parametrized policy* π_θ or $\pi(a|s,\theta) = Pr\{A_t = a|S_t = s, \theta_t = \theta\}$ to denote the probability that action a is selected at time t given that the environment is in state s at time t with policy's vector parameter $\theta \in \mathbb{R}^d$. This family of methods is called Policy Gradient (PG) and in contrast with action-value methods where they learn the values of actions and then select actions based on their estimated action values, these methods can select actions without taking into account a value function. Going one step further, we will investigate methods that learn approximations for both policy and value functions called $Actor - Critic$ methods.

Key ingredient of these methods is the maximization of performance, based on the gradient of a scalar performance measure $J(\theta)$. In our case, we aim to maximize the expected return $J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}} R(\tau)$, so the updates approximate gradient $ascent$ in J:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla_\theta J(\pi_\theta)}|_{\theta_t} \tag{3.1}$$

where $\widehat{\nabla_\theta J(\pi_\theta)}|_{\theta_t} \in \mathbb{R}^d$ is a stochastic estimate, whose expectation approximates the gradient of the performance measure with respect to its argument θ_t. Next, we derive a simple computable expression of the simplest policy gradient.

- *Probability of a trajectory.* Given actions $a \sim \pi_\theta$ the probability of a trajectory $\tau = (s_0, a_0, ..., s_{T+1})$ is:

$$P(\tau|\theta) = \rho_0(s_0)\prod_{t=0}^{T} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t) \tag{3.2}$$

- *Log-derivative trick.* Based on the simple rule of calculus $\frac{\partial logx}{\partial x} = \frac{1}{x}$, we get:

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta)\nabla_\theta \log P(\tau|\theta). \tag{3.3}$$

- *Log-probability of a trajectory.*

$$\log P(\tau|\theta) = \log\rho_0(s_0) + \sum_{t=0}^{T}\bigg(\log P(s_{t+1}|s_t, a_t) + \log\pi_\theta(a_t|s_t)\bigg). \tag{3.4}$$

- *Grad-log-probability of a trajectory.*

$$\begin{aligned}\nabla_\theta \log P(\tau|\theta) &= \cancelto{0}{\nabla_\theta \log\rho_0(s_0)} + \sum_{t=0}^{T}\bigg(\cancelto{0}{\nabla_\theta \log P(s_{t+1}|s_t, a_t)} + \nabla_\theta \log\pi_\theta(a_t|s_t)\bigg)\\ &= \sum_{t=0}^{T}\nabla_\theta \log\pi_\theta(a_t|s_t).\end{aligned} \tag{3.5}$$

Gradient expressions are zero due to no dependencies of the environment on parameter θ.

Putting all expressions together, we can derive the expectation:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right] \tag{3.6}$$

This expectation can be estimated with a *sample mean* by collecting a set of trajectories $\mathcal{D} = \{\{\tau_i\}_{i=1,\ldots,N} \big| |\mathcal{D}| = N\}$ where each trajectory is obtained by having the agent interact with the environment under policy π_θ.

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \tag{3.7}$$

With this expression we can compute the policy gradient and take an update step according to equation 3.1. This step pushes up the log-probabilities of each action in proportion to the sum of all rewards $R(\tau)$, which is controversial from the standpoint that agents should improve their actions on the basis of rewards that come after, rather than before taking the action. Based on that, the policy gradient can also be expressed:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) \tag{3.8}$$

This form is so-called *reward-to-go* policy gradient as the sum of rewards after a point in a trajectory can be expressed as:

$$\hat{R}_t \doteq \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) \tag{3.9}$$

An intermediate result from the theory of policy gradient is the Expected Grad-Log-Prob (EGLP) lemma.

EGLP Lemma. Suppose that P_θ is a parametrized probability distribution over a random variable, x. Then:

$$\mathop{\mathbb{E}}_{x \sim P_\theta} \nabla_\theta \log P_\theta(x) = 0 \tag{3.10}$$

With this at hand, it is easy to add or subtract any number of terms from the expression for the policy gradient equation 3.6, without any change in the expectation, for any function b which depends only on state:

$$\mathop{\mathbb{E}}_{a_t \sim \pi_\theta} \nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t) = 0$$

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t) \right) \tag{3.11}$$

Any function b, even a random variable, as long as it does not vary with a, can be used in this way and is called **baseline**. One natural choice of a baseline is the on-policy value function $V^{\pi}(s_t)$, which empirically has the positive effect of reducing variance in the sample estimate for the policy gradient. Because of that, faster and more stable policy learning is to be expected.

To sum up, we have derived so far the general form of the policy gradient:

$$\nabla_{\theta} J(\pi_{\theta}) = \mathop{\mathbb{E}}_{\tau\sim\pi_{\theta}}\left[\sum_{t=0}^{T} \nabla_{\theta} \log \pi_{\theta}(a_t|s_t)\Phi_t\right], \tag{3.12}$$

where Φ_t could be any weight function we have seen so far like:

$$\Phi_t = R(\tau), \quad \Phi_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}), \quad \Phi_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t).$$

The main difference across these choices is the difference in variances as discussed before. In addition, more choices are applicable as discussed in depth by (Schulman et al., 2018), from which there are two worth mentioning.

- The on-policy Action-Value function. $\Phi_t = Q^{\pi_\theta}(s_t, a_t)$
- The Advantage function. $\Phi_t = A^{\pi_\theta}(s_t, a_t)$

3.1 Algorithms

In this section, some key algorithms that shaped the domain of policy optimization will be described. These algorithms span between on-policy and off-policy methods, providing a more global perspective on the task domain of continuous action control.

3.1.1 Vanilla Policy Gradient

This is the purest version of policy gradients and will not be used in the experiments, rather than used to present the first kind of policy gradient algorithms. The main idea is to drive up the actions' probabilities which lead to higher return, while drive down the actions' probabilities that lead to a lower return until an optimal policy has been reached. Following the already introduced notation, we denote a policy π_θ with parameters θ and expected finite-horizon undiscounted return of the policy $J(\pi_\theta)$. The gradient is

$$\nabla_{\theta} J(\pi_{\theta}) = \mathop{\mathbb{E}}_{\tau\sim\pi_{\theta}}\left[\sum_{t=0}^{T} \nabla_{\theta} \log \pi_{\theta}(a_t|s_t) A^{\pi_\theta}(s_t, a_t)\right] \tag{3.13}$$

with τ trajectory and A^{π_θ} the advantage function of the current policy. Computing advantage function estimates can be done based on the infinite-horizon discounted return or using the finite-horizon undiscounted policy gradient formula. In order for the policy gradient algorithm to update the parameters via stochastic gradient ascent, the below weight update step is performed

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k}) \tag{3.14}$$

To sum up, VPG is an on-policy algorithm that trains a stochastic policy, that is, exploration is performed by sampling actions according to the latest version of the stochastic policy currently in. The proportion of randomness added in action selection depends on hyperparameters and the training process. The policy becomes less random while the train progresses, as the update rule drives the agent to exploit rewards already found, trapping him many times in local optima. The pseudocode of the VPG can be found below.

Algorithm 1: Vanilla Policy Gradient (VPG)

```
1 Input: Initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2 for $k = 0, 1, ...$ do
3     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the
        environment.
4     Compute rewards-to-go $\hat{R}_t$
5     Compute advantage estimates, $\hat{A}_t$ (with any method) based on the current value
        function $V_{\phi_k}$
6     Estimate policy gradient as
          $\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_\tau$
7     Compute policy update using standard gradient ascent
          $\theta_{k+1} = \theta_k + a_k \hat{g}_k$
8     Fit value function by regression on mean-squared error:
          $\phi_{k+1} = \underset{\phi}{\text{argmin}} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2$
        typically through a gradient descent algorithm
9 end
```

3.1.2 Proximal Policy Optimization

This method came from the OpenAI team (Schulman, Wolski, et al., 2017) almost 2 years after TRPO (Schulman, Levine, et al., 2017) and proposed a new simpler objective, rather than the second-order of TRPO. There are two main variants of PPO, one with penaltized KL-divergence and one with clipped objective. In this study we will focus on the second variant. More specific, in this variant there is no KL-divergence term in the objective and doesn't have any kind of constraints. As a substitute it uses clipping in the objective function to remove any support for the new policy to get far from the old policy.

Next, we present a brief and consistent part of the PPO's theory, following the same notation for the policy π_θ with parameters θ and policy updates such that:

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathbb{E}} \left[L(s,a,\theta_k,\theta) \right] \tag{3.15}$$

commonly taking many steps of minibatch SGD to maximize the objective. Here, the objective function is expressed by:

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a),\ \ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right) \tag{3.16}$$

where ϵ is a numerically small hyperparameter which indicates how far away the new policy is allowed to go from the old. Although this is the most analytic expression, it isn't useful for computations rather than the simplified:

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s,a)) \right) \tag{3.17}$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases}$$

It is more convenient to see what is going on if take a look at a single state-action pair (s,a) and take the two cases.

- Positive Advantage: If the advantage of that state-action pair is positive, its contribution to the objective is:

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1+\epsilon) \right) A^{\pi_{\theta_k}}(s,a) \tag{3.18}$$

  Due to the positive advantage, the objective will increase if the action becomes more probable that is, if $\pi_\theta(a,s)$ increases. To put a limit on the objective's increase bounds, the min in this term is used. When the policy $\pi_\theta(a|s) > (1+\epsilon)\pi_{\theta_k}(a|s)$, the minimum is taken and this term approaches the min value of $(1+\epsilon)A^{\pi_{\theta_k}}(s,a)$. Then the new policy does not benefit by moving too far away from the old policy.

- Negative Advantage: If the advantage of that state-action pair is negative, its contribution to the objective is:

$$L(s,a,\theta_k,\theta) = \max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon) \right) A^{\pi_{\theta_k}}(s,a) \tag{3.19}$$

Due to the negative advantage, the objective will increase if the action becomes less probable that is, if $\pi_\theta(a, s)$ decreases. To put a limit on the objective's increase bounds, the max in this term is used. When the policy $\pi_\theta(a|s) < (1-\epsilon)\pi_{\theta_k}(a|s)$, the maximum is taken and this term approaches the max value of $(1-\epsilon)A^{\pi_{\theta_k}}(s, a)$. Then the new policy does not benefit by moving too far away from the old policy.

To sum up, PPO is an on-policy methods that trains a stochastic policy and explores actions according to the latest version of the policy. The ratio of random noise to the action selection depends on hyperparameters and the training process setup. The pseudocode for PPO can be found below.

Algorithm 2: Proximal Policy Optimization (PPO)

```
1 Input: Initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2 for $k = 0, 1, ...$ do
3     Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the
        environment.
4     Compute rewards-to-go $\hat{R}_t$
5     Compute advantage estimates, $\hat{A}_t$ (with any method) based on the current value
        function $V_{\phi_k}$
6     Update the policy by maximizing the PPO-Clip objective:
          $$\theta_{k+1} = \underset{\theta}{\text{argmax}} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}}(a_t|s_t) A^{\pi_{\theta_k}}(s_t|a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t|a_t)) \right)^2$$
7     Fit value function by regression on mean-squared error:
          $$\phi_{k+1} = \underset{\phi}{\text{argmin}} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2$$
        typically through a gradient descent algorithm.
8 end
```

3.1.3 Actor-Critic

This is a family of methods also known as advantage actor-critic (A2C) and utilizes two approximators in order to learn a value function and a policy. An Actor-Critic system comprises of two components according to (Konda & Tsitsiklis, 2000), namely the **actor** and the **critic**. An illustration can be found on on Figure 3.1. The actor is represented by a policy network which returns a probability distribution of actions to select from. The critic provides us with feedback of how good the actions were. It is a common technique to bundle both of the actor and the critic to share a common network body where they can share low-level features.

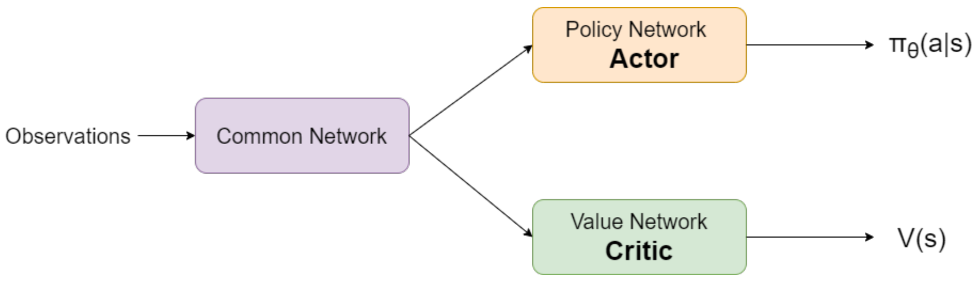


Figure 3.1: The A2C architecture with a shared network body

3.1.3.1 Off-Policy Actor-Critic

The Actor-Critic framework can also be modeled in the off-policy setting (Degris et al., 2013). From the one side, a new behaviour policy $b(a, s)$ is used to generate trajectories in contrast with the target policy $\pi_\theta(a|s)$. From the other side, the critic evaluates the state value function using the stored experience from the replay buffer. Finally, the policy's parameters θ are updated using stochastic gradient ascent and to account for the mismatch between the behavioural policy and the target policy $Target = \frac{\pi_\theta(a|s)}{b_\theta(a|s)}$, importance sampling is used (Precup, 2000). A practical implementation of this family of algorithms, is presented in the next section.

3.1.4 Deep Deterministic Policy Gradient

This method is based on the article by (Silver et al., 2014) who proved that the stochastic policy gradient is equivalent to the deterministic policy gradient. Working on the same direction, (Lillicrap et al., 2019) went a step further, improving the deterministic policy gradient by including two main improvements to stabilize training and speed up convergence.

This method has also the great property of being off-policy and belong to the Actor-Critic family. First, from actor we want the the action to take for every given state. In the continuous action domain, every action is a number, consequently the actor network will have the state as input and return N values, one for every action. This mapping is deterministic, because the same network always returns the same output if the input is the same.

Second, the role of critic is to estimate the Q-value, which is a discounted reward of the action taken in some state. As the policy is deterministic, the gradients from Q can be calculated, which is accessible from the critic network. This network uses actions produced by the actor, setting the whole system differentiable. Because of that, it is optimizable end to end with stochastic gradient descent. In the end, the critic network is updated using the Bellman equation to find the approximation of $Q(s, a)$ and minimize the mean squared error objective. An illustration of the complete architecture can be found on Figure 3.2.

Next, we present a brief and consistent part of the DDPG's theory, recalling the Bellman equation 2.35 for the optimal action-value function $Q^*(s, a)$. This Bellman equation is the starting point for learning an approximator to $Q^*(s, a)$. The approximator is a neural network

$Q_\phi(s,a)$ with parameters ϕ. We collect a set $\mathcal{D} = (s, a, r, s', d_{term})$ of transitions where $d_{term} \in [0,1]$ is a binary indicator whether state s' is terminal. Then, to measure how close is Q_ϕ to satisfy the Bellman equation, we use the **mean-squared Bellman error** (MSBE):

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathbb{E}} \left[\left(Q_\phi(s,a) - \Big(r + \gamma(1-d) \max_{a'} Q_\phi(s',a') \Big) \right)^2 \right] \tag{3.20}$$

Evaluating $(1-d)$ when s' is terminal state results the Q-function showing that the agent does not get any additional rewards after the current state. The core Q-learning algorithms for function approximators are mainly focus on minimizing this MSBE loss function. DDPG comes with two extensions or tricks that improved the training dynamics and the convergence of the algorithm.

- **Replay Buffer.** First introduced by (Lin, 1992), a set $\mathcal{D}$ of previous experiences with adequate big size to contain many of them. Because of this buffer, we are able to train off-policy this algorithm and have less correlation between the train samples. Still, there are loopholes that must be avoided, like the size and the amount of "fresh" samples collected. A big replay buffer can slow down the learning while only "fresh" samples can lead to overfitting the model. Furthermore, with old and new experiences populated that may even come from an old policy, the Bellman equation does not considers which transition tuples are used, or the way actions were selected because the optimal Q-functions must satisfy the Bellman equation $\forall$ possible transitions.

- **Target Networks.** Where **Target** = $r + \gamma(1-d) \max_{a'} Q_\phi(s',a')$ and is the value we want to approach with the Q-functions by minimizing the MSBE loss. But this target has a dependance on the parameters ϕ we train, making the MSBE minimization uncontrollable. To solve this issue, we use a set of parameters which are close to ϕ but with some time delay meaning that a second network, the target network wanes the first. A major difference with the DQN-based algorithms and DDPG implementation is the updated of target network, once per main network update rather just copying the target network from the main network every fixed number of steps. This is done by Polyak Averaging $\theta_t = \frac{1}{t}\sum_i \theta_i$, for t iterations and parameters $(\theta_1, ..., \theta_t)$, that is $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$ with $\rho \in (0,1)$ close to 1 typically.

A challenging part on continuous action spaces it the computation of the maximum over action. To overcome this, DDPG uses a **target policy network** to compute an action which approximately maximizes $Q_{\phi_{targ}}$. The target policy network is calculated the same way as the target Q-function by Polyak Averaging the policy parameters over the training. Having said all that, DDPG's Q-learning is performed by minimizing the below MSBE loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathbb{E}} \left[\left(Q_\phi(s,a) - \big(r + \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s')) \big) \right)^2 \right] \tag{3.21}$$

where $\mu_{\theta_{\text{targ}}}$ denotes the target policy.

The last step of this method concerns the policy learning side and is quite simple. The goal is to learn a deterministic policy $\mu_\theta(s)$ which selects the action that maximizes $Q_\phi(s,a)$. We assume that the Q-function is differentiable with respect to action and it is straightforward to solve by gradient ascent:

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathbb{E}}\left[Q_\phi(s,\mu_\theta(s))\right] \tag{3.22}$$

with respect to policy parameters only and treat the Q-function parameters as constants.

To sum up, DDPG trains a deterministic policy in an off-policy manner while the original paper uses time-correlated Ornstein-Uhlenbeck noise to explore further the action space. Nevertheless, later results indicate the use of this kind of exploration not very helpful and complicated. The alternative is simple, add uncorrelated Gaussian noise $\mathcal{N}(0,\sigma^2)$, proved to work best. To obtain higher quality training samples, we reduce the scale of noise by decreasing $\sigma^2 = 1 \rightarrow 0.01$ over the course of training. The pseudocode for the DDPG can be found below.

Algorithm 3: Deep Deterministic Policy Gradient (DDPG)

```
1  Input: Initial policy parameters $\theta_0$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2  Set target parameters equal to main parameters $\theta_{targ} \leftarrow \theta, \phi_{targ} \leftarrow \phi$
3  repeat
4      Observe state $s$ and select action
       $a = \text{clip}(\mu_\theta(s) + \epsilon,\ a_{low},\ a_{high})$, where $\epsilon \sim \mathcal{N}(0, \mathcal{I})$
5      Execute $a$ in the environment
6      Observe next state $s'$, reward $r$, and done signal $d$ to indicate if $s'$ is terminal
7      Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8      If $s'$ is terminal, reset environment state
9      if time to update then
10         for however many updates do
11             Randomly sample a batch of transitions, $\mathcal{B} = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12             Compute targets
                   $y(r, s', d) = r + \gamma(1-d)Q_{\phi_{targ}}(s', \mu_{\theta_{targ}}(s'))$
13             Update Q-function by one step of gradient descent using
14                 $\nabla_\phi \frac{1}{|\mathcal{B}|} \sum_{(s,a,r,s',d)\in\mathcal{B}} \left(Q_\phi(s,a) - y(r,s',d)\right)^2$
15             Update policy by one step of gradient ascent using
                   $\nabla_\theta \frac{1}{|\mathcal{B}|} \sum_{s\in\mathcal{B}} Q_\phi(s, \mu_\theta(s))$
16             Update target networks with
                   $\phi_{targ} \leftarrow \rho\phi_{targ} + (1-\rho)\phi$
                   $\theta_{targ} \leftarrow \rho\theta_{targ} + (1-\rho)\theta$
17         end
18     end
19 until convergence
```

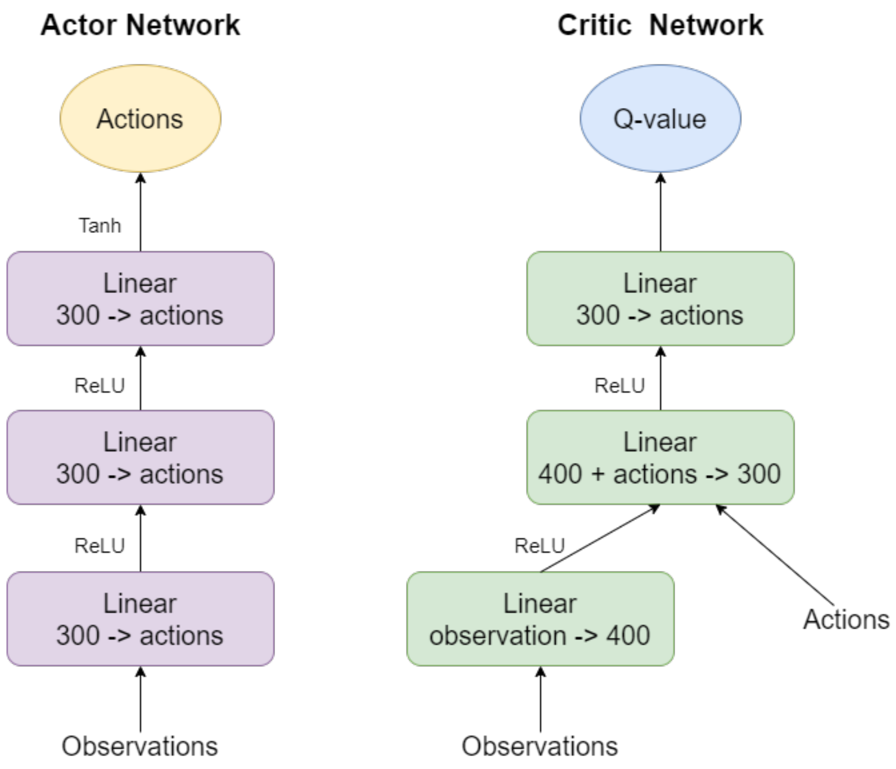


Figure 3.2: The DDPG network architecture

3.1.5 Twin Delayed Deep Deterministic Policy Gradient

A new method introduced by (Fujimoto et al., 2018) for short **TD3** that tries to mitigate the overestimation bias in Q-learning. Overestimation bias is a natural consequence of RL algorithms that try to approximate maximum expected values by maximizing uncertain estimates. This is of particular interest in the actor-critic methods described so far because of the temporal difference update. More specific, in the function approximation setting when the Bellman equation is never exactly satisfied, each update ends with some amount of residual TD-error. So far, we have seen that the policy is updated with respect to the value estimates of an approximate critic, making it susceptible to exploit the critic's over/under estimates. To address all the above issues, three new extensions introduced.

- **Target policy smoothing.** Deterministic policies tend to be susceptible to inaccuracies induced by function approximation error that increases the variance of the target. To combat this side effect, a regularization strategy is needed that pushes similar actions to have similar values, named target policy smoothing. The recipe is simple; adding clipped noised on each dimension of the action and then clip the target action to lie in the valid action range $a_{low} \leq a \leq a_{high}$ with $a_{low} = -1, a_{high} = 1$ typically. The target actions are:

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{low}, a_{high}\right), \quad \epsilon \sim \mathcal{N}(0, \sigma) \tag{3.23}$$

- **Clipped Double Q-Learning.** Originally introduced back in 2010 (Hasselt, 2010), this method augmented with clipped terms found to avoid introducing any additional overestimation over using the standard Q-learning target. It is interesting that it may introduce underestimation bias which is preferable to overestimation bias as the value of underestimated actions will not be explicitly proliferated along the policy update. In practice, TD3 learns two Q-functions and uses the minimum between the two Q-values to assign the targets in the Bellman error loss functions such that the target value is:

$$y(r, s', d) = r + \gamma(1-d) \min_{i=1,2} Q_{\phi_{i,\text{targ}}}(s', a'(s')) \tag{3.24}$$

  and then learn both of them by regressing on the targets:

$$L(\phi_i, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_{\phi_1}(s,a) - y(r,s',d) \right)^2 \right], \quad i = 1,2 \tag{3.25}$$

- **Delayed policy updates.** The authors found that updating the policy and the target networks less frequently, allowing the Q-value error from the critic to be as small as possible before the network update, improved the performance. Thus, the actor and target networks update their weights every two critic updates.

To train the actor, TD3 follows DDPG's training and maximizes

$$\max_{\theta} \mathop{\mathbb{E}}_{s\sim\mathcal{D}} \left[Q_{\phi_1}(s, \mu_\theta(s)) \right] \tag{3.26}$$

To facilitate better exploration of the action space, TD3 simply adds Gaussian noise to the actions at training time with decreasing variance to near 0 towards the end of training. To conclude the presentation of TD3, it is important to highlight the differences to DDPG architecture. In addition to the second Q-value network, smaller actor-critic networks are used with layer size 256. An illustration can be found below on Figure 3.3.

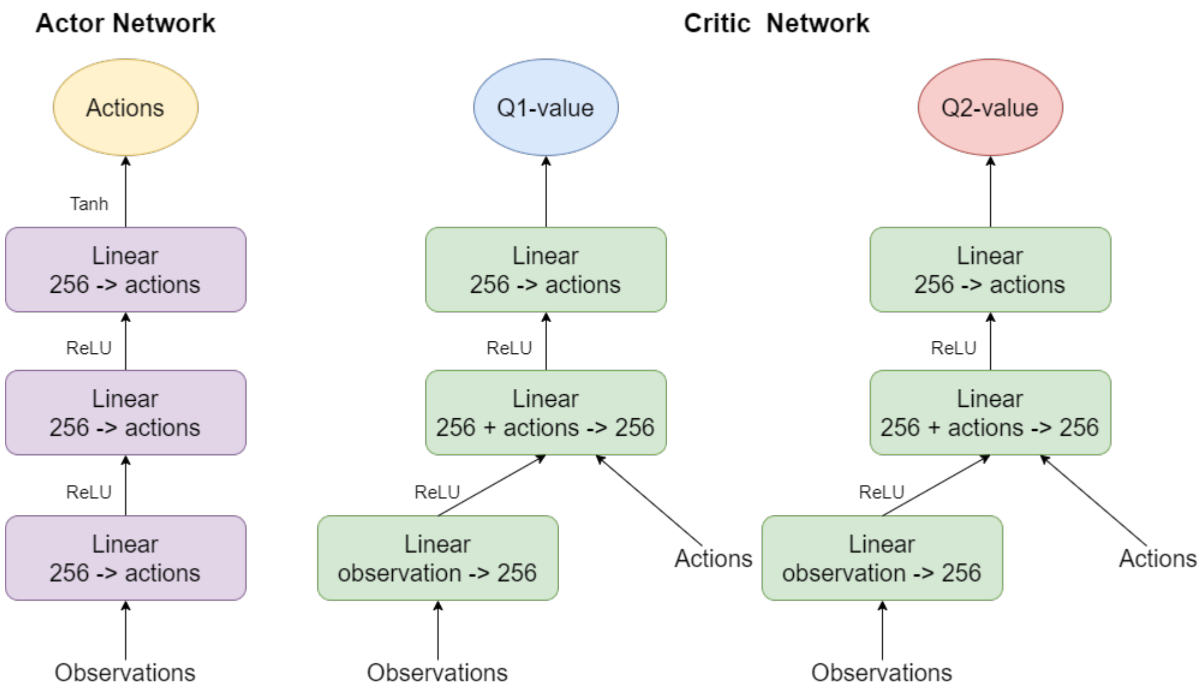


Figure 3.3: The TD3 network architecture

3.1.6 Conclusion

In this chapter, we presented the theoretical background of policy optimization in Deep Reinforcement Learning. First, we introduced the basic concepts and tools to solve the RL problem and then we dove deeper into exploring various modern algorithms that shaped the field. Starting from vanilla policy gradient methods, we explored both on-policy and off-policy methods. More focus was placed on off-policy actor-critic methods. Many problems that arise on policy optimization were addressed and in place solutions to them were described adequately.

In the next chapter we focus on a new off-policy actor-critic algorithm named Sample Policy Gradient or SPG for short.

4 Sample Policy Gradient

A relative new off-policy actor-critic learning algorithm introduced in a paper by (Wiehe et al., 2018). The main motivation behind SPG is to explore the action space more globally and avoid local optima that deterministic algorithms like DDPG fall into. In contrast with DDPG that calculates weight changes of the actor deterministically using backpropagation, SPG samples actions around the current policy $\mu_\theta(s)$ for a specific state. In this study we will use offline Gaussian exploration that is proven to perform very well in continuous action spaces and the racing domain from (Holubar & Wiering, 2020). Although in this research the interest lies on the broader continuous action space domain including locomotion and other tasks presented in the next section in detail.

SPG needs to compute the backpropagation target for the actor. In order to do so, an evaluation must occur between the action of a transition $Q(s_t, a_t)$ and the action predicted by the actor under the current policy $Q(s_t, \mu_\theta(s_t))$, where $\mu_\theta(s_t)$ is the actor's prediction for the state s_t:

$$If\ Q(s_t, a_t) > Q(s_t, \mu_\theta(s_t)) : Target(s_t) = a_t \tag{4.1}$$

The SPG can extend to SPG-OffGE by creating a new sampled action; predefined number of times M by applying Gaussian noise to the best action found so far. The variance of the Gaussian noise σ^2 is decreasing over the course of training from $1 \rightarrow 0.01$. The pseudocode for this version that will be used for this part of the study is given below on Algorithm 4.

Algorithm 4: Offline Gaussian Exploration

```
1  Input: Batch of $N$ tuples $(s_t, a_t, r_t, s_{t+1}, \sigma^2)$
2  for $(s_t, a_t)$ of transition $i$ in input do
3      best ← $\mu_\theta(s_t)$
4      if $Q(s_t, a_t) > Q(s_t, \mu_\theta(s_t))$ then
5          best ← $a_t$
6      end
7      for count in $1, ..., M$ do
8          sampled ← best $+\mathcal{N}(0, \sigma^2)$
9          if $Q(s_t,$ sampled$) > Q(s_t,$ best$)$ then
10             best ← sampled
11         end
12     end
13     if $Q(s_t,$ best$)$ then
14         Target$_i$ $= best$
15     end
16 end
17 Output: Target actions represent the best actions so far denoted $a^*$
```

The critic's architecture figure 3.2 and the training part is exactly the same with DDPG, although the original SPG paper used DPG's critic. It is natural to upgrade to a new and more stable architecture. A small addition is the parameter $\lambda \in (0, 1)$ that controls the amount of noise added to action selection. In the same theme, SPG's critic Q-learning is performed by minimizing the Mean Squared Bellman Error (MSBE) loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathbb{E}} \left[\left(Q_\phi(s,a) - \big(r + \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))\big) \right)^2 \right] \tag{4.2}$$

The actor's architecture is the same but the training part is different. The main difference is that we use the mean squared error objective between actions a currently sampled under μ_θ and best actions a^* found from offline Gaussian exploration. We seek to minimize this objective function, subsequently solve:

$$\min_\theta \underset{s\sim\mathcal{D}}{\mathbb{E}} \left[\left(a(s) - a^*(s) \right)^2 \right] \tag{4.3}$$

By minimizing this objective, we push the actor to select actions closer to the best actions found by exploration for every batch of trajectories. This heavily depends on an accurate critic in order to have as good as possible Q-value estimations when searching for the best actions. The pseudocode for the training of SPG can be found on algorithm 5.

An addition to baseline SPG-OffGE is the prioritized version which uses a **Priority Replay Buffer**, conceptualized by (Schaul et al., 2015) that tries to improve the efficiency of samples in the replay buffer by prioritizing those samples according to the training loss. The authors claim, "training more on data that surprises you" improves convergence and the policy quality. To be more specific, the priority of every sample in the buffer is calculated as:

$$P(i) = \frac{p_i^{\text{a}}}{\sum_k p_k^{\text{a}}} \tag{4.4}$$

with p_i indicating the priority of the $i-$th sample in the buffer. The exponent hyperparameter a $\in [0, 1]$ controls the emphasis given to the priority and is set to 0.6 as proposed by the paper. Larger a values emphasize more on samples with higher priorities. For a $= 0$, the sampling is uniform. By adjusting the priorities for the samples, bias is introduced into the data distribution because some transitions are sampled more frequently than others. This creates the need to compensate in order for stochastic gradient descent to work effectively. The solution is to use sample weights for each batch of transitions w_i, multiplied by the individual batch's sample loss before backpropagating it. The weight value for each sample is:

$$w_i = (N \cdot P(i))^{-\beta} \tag{4.5}$$

where the hyperparameter $\beta \in [0, 1]$ should progressively increase from $0 \to 1$ over the course of training. For $\beta = 1$ the bias introduced by the prioritize sampling is fully compensated.

To sum up this part, we presented SPG-OffGE upgraded with DDPG's architecture along with the improvement of prioritized replay buffer. In the next chapter, Safe Reinforcement Learning is introduced.

Algorithm 5: Sampled Policy Gradient with offline Gaussian Exploration (SPG-OffGE)

```
1  Input: Initial policy parameters $\theta_0$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2  Set target parameters equal to main parameters $\theta_{targ} \leftarrow \theta, \phi_{targ} \leftarrow \phi$
3  **repeat**
4      Observe state $s$ and select action
         $a = \text{clip}(\mu_\theta(s) + \lambda * \epsilon,\ a_{low},\ a_{high})$, where $\epsilon \sim \mathcal{N}(0, \mathcal{I})$
5      Execute $a$ in the environment
6      Observe next state $s'$, reward $r$, and done signal $d$ to indicate if $s'$ is terminal
7      Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8      If $s'$ is terminal, reset environment state
9      **if** *time to update* **then**
10         Decrease $\sigma^2$
11         **for** *however many updates* **do**
12             Randomly sample a batch of transitions, $\mathcal{B} = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
13             Run Offline Gaussian Exploration on $\mathcal{B}$, algorithm 4 to find the best
                 actions a*
14             Compute targets
15                 $$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{targ}}(s', \mu_{\theta_{targ}}(s'))$$
16             Update Q-function by one step of gradient descent using
                   $$\nabla_\phi \frac{1}{|\mathcal{B}|} \sum_{(s,a,r,s',d)\in\mathcal{B}} \Big(Q_\phi(s, a) - y(r, s', d)\Big)^2$$
17             Update policy by one step of gradient descent using
                   $$\nabla_\theta \frac{1}{|\mathcal{B}|} \sum_{s\in\mathcal{B}} \Big(a_{\mu_\theta}(s) - a^*(s)\Big)^2$$
18             Update target networks with
                   $$\phi_{targ} \leftarrow \rho\phi_{targ} + (1 - \rho)\phi$$
                   $$\theta_{targ} \leftarrow \rho\theta_{targ} + (1 - \rho)\theta$$
19         **end**
20     **end**
21 **until** *convergence*
```

5 Safe Reinforcement Learning

5.1 Introduction

Reinforcement Learning and more specifically Deep RL as described in the previous chapter can approximately solve a great variety of modern computational continuous control problems. Because of this success on artificial domains and simulated environments an important question rises. Can this framework be extended to real-world scenarios? The answer is rather satisfying and poses a real up-to-date challenge (Dulac-Arnold et al., 2019). The first most notable challenge concerns the large action space continuous problems come with. Cases like this have already been considered, for example, the Humanoid task where the agent must use many sensors and control many actuators to complete the task successfully. The solution given so far is the offline Gaussian exploration with non-linear function approximation to find the best actions. There are many other alternatives like AE-DQN (Zahavy et al., 2019) where they solve MDPs by eliminating actions while performing Q-learning.

The second most notable challenge concerns **safety** which is usually overlooked. Commonly, RL applications focus on the behaviour of the agent, driven solely by the reward signal. This classic reward side approach is not compatible with real-world applications and cannot ensure nominal performance. The examples are abundant starting from autonomous robots/rovers where the goal is fixed but while on the exploration process, fatal situations must be avoided i.e obstacles and cliffs. In the aerospace domain, an RL controller that is responsible for an Unmanned Air Vehicle (UAV) is constrained by the vehicle's flight envelope that requires a careful avoidance of high-risk situations. Adding on the aerospace domain, a satellite constellation operating on Low or Geostationary Earth Orbit must avoid collisions both with space debris and other satellites. This is a difficult task requiring safe navigation when modeled under the Multi-Agent framework. An autonomous car vehicle should drive without causing physical harm to humans. This example has a dual reading regarding safety; giving birth to the question of whether the life of passengers or the humans' in the scene must be preserved in an accident scenario. This kind of scenarios are evaluated and studied under the field of Ethics in AI and we will not extend the notion of safety in that direction. In the autonomous vehicle paradigm, our study will focus solely on navigating safely to the goal and omitting the Ethics extension. Adding on all that, power infrastructure controllers should not damage critical systems while balancing between production and demand. Another major area that safety plays an important role, is the financial domain and more specific the trading activities RL agents commence like the work of (Tsantekidis et al., 2021). In this line of business, trading with risk can be interpreted as trading safe to accumulate the maximum profit over a time horizon. Last, a very recent article from (Bastani et al., 2021) that implemented an RL system to assist the border control policies during the COVID-19 pandemic in Greece, does not take into account safety considerations during training and gives birth to questions of whether this kind of system can be effective if we consider the aftermath of its implementation. The bottom line is that safety must be ensured to expand the RL framework to real-world applications.

Next, safety is considered to account for both the agent itself and its environment as a general convention. Nevertheless, in this study, we will focus on predefined constraints on the environment only. To do so, we introduce the notion of **safe exploration** problem which stems from the trial-and-error nature of RL where the agents experience dangerous or harmful behaviours during learning (Hans et al., 2008) (Moldovan & Abbeel, 2012) (Pecka & Svoboda, 2014) (García et al., 2015) (Amodei et al., 2016). This behaviour is acceptable for simulated environments but for real-world environments, it is not. To formalize this contrast, we must formulate safety specifications and try to incorporate them into the RL framework. These specifications must be independent of the task performance specifications and a convenient encoding is through **constraints**. This version of RL namely constrained RL, is scalable to the extend of high-dimensional function approximation.

5.2 Safe Exploration

A great amount of work has been conducted into safe exploration, still, there is no unified definition of safety that would satisfy all cases. An early definition given by (Hans et al., 2008) requires humans to label states of the environment as safe and unsafe and the agents are safe when they never enter into unsafe states. The algorithm used for this setup required prior knowledge of a well-defined safety function acting as a decision-maker over possible actions. A different approach connects safety with value alignment between autonomous systems and humans, where the systems' actions contribute to the value maximization for the human through inverse reinforcement learning (Hadfield-Menell et al., 2016). On the same theme, (Christiano et al., 2017) explored agent safety under non-expert human preferences where the goal is to learn a reward function from human feedback and then try to optimize that reward function via binary or ranked preferences. The iterated amplification and distillation training strategy proposed by (Christiano et al., 2018) seeks to combine easier sub-problem solutions and build a training signal for more difficult problems. All these approaches have a common goal to avoid defining explicit safety specifications because they can fail in many ways and try to solve the alignment problem by human-based guidance in order to construct a proper objective or reward function to train the agents (Leike et al., 2018). Nevertheless, a bad reward specification that seems correct at first, may lead to incorrect and unsafe behaviour.

Another line of work (Moldovan & Abbeel, 2012) characterizes the agent safe when it satisfies the ergodicity requirement. That means the state currently be, can be visited from any other state. Subsequently, an agent is safe if never enters into a state where it cannot return from; every mistake is reversible (Eysenbach et al., 2017). This kind of method provides good safety results for some practical cases mostly found on robotics but does not account for more general cases as debated by (Pecka & Svoboda, 2014).

5.3 Constrained RL

In this section we formulate the general problem of training an RL agent with constraints aiming at constraints satisfaction throughout both training exploration and test evaluation.

5.3.1 Constrained Markov Decision Process Framework

A constrained Markov decision process (CMDP) is an MDP essentially, with the constraints addition that restrict the set of policies that allowed for the MDP to solve and are very meticulously studied by (Altman, 1999). More specific, the MDP is extended by adding a set C of auxiliary cost functions, $C_1, ..., C_m$ where each cost function $C_i : S \times A \times S \to \mathbb{R}$ is mapping transition tuples to costs and limits or thresholds $d_1, ..., d_m$ typically human-selected hyperparameters . We denote $J_{C_i}(\pi)$ the expected discounted return of policy π with respect to cost function such that $C_i : J_{C_i}(\pi) = \underset{\tau\sim\pi}{\mathbb{E}}\left[\sum_{t=0}^{\infty} \gamma^t C_i(s_t, a_t, s_{t+1})\right]$ and the set of feasible stationary policies is:

$$\prod_C \doteq \left\{ \pi \in \prod : \forall i, J_{C_i}(\pi) \leq d_i, \;\; i = 1, ..., m \right\} \tag{5.1}$$

This framework can be extended even more and include all kind of cost-based constraints.

5.3.2 Constrained Optimization Problem

The problem we seek to solve is formulated as:

$$\begin{aligned} &\max_{\pi_\theta} \underset{\tau\sim\pi_\theta}{\mathbb{E}}\left[\sum_{t=0}^{T} r_t\right] \\ &\text{s.t} \underset{\tau\sim\pi_\theta}{\mathbb{E}}\left[\sum_{t=0}^{T} C_t\right] \leq d \end{aligned} \tag{5.2}$$

where C_t is the accumulated indicator cost function for the environment where $C_t = 1$ denotes an unsafe situation and d is a hyperparameter.

5.4 Methods

In this section, we provide the description of two methods developed to solve the constrained optimization problem, mainly utilizing NNs. Through my research, I explore various methods which were either very complicated or lacked open source implementation to evaluate their results. These reasons led me to first focus on one of the most cited methods in literature. The second novel method was developed after research and guidance from dr. Marco Wiering.

5.4.1 Safety Layer

The main motivation behind the Safety Layer (Dalal et al., 2018) is to correct dangerous actions to safe ones before the agent performs them. A schematic overview of the architecture, adopted from the paper's authors can be found on figure 5.1.

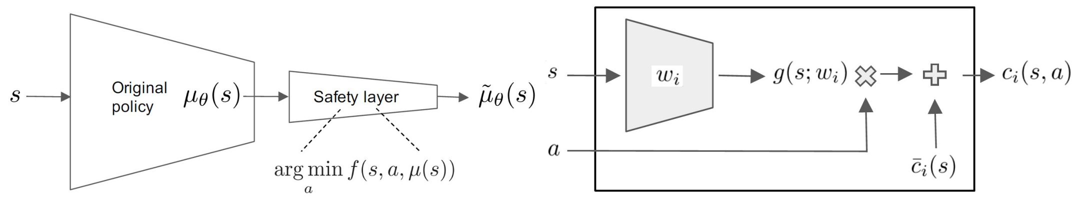


Figure 5.1: Safety Layer architecture.

Starting from the left side, the inputs are the state s, the actions selected by the NN $\mu_\theta(s)$ and the perturbed action α. The output is the new safe action to be performed $\tilde{\mu_\theta}(s)$. If an agent comes close to a state s' with a cost function $c(s')$ that violates a threshold d such that $c(s') \leq d$ this action is marked as dangerous. In order to predict the value of $c(s')$ without having to visit s', an approximation of the cost function is used.

The right side of figure 5.1 explains how the cost function is being approximated. A Neural Network is employed with goal to associate at each state's s' input an approximated cost function $\bar{c}'(s)$ as output. More specific, a linearization is performed:

$$\bar{c}_i(s') \triangleq c_i(s,\alpha) = \bar{c}_i(s) + g(s;w_i)^T\alpha \tag{5.3}$$

with w denoting the weights of a NN and the $g(s;w)$ function, representing how much the cost function is sensitive to variations caused by actions, at state s.

The constrained optimization problem Safety Layer seeks to solve is:

$$\begin{aligned} \alpha^* &= \underset{a}{\text{argmin}}\frac{1}{2}\|\alpha - \mu_\theta(s)\|^2 \\ \text{s.t}\quad & \bar{c}_i(s) + g(s;w_i)^T\alpha \leq d_i, \forall i \in [K] \end{aligned} \tag{5.4}$$

where the subscript i denotes the constraint number. In order to solve the above problem, the authors assumed that there is only a single constraint active at a time in order to reach a closed-form analytical solution. That is:

$$\alpha^* = \mu_\theta(s) - \lambda^*_{i^*}g(s;w_{i^*}) \tag{5.5}$$

$$where\ \ \lambda^*_i = \left[\frac{\bar{c}_i(s) + g(s;w_i)^T\mu_\theta(s) - d_i}{g(s;w_i)^T g(s;w_i)}\right] \ and\ \ i = \underset{i}{\text{argmax}}\lambda^*_i \tag{5.6}$$

The λ^*_i is the optimal Lagrange multiplier, associated with the i-th constraint. If the cost function at the nominator of equation 5.6 is above the threshold d_i means that the action selected from the agent $a = \mu_\theta(s)$ results a dangerous state. This solution is a linear projection of the original action $\mu_\theta(s)$ to the *safe* hyperplane with slope $g(s; w_i^*)$ and intercept $\bar{c}_{i^*}(s) - d_{i^*}$. Furthermore, the implementation felt rational, consisting of vector products and a "max" operation on tensor objects. The proof of the validity of the mathematical part of this method can be found on the paper and is omitted. This concludes the presentation of the Safety Layer method. The results from our experimentation with Safety Layer can be found on the next section 7.4.1.

5.4.2 Safe RL through Bounded Transfer Learning

In this part, we present a new novel method, trying to solve the constrained optimization problem with function approximation and transfer learning. Inspiration for our work came from the paper of (Ray et al., 2019) and the article of (Gros et al., 2020). The research material from Ray et al. provided us with the environment to test and evaluate our algorithms, while the work from Gros et al. provided the mathematical foundations to test our hypothesis that is: Can we approximate a bounded area on the Q-value of a state action pair, that is safe?

Our research at this point have gone through the safety layer's results and decided not to invest time on 2-nd order Lagrangian methods and on-line algorithms like PPO and CPO (Achiam et al., 2017). We came with a better idea, to use our competitive SPG and with few modifications, try and solve the constrained policy optimization problem. Towards that end, we searched for a way to incorporate the additional **cost signal**:

$$c \in \mathcal{C} \doteq \Big\{ c_{i,j} : (i,j) = (binary, scalar) \Big| c_i \in \mathcal{I}, c_j \in \mathbb{R}^+ \Big\} \tag{5.7}$$

which can be either a binary indicator of an unsafe state if 1 and also provide a scalar aggregate cost from the agent's sensors. As discussed in the Theory part 2.13 and have seen through Policy Gradient methods, we can approximate very well Q-functions, thus we tried to learn and constrain the Q^{cost}-function which is defined for our case as:

$$Q^{cost}(s,a) = \max_{\mu_\theta} \mathop{\mathbb{E}}_{\tau \sim \mu_\theta} \big[R(\tau) | s_0 = s, a_0 = a \big] \text{ and } R(\tau) = \sum_{t=0}^{\infty} \gamma_{cost}^t c_t \tag{5.8}$$

for a trajectory τ which computes the expected cost return if the start point is state s, the agent takes arbitrary action a and then always act according to the deterministic policy μ_θ. Remember that the discount factor for the reward case, maximizes the discount of rewards obtained later than rewards obtained earlier. In our case, we chose to have a low gamma values and treat them as hyperparameter $\gamma_{cost} \in (0, 0.65]$ as we care for more immediate costs. Essentially, we do not seek to approximate the long-term cost, rather the near-future or imminent one.

The addition of the Q^{cost} requires a second critic or cost critic. In order to train this critic, we follow the traditional DDPG procedure as described on 3.1.4. More specific, we train an approximator that is a neural network Q_ϕ^{cost} with parameters ϕ. We collect a set

$\mathcal{D} = (s, a, r, c, s', h_{term})$ of transitions where $h_{term} \in [0, 1]$ is a binary indicator whether state s' is terminal. Then to measure how close is Q_{ϕ}^{cost} to satisfy the Bellman equation, we try to minimize the MSBE:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,c,s',h)\sim\mathcal{D}}{\mathbb{E}} \left[\left(Q_{\phi}^{cost}(s, a) - \left(c + \gamma_{cost}(1 - h) Q_{\phi_{\text{targ}}}^{cost}(s', \mu_{\theta_{\text{targ}}}(s')) \right) \right)^2 \right] \quad (5.9)$$

where $\mu_{\theta_{targ}}$ denotes the target policy as we use Target Networks.

5.4.2.1 Safe-SPG

In this part, we introduce **Safe-SPG** and in the next section, we explain how we performed Safe Exploration by incorporating Transfer Learning. First, two additional hyper parameters $\mathcal{H} = \left\{ \delta_{1,2} : \delta_1 < \delta_2, \kappa | \delta_{1,2} \in \mathbb{R}^+, \kappa \in (0, 1) \right\}$ are added to control the *magnitude* of the constrains we enforce. In order to have the previous inequality between the δ values, we scale the second value with an appropriate scalar κ, whose value depends on whether the binary or the scalar cost signal is taken into account:

$$\delta_1 = \kappa \delta_2 \quad (5.10)$$

The small modification on the Offline Gaussian Exploration of SPG subsequently has the form:

$$If : Q(s_t, a_t) > Q(s_t, \mu_\theta(s_t)) - \delta_1 \wedge Q^{cost}(s_t, a_t) < Q(s, \mu_\theta(s_t)) - \delta_2 : Target(s_t) = a_t \quad (5.11)$$

summarized in the algorithm 6. This means the Q-value value could be a bit lower, but the cost should then be lower as well. The goal of this safe exploration is to explore the action space as good as possible, satisfying the constrains. In the end, from every batch of trajectories, Safe-SPG's exploration returns the best in terms of Q-value, actions α^* that satisfy the constraints.

On the actor's training side, concerning the policy learning part, we keep the same objective as DDPG where the goal is to learn a deterministic policy $\mu_\theta(s)$ which selects the action that maximizes $Q_\phi(s, a)$. This Q-function is differentiable with respect to action and it is straightforward to solve by gradient ascent:

$$\max_{\theta} \underset{s\sim\mathcal{D}}{\mathbb{E}} [Q_\phi(s, \mu_\theta(s))] \quad (5.12)$$

with respect to policy parameters only and treat the Q-function parameters as constants. It is important to note that other objective functions for the actor were tested like SPGR's 7.1 but due to incomplete experimentation, we present the baseline version here. This concludes a short and laconic description of Safe-SPG.

Algorithm 6: Offline Gaussian Safe Exploration

```
1  Input: Batch of $N$ tuples $(s_t, a_t, r_t, c_t, s_{t+1}, \sigma^2, \delta)$
2  for $(s_t, a_t)$ of transition $i$ in input do
3      best $\leftarrow \mu_\theta(s_t)$
4      if $Q(s_t, a_t) > Q(s_t, \text{best}) - \delta_1$ and $Q^{cost}(s_t, a_t) < Q(s, \text{best}) - \delta_2$ then
5          best $\leftarrow a_t$
6      end
7      for count in $1, ..., M$ do
8          sampled $\leftarrow$ best $+\mathcal{N}(0, \sigma^2)$
9          if $Q(s_t, \text{sampled}) > Q(s_t, \text{best}) - \delta_1$ and
            $Q^{cost}(s, \text{sampled}) < Q(s_t, best) - \delta_2$ then
10             best $\leftarrow$ sampled
11         end
12     end
13     if $Q(s_t,$ best$)$ then
14         Target$_i = best$
15     end
16 end
17 Output: Target actions represent the safe actions so far denoted $a^*$
```

5.4.2.2 Bounded Transfer Learning

After having conducted a series of experiments, first with Safety Layer and then with Safe-SPG, it became apparent that the agent due to constraints enforcement, had difficulties even driving around the environments. It was later observed, that this problem was partially alleviated by training for lengthier periods. To deal with situations like this, and inspired by the evolution in nature, where an organism first balances, then walks and finally runs avoiding obstacles; we propose a novel three-stage Transfer Learning scheme that despite not yet fully optimized, increased significantly the performance and constraint satisfaction rate of Safe-SPG. We call it Bounded because with Safe-SPG's exploration we bound an area on Q^{cost}-value by a threshold operation whose approximated value in the end of the training run is transferred to the next stage. More specific:

- **Stage 0**: In this stage the agent explores an obstacle-free environment without any cost signal, learning agile movement. We utilize DDPG's pure Q-learning to solve this environment, having a learning rate of 0.01 for the optimizer, to search as global as possible. The final actor's weights are saved and transferred to the next stage automatically upon finish of the run.

- **Stage 1**: In this stage, the agent explores an environment with two obstacles, only the one providing a cost signal, thus requiring one constrain to be satisfied. Here, we start training the second critic or cost critic with this cost signal, using Safe-SPG's exploration for safe action selection. During the training, we leave some time before beginning the evaluation process in order for the Q^{cost} to be approximated as good as possible. The learning rate of the actor's optimizer decreases to 0.001 while the δ_1 is initialized and decreases over the course of training with a constant κ. The final weights of the actor, cost critic and optionally the critic's are saved and transferred to the next stage automatically upon finish of the run.

- **Stage 2**: This is the final stage where the agent explores an environment with four different kind of obstacles, having four cost signals. To properly use these signals, we take the mean value of them and use it to train the cost critic if the aggregated cost is selected for the run. Else, we use the binary cost indicator to train the cost critic. Here, we start the training by first loading the weights from the stage 1 and then use Safe-SPG's exploration for safe action selection. During the training, we give some time to the networks to properly adjust their parameters before starting evaluating the agent. The learning rate of the actor's optimizer decreases to 0.0001, searching more local and deeper to the policy gradient, thus making small adjustments to the final policy. Because there are more obstacles resulting to more hazardous areas, we expect the Q^{cost} to be higher and further adjust the δ_1 and κ. The final weights of the actor are saved at the end of the training run.

The proposed Bounded Transfer Learning framework is presented schematically on figure 5.2. This concludes the presentation of the three-stage Transfer Learning scheme that was realized to solve the constrained optimization problem. In the next section, we present the experimentation setup and the results on section 7.4.2.

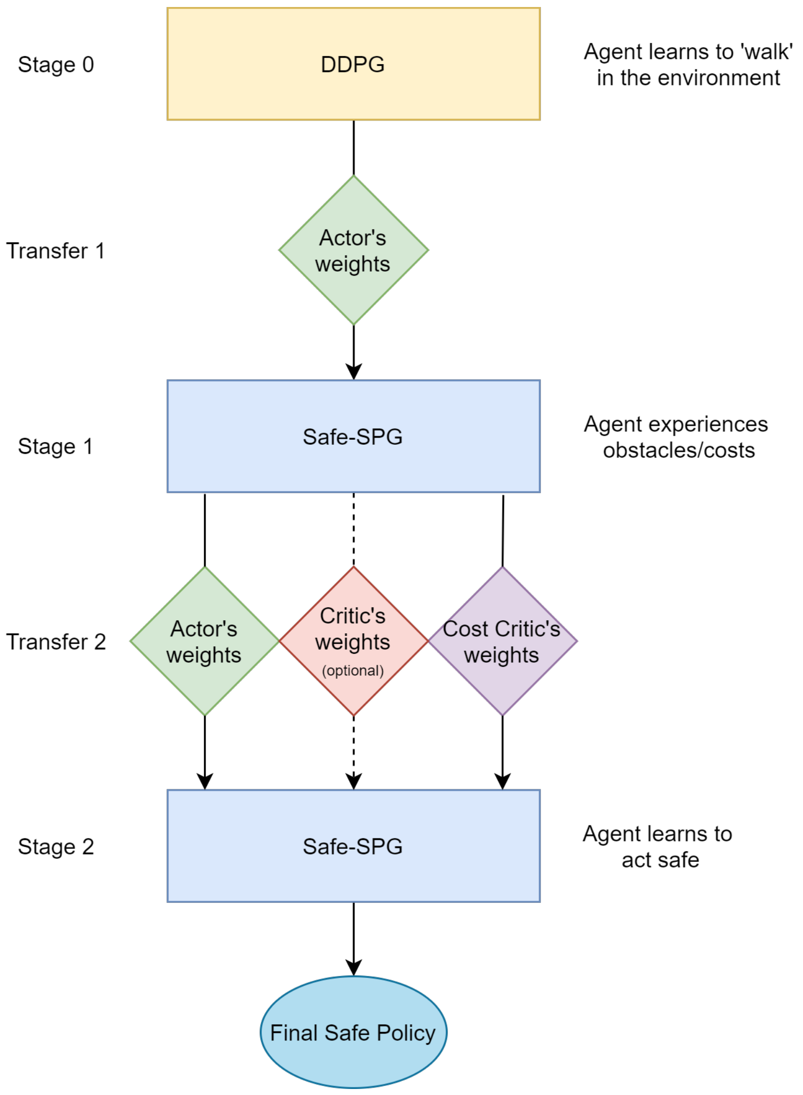


Figure 5.2: The proposed three-stage Bounded Transfer Learning pipeline.

6 Experiment Setup

We present the Sampled Policy Gradient with Offline Gaussian Exploration (SPG-OffGE), augmented with the Twin Delayed Deep Deterministic policy gradient algorithm (TD3) by applying the modifications described in Section 3.1.5 to increase the stability, performance and convergence properties with consideration of function approximation errors. The main theme of experiments is continuous control and in order to evaluate our algorithm, PyBullet physics engine (Coumans & Bai, 2016–2021), wrapped with OpenAI Gym (Brockman et al., 2016) was used. In addition Multi-Joint dynamics with Contact MuJoCo (Todorov et al., 2012) physics engine was used for hyper-parameter tuning. The implementation is pure Python3 (Van Rossum & Drake, 2009) with additional use of Pytorch (Paszke et al., 2019) library and can be found on GitHub[1].

To benchmark the PyBullet environments we utilized the University's High Performance Cluster (HPC) Peregrine and more specific its 36 nodes with 12 cores per node of Intel Xeon Gold 6150 @ 2.7 Ghz, 128GB memory and 1 Nvidia V100 GPU accelerator. To benchmark the MuJoCo environments my personal computer (PC) was used with an Intel i7 9700K @ 5 Ghz, 32GB memory and 1 Nvidia RTX 2080Ti.

6.1 Environments

The task of the environments is **robotic locomotion**, meaning the act or ability of the robot to transport or move itself from place to place. Essentially, given a set of actuators/motors which act and a set of sensors which observe and monitor the environment, the aim is to produce movement. The environments used for experimentation along with their action $\mathcal{A}$ and state $\mathcal{S}$ dimensions can be found on Table 6.1. It is useful to point out that the most difficult task is that of Humanoid, having the largest action and state space from the PyBullet domain.

Domain Pybullet	$\mid \mathcal{A} \mid$	$\mid \mathcal{S} \mid$	**Domain Mujoco**	$\mid \mathcal{A} \mid$	$\mid \mathcal{S} \mid$
HalfCheetah-v0	6	26	HalfCheetah-v2	6	17
Hopper-v0	3	15	Hopper-v2	3	11
Walker2d-v0	6	22	Reacher-v2	2	11
Ant-v0	8	28	Ant-v2	8	111
Minitaur-v0	8	28	InveredDoublePendulum-v2	1	11
Humanoid-v0	17	44	Swimmer-v2	2	8

Table 6.1: Environment domains with their action and state dimensions.

[1] https://github.com/DjAzDeck/SPG

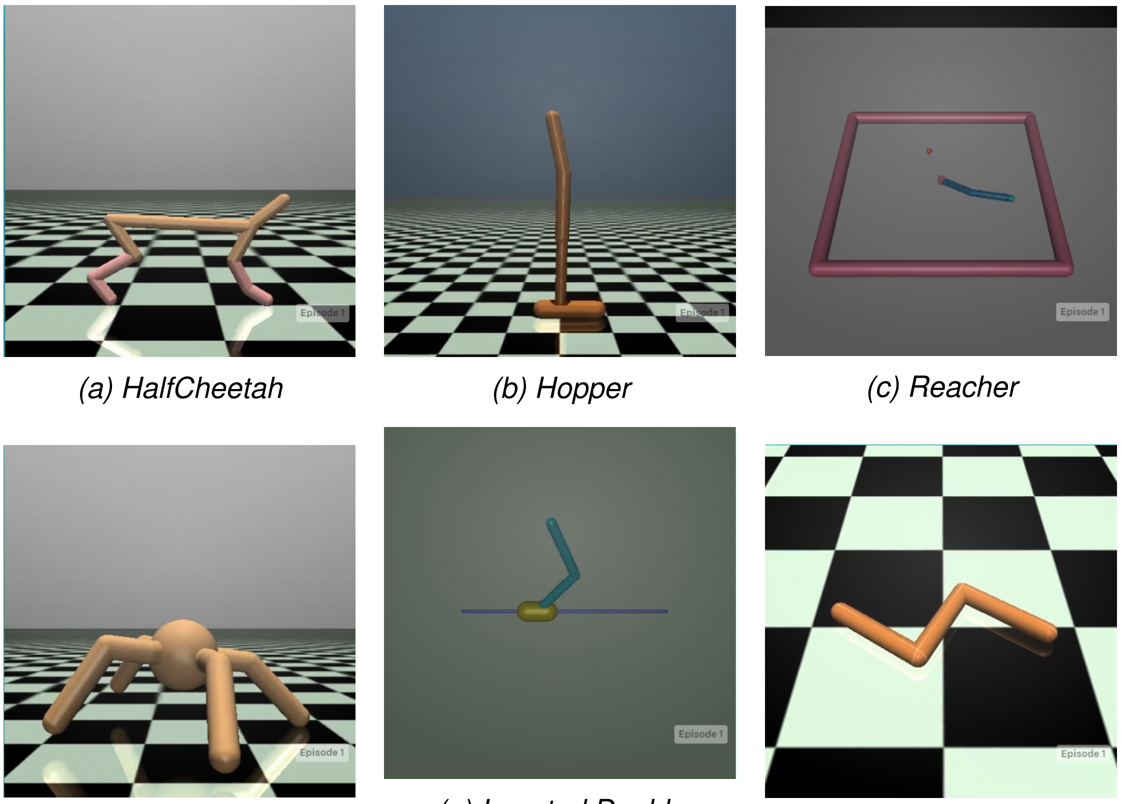
6.2 MuJoCo Experiments

The main motivation behind the use of MuJoCo physics engine was its speed in terms of experience accumulation from initial experimentation conducted, that is why it was preferred over PyBullet for hyperparameter tuning. A drawback of this simulator is its licence which requires payment but there is a free student license for 1 year. Another handicap is the single GPU linkage it enforces, banning its usage on HPC.

Two main hyperparameters were tested across the 6 tasks listed on the right side of the Table 6.1, the search number and the batch size. For these experiments the SPG with DDPG architecture was used rather than TD3 which is more complicated and a bit slower in terms of experience accumulation. The number of searches tested was $[8, 12, 16]$ and the batch sizes $[64, 128, 256]$. All the tasks ran for 1 million time steps with exactly the same settings 7.2 on the PC.

A brief overview of each task is given below while their graphic overview can be found on Figure 6.1.

- HalfCheetah: A 2D cheetah robot, where the goal is to balance and run as fast as possible. The reward gained is proportional to the distance traveled.
- Hopper: A 2D one-legged robot, where the goal is to hop forward as fast as possible.
- Reacher: A 2D robot in a squared arena, where the goal is to reach the randomly located target.
- Ant: A 3D four-legged robot, where the goal is to walk forward as fast as possible.
- Inverted Double Pendulum: A 2-link pole, where the goal is to balance while on top of a moving cart.
- Swimmer: A 3-link swimming robot in a viscous fluid, where the goal is to make it swim forward as fast as possible, by actuating the two joints.



(a) HalfCheetah (b) Hopper (c) Reacher

(d) Ant (e) Inverted Double Pendulum (f) Swimmer

Figure 6.1: Graphic illustration of the Mujoco domain tasks.

6.3 Pybullet Experiments

This physics engine was used for large-scale experimentation mainly because it is open source and does not come with any GPU linkage constraints. It was found to be slower in comparison with Mujoco in terms of experience accumulation but this is mitigated when HPC is used. Furthermore, the extensive documentation provided by the developers made its usage easier in comparison with the brief and laconic documentation of Mujoco.

The environments tested can be found on the left side of Table 6.1. Here, after the hyperparameter tuning is done, the aim is to compare the performance of SPG with the state-of-the-art models, leading to the usage of both architectures DDPG and TD3. All the tasks ran for 48 hours, accumulating a different number of experience mainly depending on the environment and less on the architecture.

A brief overview of each task is given below while their graphic overview can be found on Figure 6.2.

- HalfCheetah: A 2D cheetah robot, where the goal is to balance and run as fast as possible. The reward gained is proportional to the distance traveled and is similar with the Mujoco version.
- Hopper: A 2D one-legged robot, where the goal is to hop forward as fast as possible and is similar with the Mujoco version
- Ant: A 3D four-legged robot, where the goal is to walk forward as fast as possible. Here the ant is heavier, encouraging it to typically have two or more legs on the ground.
- Walker: A 2D bipedal robot, where the goal is to walk forward as fast as possible.
- Minitaur: A quadruped robot with eight-drive actuators based on the platform from Ghost Robotics. The goal is to move fast forward and gallop while the reward encourages fast running speed and penalizes high energy consumption.
- Humanoid: A 3D bipedal robot, where the goal is to walk forward as fast as possible, without falling over. This robot benefits from more realistic energy cost (torque $\times$ angular velocity) subtracted from the reward.

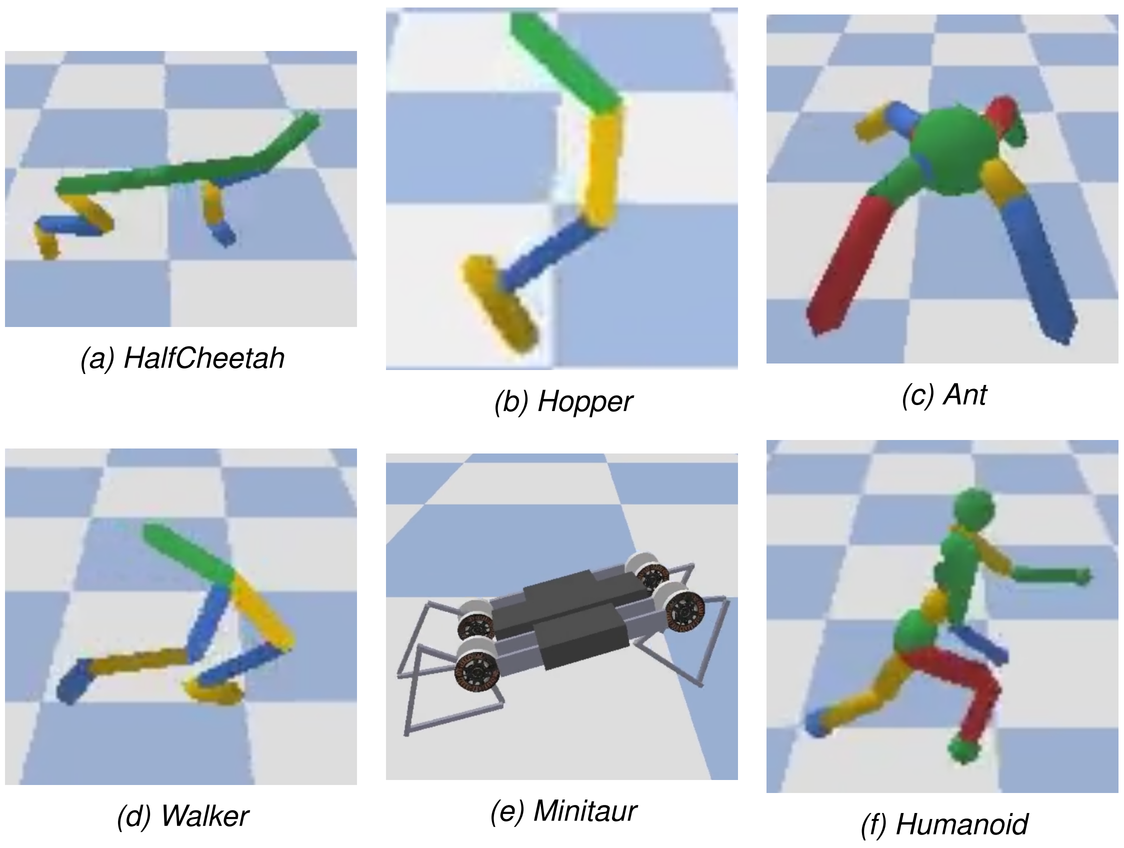

(a) HalfCheetah
(b) Hopper
(c) Ant
(d) Walker
(e) Minitaur
(f) Humanoid

Figure 6.2: Graphic illustration of the PyBullet domain tasks.

6.4 Safety Gym Experiments

In order to evaluate our methods, we used the OpenAI's Safety Gym suite (Ray et al., 2019), that comes with a variety of constrained environments and 3 base agents. We omit to present the full environments and configurations in order to avoid the overpopulation of this chapter with images, but we encourage the curious reader to visit the OpenAI's blog post (Joshua Achiam, n.d.) to get a visual impression. In this work, we decided to experiment with with the Point environment and the Goal task. An illustration of the environment for level 2 can be found on figure 6.3 and the agent's state and action space dimensions are $|\mathcal{S}| = 60, |\mathcal{A}| = 2$. The Safety suite offers three levels of difficulty to experiment with, thus we adopted our Transfer Learning method similarly.

To compare the training runs and the performance of our agent, we keep track on the *average binary cost per evaluation run* ρ_c, which is computed every 10.000 time steps, with a maximum length of 1000 time steps. We believe that this metric provide us with a solid feedback if the agent avoids any kind of obstacles, by satisfying the constraints. By plotting this average cost per evaluation for the whole training run, we can draw useful conclusions for both methods tested.

$$\rho_c \doteq \sum_{t=1}^{T} c_i, i = binary \tag{6.1}$$

On the implementation side, the Safety Gym requires Mujoco engine to work, thus making the use of HPC impossible, as discussed previously on 6.2. Because of that, my personal computer (PC) was used to conduct experiments.

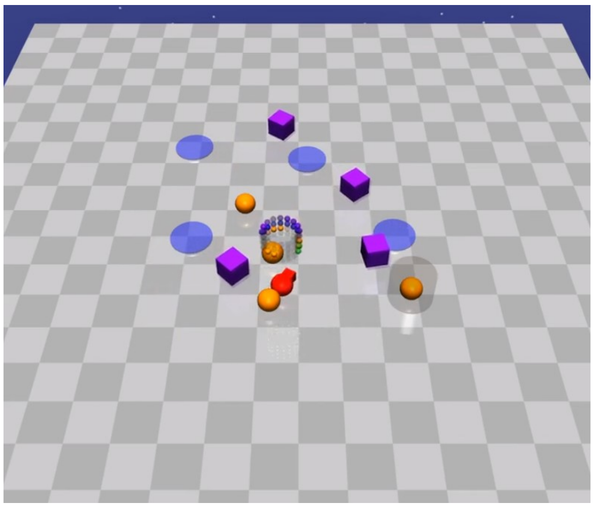


Figure 6.3: The PointGoal-2 environment

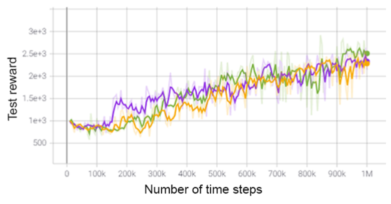
7 Results

In this chapter we present the results of our experiments and discuss our findings. In the first section we present and discuss the results from hyperparameter tuning. In the second section we present and discuss the results from the use of prioritized replay buffer. In the third section we present and discuss the results from the large-scale experiments. In the fourth and final section we present and comment on the results from the safety experiments.

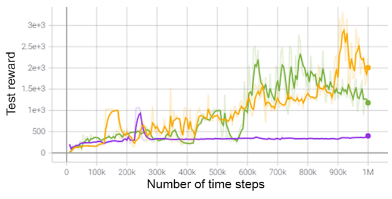
7.1 Hyperparameter Tuning

Starting from figure 7.1, we show the results obtained using SPG with DDPG architecture and batch size 64 while the full parameters and their corresponding values can be found on table 7.2. We consider the test phase to be every 5000 training time steps where the agent is called to perform on the test environment for 1000 time steps. On the top row we show the reward achieved at test phase while on the bottom row we show the speed on training phase. The speed is measured by counting the frames processed per second (fps): **speed** $= \frac{\#frames}{seconds}$. We believe this is the best way to highlight the trade-off between the number of searches and the effect this number has on the algorithm's speed and test reward.

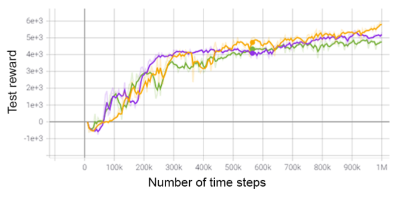
We can infer from figure 7.1 that performing 8 searches led to higher reward in most of the environments while on other environments like Ant, the difference is negligible in comparison with the speed performance. The speed results vary on a magnitude of $\sim 7-10\%$ across different environments and can be found analytically on table 7.1. We note the oscillation behaviour observed in figure 7.6f mostly towards the end, is due to performing other tasks on PC the time of training which effected the performance of the specific run.



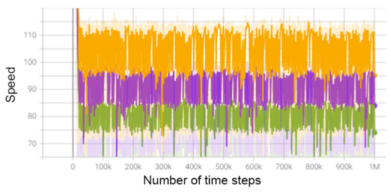
(a) Ant



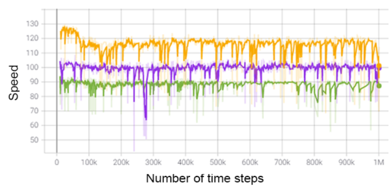
(b) Hopper



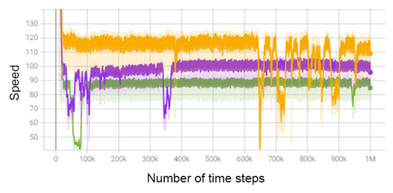
(c) Cheetah



(d) Ant

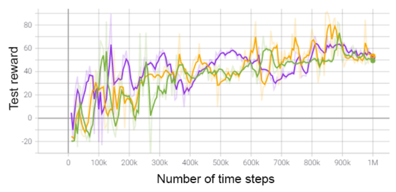


(e) Hopper

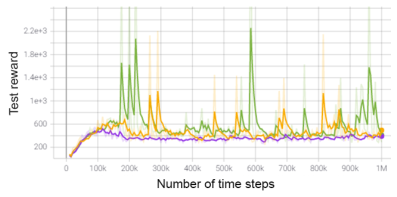


(f) Cheetah

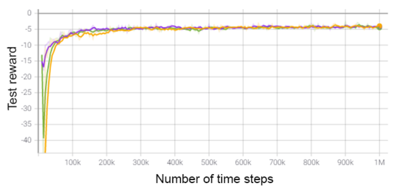
Figure 7.1: Test reward and speed plots for batch size 64. The number of searches is highlighted with colours – 8, – 12 and – 16.



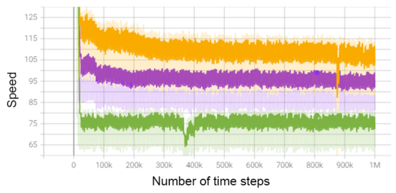
(a) Swimmer



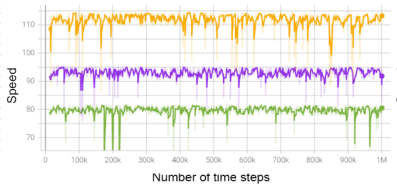
(b) Double Pendulum



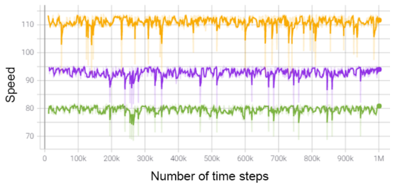
(c) Reacher



(d) Swimmer



(e) Double Pendulum



(f) Reacher

*Figure 7.2: Test reward and speed plots for batch size 64. The number of searches is highlighted with colours **– 8**, **– 12** and **– 16**.*

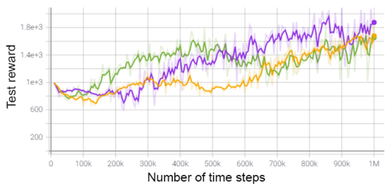
Continuing to figure 7.2, we can observe the results from the other 3 tasks. The test reward dynamics follow the same trend with the previous three environments, with a notable difference on the Inverted Double Pendulum task 7.2b. In this task, performing 16 searches led faster to higher rewards at least three times in 1 million time steps. Also, performing 8 searches led to high rewards but with lower magnitude. Performing 12 searches had the lowest test reward magnitude, following a flat trend till the end of the run. On the speed performance side and observing the bottom row of figure 7.2, makes clear the effect of search number. Similarly with the previous 3 tasks, performing 8 searches had the best performance while performing 16 searches the worst.

In general, performing 8 searches on a batch size 64 had an average speed of 111 fps, performing 12 searches had an average speed of 95.3 fps and performing 16 searches had an average speed of 81 fps. With these numbers, we can calculate the averaged performance difference per number of searches in percentage to be $P_V = \frac{|V_1 - V_2|}{\frac{V_1+V_2}{2}} \cdot 100$. This informs us that by doubling the number of searches from 8 to 16, the average performance speed drops by 31.25%.

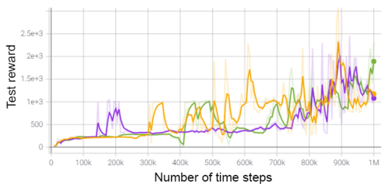
In figure 7.3 we show the results obtained when the batch size increase to 128. The reward follows similar trend with the 64-size version but now the difference between the number of searches is more apparent. Furthermore, from table 7.1 we can see the impact of bigger batch (double size) on the average speed. There is an average speed reduction of 8-10% per search number across all tasks, an expected result from the standpoint of when increasing the training examples, more information needs to be processed. On figure 7.4 we can observe the performance plots from the other three tasks, starting from reward plots on top row and speed plots on the bottom row. We note the double inverted pendulum task in figure 7.4e that exhibited superior performance with 16 searches, a result similar with the 64-version. The

Reacher task had analogous reward dynamics with the smaller batch version while the Swimmer exhibited a steep increase in the reward for 16 searches on 500k time steps; following the trend of the other search numbers afterwards.

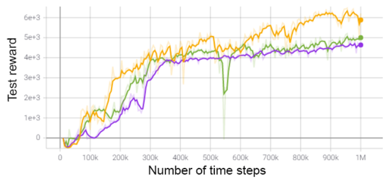
In general, performing 8 searches on a batch size 128 had an average speed of 103.8 fps, performing 12 searches had an average speed of 86.5 fps and performing 16 searches had an average speed of 74.8 fps. It is important to note that these numbers are lower in comparison with the 64 batch size, a natural consequence of the increased size. Next, we calculate the difference per number of searches in percentage, that is increasing the number from 8 to 12, decrease the average performance speed by 18.18%. Doubling the searches from 8 to 16, decreased the average performance speed by 32.47%.



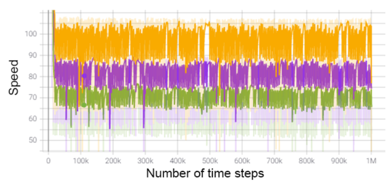
(a) Ant



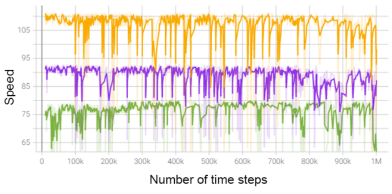
(b) Hopper



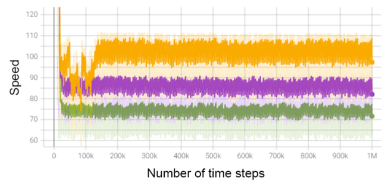
(c) Cheetah



(d) Ant

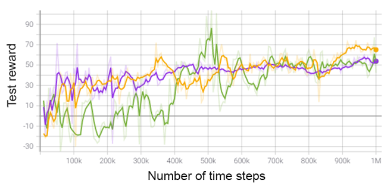


(e) Hopper

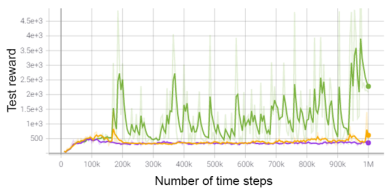


(f) Cheetah

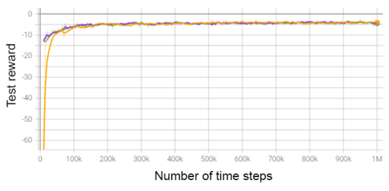
*Figure 7.3: Test reward and speed plots for batch size 128. The number of searches is highlighted with colours **– 8**, **– 12** and **– 16**.*



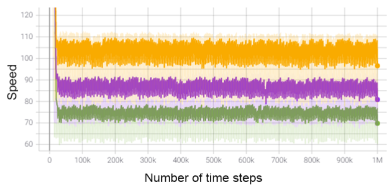
(a) Swimmer



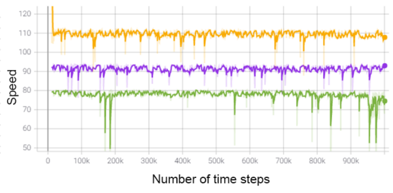
(b) Double Pendulum



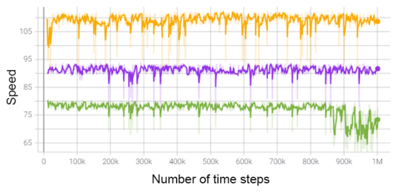
(c) Reacher



(d) Swimmer



(e) Double Pendulum



(f) Reacher

Figure 7.4: Test reward and speed plots for batch size 128. The number of searches is highlighted with colours ***– 8***, ***– 12*** *and* ***– 16***.

# searches	8	12	16	# searches	8	12	16	# searches	8	16
Env (batch)	Average fps			batch	Average fps			batch	Average fps	
Ant (64)	105	93	79	(128)	95	81	72	(256)	94	68
Hopper (64)	115	98	89	(128)	104	85	76	(256)	92	72
Cheetah (64)	111	97	88	(128)	103	87	75	(256)	97	73
Swimmer (64)	114	97	73	(128)	104	86	76	(256)	98	73
Double Pendulum (64)	109	93	79	(128)	109	89	78	(256)	107	76
Reacher (64)	112	94	78	(128)	108	91	72	(256)	105	77
Average	111	95.3	81	(128)	103.8	86.5	74.8	(256)	98.8	73.1
Percentage Difference (%)		15.22	31.25	(128)		18.18	32.47	(256)		29.9

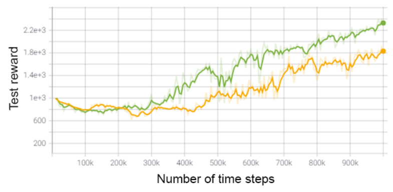
Table 7.1: Average speed (fps) round to integer part, per environment and number of searches.

Parameter	Value
optimizer	Adam
learning rate	0.0001
discount(γ)	0.99
replay buffer size	1000000
number of hidden layers	2
number of hidden units per layer	[400, 300]
number of samples per mini batch	[64, 128, 256]
number of searches	[8, 12, 16]
Noise (μ, σ)	(0,1)
activation function	ReLU
warm-up steps	10000

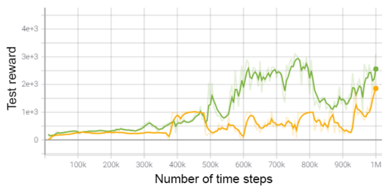
Table 7.2: SPG Hyperparameters

The final experiments for this part were conducted by increasing the batch size even more to 256 and searching 8 and 16 times. The performance plots can be found on figure 7.5. Starting from the Ant task, we can immediately spot that performing 16 searches yield higher reward than performing 8. The same result stands also for the Hopper task, where there is a steepest increase in the reward after 500k time steps. For the Cheetah task, performing 8 searches led to higher reward earlier in the train but after 650k time steps, performing 16 searches exhibited comparable and slightly better reward. Figure 7.6 shows the performance results from the other 3 tasks. Starting from the Swimmer, performing 8 searches led to higher reward towards the end of the run. Double inverted pendulum exhibited a sharp increase in reward around 200k time steps for both 8 and 16 searches. Reacher's reward followed the same trend with the previous batch size version without providing any significant change worth mentioning.

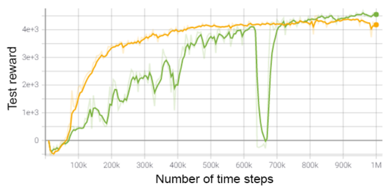
In general, performing 8 searches on a batch size 256 had an average speed of 98.8 fps while performing 16 searches had an average speed of 73.1 fps. It is important to note that these numbers are slightly lower in comparison with the 128 batch size. In comparison with the 64 batch size, the average speed difference per search number is 12.2 fps for 8 searches and 7.9 fps for 16 searches. Also, the difference per number of searches in percentage by doubling the searches from 8 to 16 is 29.9%.



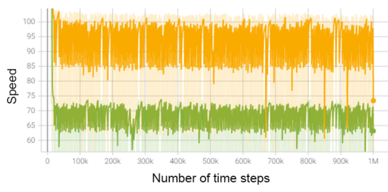
(a) Ant



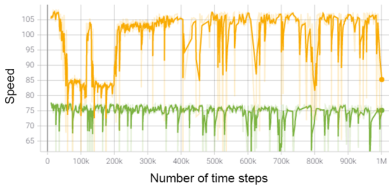
(b) Hopper



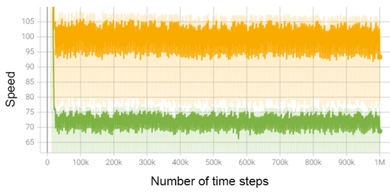
(c) Cheetah



(d) Ant

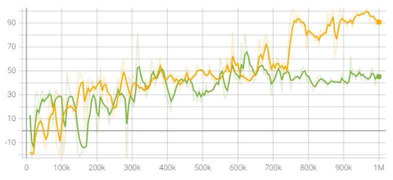


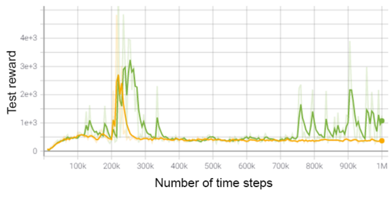
(e) Hopper



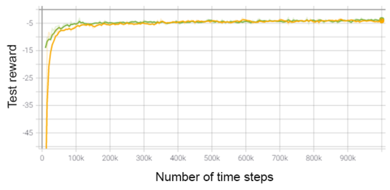
(f) Cheetah

Figure 7.5: Test reward and speed plots for batch size 256. The number of searches is highlighted with colours – 8 and – 16.

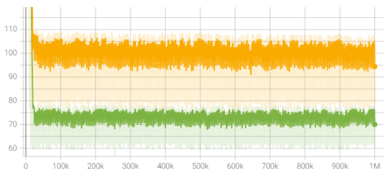

(a) Swimmer

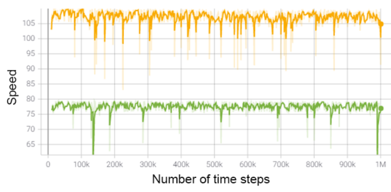


(b) Double Pendulum

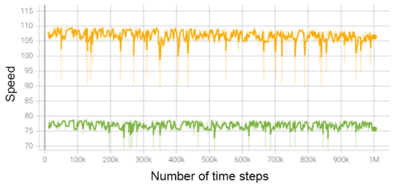


(c) Reacher


(d) Swimmer



(e) Double Pendulum



(f) Reacher

Figure 7.6: Test reward and speed plots for batch size 256. The number of searches is highlighted with colours – 8 and – 16.

To sum up this part, we presented the experimentation setup for testing the number of searches on different batch sizes. We presented the results in a meaningful way by including the reward and speed plots to highlight the trade-off between reward gain and speed bottleneck. From our experiments, we can verify the optimal number of searches for a batch size 64 to be 8. For a batch size of 128, performing 8 searches had positive effect on some tasks but also performing 16 searches offered improvements in some others. In general, performing 12 searches did not offer any significant improvement worth mentioning. Finally, for a batch size 256 we found that performing 16 searches is more advantageous in most of the tasks. Nevertheless, doubling the number of searches imposed a big performance bottleneck; decreasing the average speed approximately by 30%. This may be tolerable for Mujoco engine that is faster in terms of experience accumulation but for PyBullet engine this bottleneck cripples the performance of SPG. That is the main reason we decided to use batch size 64 and perform 8 searches in the forthcoming experiments.

7.2 Prioritized Replay Buffer

In this part, we present the results from experimentation with Prioritized Replay Buffer as an extension over the default replay buffer, introduced in chapter 4. The experiments goal is to highlight if prioritized replay buffer offers any significant performance increase as the literature suggest. Two versions of SPG were tested, one with a default replay buffer and one with a prioritized one. The exact parameters used for these experiments can be found on table 7.3. The models ran on HPC for 72 and 24 hours for default and prioritized version respectively.

Parameter	Value
optimizer	Adam
learning rate	0.0001
discount(γ)	0.95
prioritized replay buffer size	1000000
prioritized a	0.6
prioritized β start	0.4
prioritized decay steps	2000000
number of hidden layers	2
number of hidden units per layer	[400, 300]
number of samples per mini batch	64
number of searches	8
Noise (μ, σ)	(0,1)
σ decay steps	6000000
activation function	ReLU
warm-up steps	20000

Table 7.3: Prioritized SPG Hyperparameters

Starting from the Ant environment and figure 7.7 we spot the big reward difference between the default and the prioritized version of SPG. The prioritized version achieved very early after 1 million time steps, steepest increase in the test reward. The actor's loss (eq. 4.3) magnitude, is smaller and converges faster near zero for the prioritized version while the actor's loss for the default version takes almost 8 million time steps. Regarding the critic's loss (eq. 4.2), the default version is very spiky, indicating noisy and unstable training while the prioritize version is more restrained. Finally the average speed performance is slightly lower for the prioritized version and more specific the average fps were 42 and 55 for the default version respectively. The performance cost of 12 fps versus the test reward increase we believe is tolerable and unavoidable when you perform any kind of weight sampling and storing technique (of course, the fps' range per se heavily depends on the implementation).

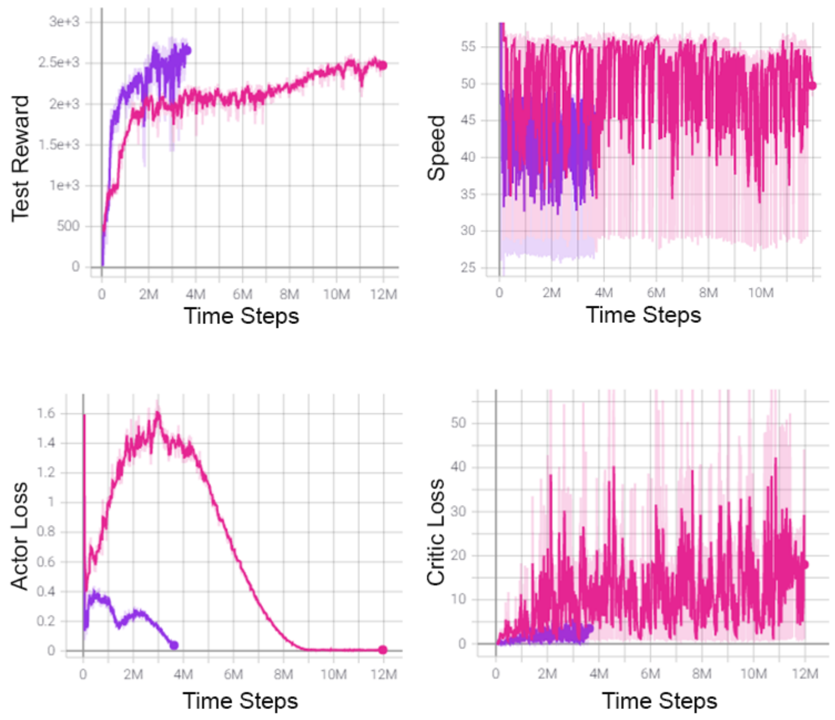


Figure 7.7: Performance plots for the Ant environment with ***Default SPG*** *and* ***Prioritized SPG****.*

Continuing with the Cheetah environment and figure 7.8 we don't find the robust increase in the test reward for the prioritized version, rather we could say it exhibits similar dynamics with the default version. In this environment the average speed performance difference is more apparent with average fps 50 and 37 for the default and prioritized version respectively. The actor's loss has not converged by the end of the run for the prioritized version while the critic's loss exhibits similar spiky and noisy dynamics with the default version. In this environment, the prioritized version did not offer any significant gain in test reward with a performance cost of 13 fps.

The next environment is Walker and its performance plots can be found on figure 7.9. From the test reward plot we can immediately spot the superiority of the prioritized version in terms or reward gain. The actor's loss converges faster near 0 in 4 million time steps for the prioritized in comparison with the default version that takes 9 million time steps. Also, the critic's loss magnitude is more conservative and avoids the extrema of the default version. In terms of average speed performance, the prioritize version is slower with an average speed of 43 versus 52 fps of the default version. This performance cost of 9 fps in comparison with the higher reward dynamics, justifies the use of the prioritized version to solve this task.

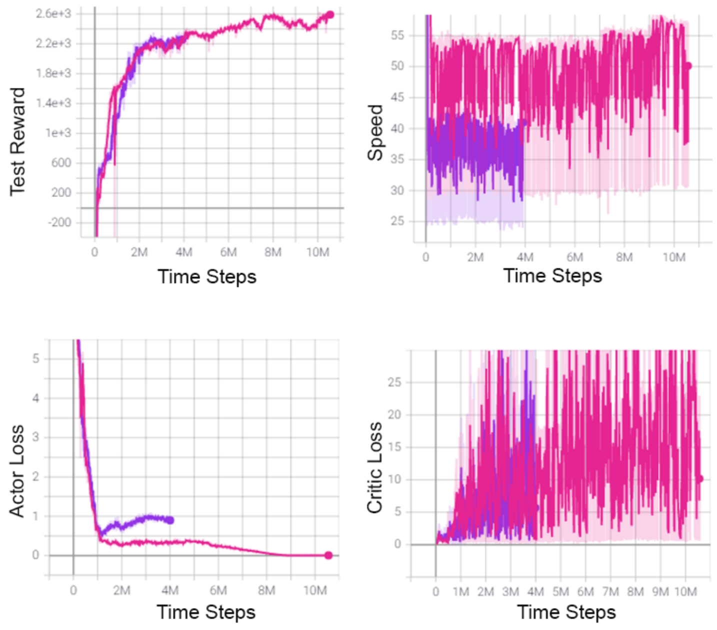


*Figure 7.8: Performance plots for the Cheetah environment with **Default SPG** and **Prioritized SPG**.*

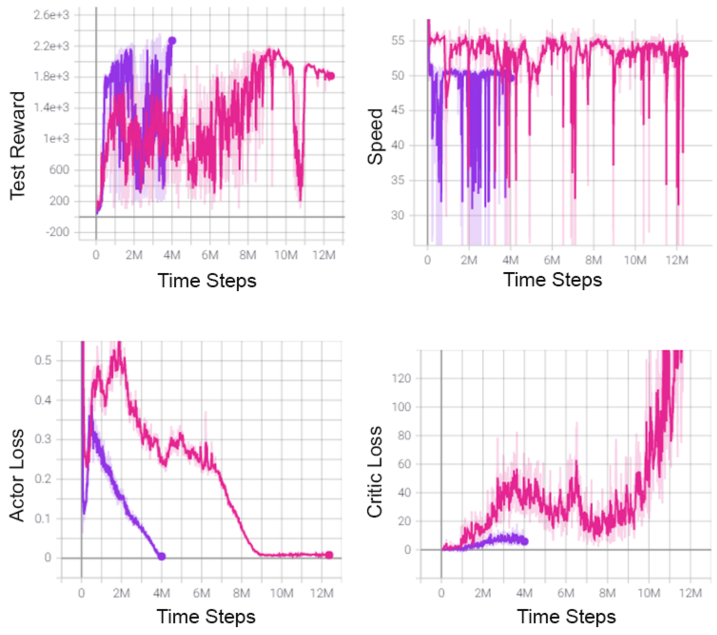


*Figure 7.9: Performance plots for the Walker environment with **Default SPG** and **Prioritized SPG**.*

The final environment we discuss, namely Humanoid, is one of the most difficult ones, having very large state and action spaces as noted in table 6.1 and is rarely included into papers as derived from my research. The performance plots can be found on figure 7.10 and from the first plot we can immediately spot the steepest increase in the test reward for the prioritized version. The prioritize version is able to reach higher test reward in the course of training immediately after 2.5 million time steps. The actors loss has lower magnitude for the prioritized version in comparison with the default one, following similar trend. It is apparent that the prioritized has not converged yet as the loss is still decreasing and following the default's version trend we expect it to reach closer to 0. The critic's loss magnitude is smaller from the prioritized version in comparison with the default one. Finally, the average speed is higher for the default version by a margin of 8 fps with 49 and 41 for the prioritized version respectively. We believe that the performance cost of 8 fps is negligible in comparison with the reward gain and the smoother overall training dynamics that resulted in faster convergence.

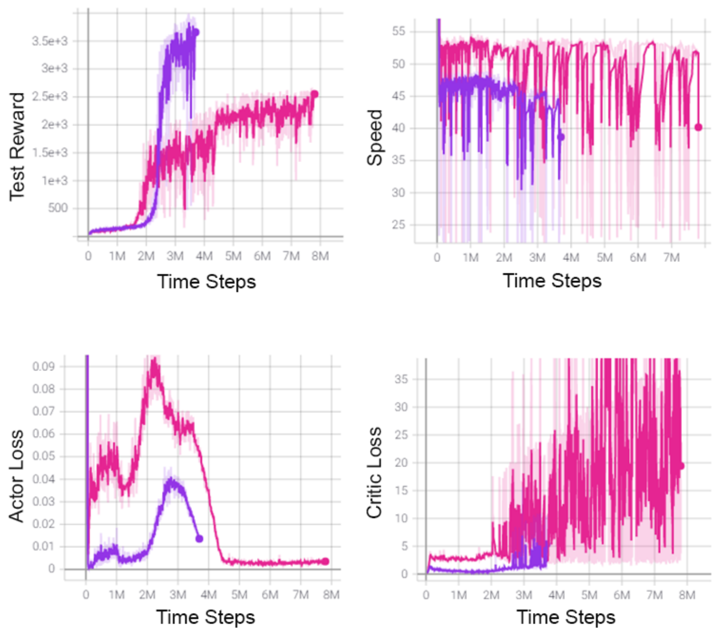


Figure 7.10: Performance plots for the Humanoid environment with ***Default SPG*** *and* ***Prioritized SPG****.*

Last but not least, it is important to comment on the reward statistics available on table 7.4, obtained from evaluating the final weights 10 times for 1000 time steps. Also, note that default SPG ran for 72 hours while the prioritized SPG only for 24 hours. Starting from the Ant task, prioritized SPG achieved a mean reward of 2826, 13.76% more than default SPG. In the Cheetah, prioritized SPG scored 2623 mean reward in comparison with 2546 for the default version, that is 2.97% increase. Next, on the Walker task, prioritized SPG achieved mean reward 2387, 12.6% higher than 2104 of default SPG. Finally, on the Humanoid task, prioritized SPG scored average reward of 4004, 33.29% more than the default SPG that scored 2861.

Task: Ant	mean	max	std	Task: Cheetah	mean	max	std
Algorithm							
Default SPG	2462	2489	14.273		2546	2554	12.12
Prioritized SPG	**2826**	2859	16.664		**2623**	2654	17.209
Percentage diff.	13.75%				2.97%		
Task: Walker				Task: Humanoid			
Algorithm							
Default SPG	2104	2132	13.889		2861	2920	25.967
Prioritized SPG	**2387**	2409	11.994		**4004**	4035	18.764
Percentage diff.	12.6%				33.29%		

Table 7.4: Reward statistics from final weights evaluation. Bolt numbers indicate the highest mean reward.

To sum up, this round of experiments highlighted the importance of prioritized sampling when selecting experience for Q-learning, in an off-policy algorithm. As verified from our experiments, this method had a positive impact on the overall performance of SPG on different tasks; some of them benefited more while one of them, the Cheetah, less. The most surprising result was that of the Humanoid that verified the ability of SPG to solve challenging tasks and opens the road for more complicated ones with even larger state and action spaces. In the next section we will keep using the prioritized SPG and provide the results for a lengthier train version.

7.3 Large-scale Experiments

In this section we present the experiments conducted on HPC for 48 hours using SPG augmented with DDPG and TD3 architecture. Our aim here is to highlight if combating bias overestimation in Q-learning offers any significant improvement to the SPG. First, we present the results of tuning the noise parameter on the SPG. Then, we present our findings when using SPG along with the TD3 architecture in two configurations.

7.3.1 Noise Hyperparameter

During our initial experiments, various values of the variance σ^2 in the Gaussian distribution $\mathcal{N}(0, \sigma^2)$ were used, leading to ambiguous results. It was also observed that the decay rate of this parameter had also an effect on the final policy. For this reason, we provide the experimentation results utilizing the prioritized SPG, for 3 different variance values [0.7, 0.8, 0.9]. We have tested smaller values that did not offer any significant improvement to the overall performance, thus for the clarity of figures, are omitted. The complete list of hyperparameters used can be found on table 7.5.

Starting from the Cheetah environment and figure 7.11, we observe that initializing the variance parameter at 0.7, resulted in the best final performance, while using the value 0.8 had a faster increase in the test reward as can be seen from the test reward plot. It is worth mentioning that using the value 0.9 resulted a faster convergence of the actor as can be seen from the actor's loss plot, but did not yield a better final test reward.

Next, on figure 7.12 we can observe the performance plots for the Ant environment. In this environment, using the value 0.9 resulted better test reward, while the other two values follow similar trend till the end of training. On the actor's loss side, using the value 0.9 resulted to faster convergence at almost 2 million time steps.

The next environment is Hopper and the performance plots can be found on figure 7.13. From the test reward plot it is apparent that the value 0.7 yield the best final test reward by far. The other two values did not provide any comparable performance gain. It is worth mentioning that the actor's loss has not converged for the value 0.7 by the end of training in contrast with the other two values.

Continuing with the Walker environment and the performance plots that can be found on figure 7.14, we observe an oscillation trend in the test reward for both values 0.7 and 0.9. Using the value 0.8 had a positive effect early in the training but after 5 million time steps, the performance degraded. The actor converged faster near 0 for the value 0.9 but the spiking behaviour of the critic's loss is also reflected in the test reward plot which exhibits steep decreases.

Next, the Humanoid environment and its performance plots can be found on figure 7.15. The values 0.7 and 0.8 follow similar trends, with the value 0.7 having an the earliest steep increase in terms of test reward around 2.5 million time steps. The actor's loss converges near 0 for the 0.9 value, while the other two values follow akin trend.

The final environment we tested our algorithm is the Minitaur and its performance plots can be found on figure 7.16. In this environment the value 0.7 performed better as can be seen from the test reward plot. Furthermore, the actor's loss has not converged yet for this value in contrast with the other two values. We can also observe the noisy plot on the critic's loss for the value 0.7.

To conclude this part of experiments, we found out that using $\sigma^2 = 0.7$ to initialize the variance parameter of the Gaussian distribution, subsequently the noise parameter, had the most positive effect during the runs. First, with this setting, the prioritized SPG was able to solve all the environments with high reward return. It became clear that big variance values but also low values like 0.3 and 0.4 that were also tested, resulted into under performance, yielding instances where the agent was not able to fully solve the task. For the large values it is an expected result because as observe when evaluating the algorithm's internal returns, many actions from SPG take the values [-1, 1] due to the clipping mechanism implemented. It is natural that large action values early in the training that may exist in the batch and are explored by the SPG by adding also big noise values sampled from a Gaussian with large variance, will surpass the bounds, resulting actions $a > 1$ or $a < -1$. This leads to extreme loss of information and batches almost full of -1 and 1 values, trapping the agent to under optimal actions. We expect that this is alleviated when bigger batches sizes are used but still, the exploration will not be successful. The explanation for the under performance of lower variance values i.e 0.3, 0.4 is that the agent does not explores sufficiently the action space, resulting into poor final performance.

Parameter	Value
optimizer	Adam
learning rate	0.0001
discount(γ)	0.95
prioritized replay buffer size	1000000
prioritized α	0.6
prioritized β start	0.4
prioritized decay steps	2000000
number of hidden layers	2
number of hidden units per layer	[400, 300]
number of samples per mini batch	64
number of searches	8
Noise variance σ^2	[0.4, 0.6, 0.7, 0.8, 0.9]
σ decay steps	6000000
activation function	ReLU
warm-up steps	30000

Table 7.5: Prioritized SPG Hyperparameters

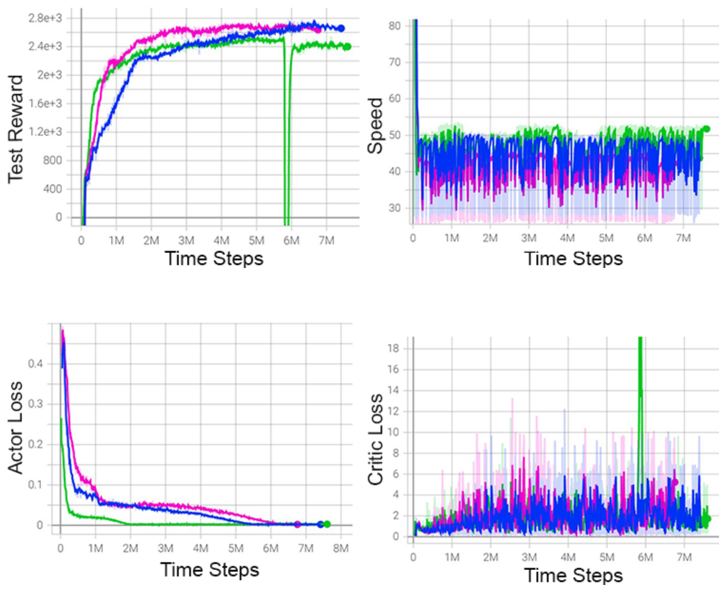


*Figure 7.11: Performance plots for the Cheetah environment with SPG and variance parameter **0.7**, **0.8** and **0.9**.*

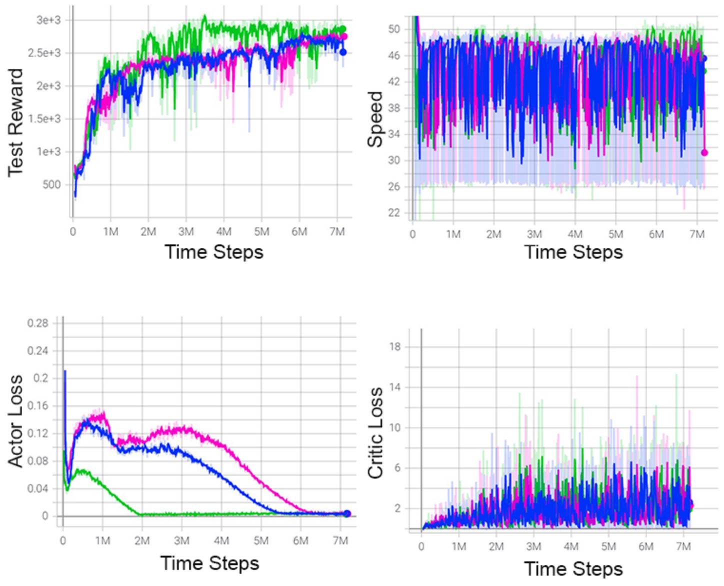


*Figure 7.12: Performance plots for the Ant environment with SPG and variance parameter **0.7**, **0.8** and **0.9**.*

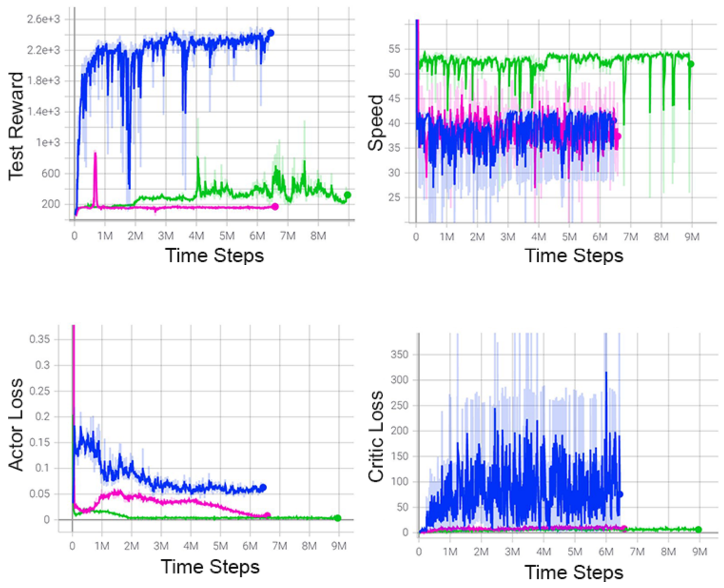


*Figure 7.13: Performance plots for the Hopper environment with SPG and variance parameter **0.7**, **0.8** and **0.9**.*

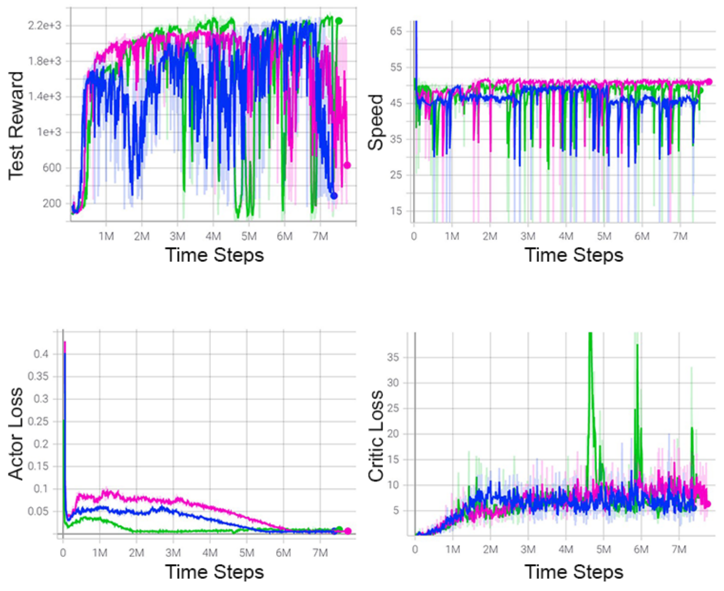


*Figure 7.14: Performance plots for the Walker environment with SPG and variance parameter **0.7**, **0.8** and **0.9**.*

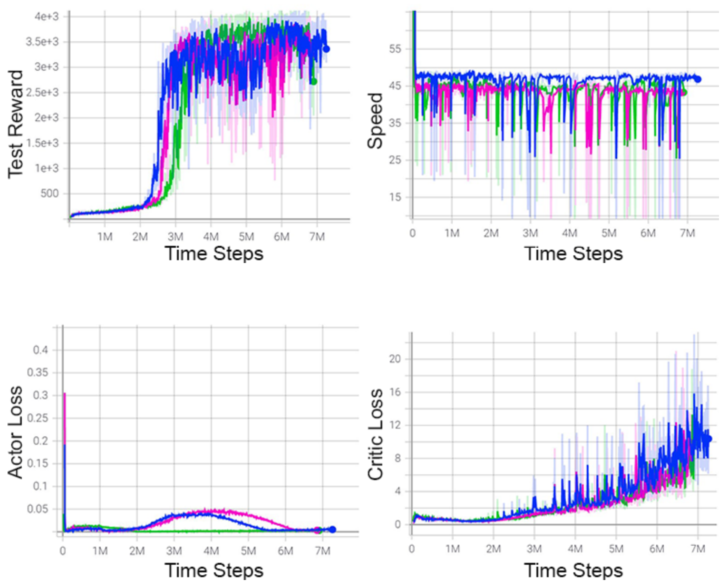


*Figure 7.15: Performance plots for the Humanoid environment with SPG and variance parameter **0.7**, **0.8** and **0.9**.*

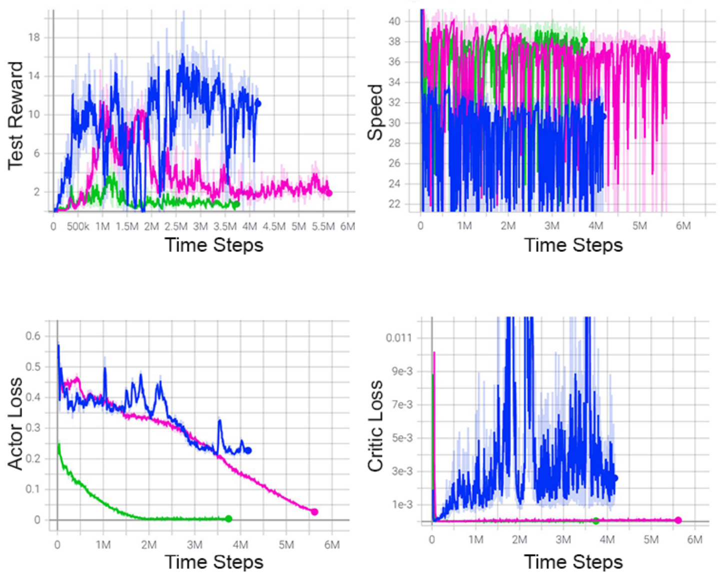


*Figure 7.16: Performance plots for the Minitaur environment with SPG and variance parameter **0.7**, **0.8** and **0.9**.*

7.3.2 Adding TD3

In this section we use the optimized SPG and investigate if combating bias overestimation with the incorporation of the TD3 architecture 3.1.5 offers any significant performance improvement. Towards that end, we investigated an upgraded version of this architecture by (Fujimoto & Gu, 2021) called TD3+BC (behavior cloning) that simply adds a term to regularize the policy. In total we present the results from two variants of SPG:

- **SPG+TD3**: In this setting we use the architecture TD3 as described in 3.1.5 with only difference on the exploration of action space that replaces the clipped noise addition of TD3 with the SPG's exploration.
- **SPGR+TD3**: In this setting we use the architecture TD3 and inspired by the work of Fujimoto and Gu, we add a behaviour cloning term coming from SPG to **R**egularize the policy. More specific, we train the policy μ_θ to maximize:

$$\underset{\mu_\theta}{\operatorname{argmax}} \underset{s\sim\mathcal{D}}{\mathbb{E}}\left[\lambda Q(s,\mu_\theta(s)) - (\mu_\theta(s) - \alpha^*)^2\right] \tag{7.1}$$

where α^* is the best action found from SPG's exploration and λ is a hyperparameter to control the strength of the regularizer. We keep the same scalar λ definition as the literature suggests:

$$\lambda = \frac{\text{a}}{\frac{1}{N}\sum\limits_{(s_i,\alpha_i^*),i=1}^{N} \mid Q(s_i,\alpha_i^*) \mid} \tag{7.2}$$

This term serves as an estimator of the average absolute value of Q over a dataset of N transitions (s, α^*), in our case a mini-batch with size of $N = 64$. This estimator is not differentiated over and simply serves to scale the loss.

The main purpose of the SPGR is to to combat the *extrapolation error* generated while evaluating a policy when the agent poorly estimates the value of state-action pairs not contained in the dataset. The solution provided above is mainly based on the logic that the learned policy should lie close to the behavior policy (or data-generating policy). We selected this methods because of its simplicity and solid theoretical foundations, while various names exist in the literature for this kind of methods like behaviour-regularized (Wu et al., 2019) and policy constrained (Levine et al., 2020) that try to bridge the gap between the behaviour policy and the learned policy.

Moving to experiments, we decided to visualize and compare the SPG+TD3 with two variants of SPGR+TD3, testing the discount factor's (γ) effect. All algorithms ran on HPC for 48 hours. The complete list of hyperparameters used for experimentation can be found on table 7.6.

We begin from the Cheetah environment and figure 7.17, where we observe better performance in terms of both test reward and speed for the SPGR variants. It is also apparent that the $\gamma = 0.99$ version exhibited two steep decreases in test reward during training (4.5 and

7.5 million time steps) that can be attributed to the overfitting of the actor. Nevertheless, the agent recovered from the first one, while the second one occurred during the end of training.

Next, we can see the performance plots for the Ant environment on figure 7.18. In this environmet, SPG+TD3 and SPGR+TD3 with $\gamma = 0.95$ performed better yielding more reward on the evaluation phase. The SPGR+TD3($\gamma = 0.99$) did not surpassed the 1000 test reward threshold. The SPG+TD3 is on average ~ 10 fps slower in comparison with the SPGR+TD3 versions.

The next environment is Hopper and the results from experimentation can be found on figure 7.19. This environment was solved only with SPGR+TD3($\gamma = 0.95$) variant, while the other two versions followed the same test reward trend until 6.8 million time steps where the SPG+TD3 exhibited a spiking increase in the test reward.

On figure 7.20 we can find the results for the Walker environment, where both SGPR+TD3 variants outperformed the SPG+TD3 as derived from the test reward plot. The 0.95 version achieves around 1 million time steps very high test reward, followed by an increasing trend till the end of training. The 0.99 version matches the performance of the latter in around 3 million time steps, following similar trend till the end of training. The SPGR+TD3 variants had also higher average speed in comparison with the SPG+TD3 by ~ 10 fps.

Next, on figure 7.21 we have the performance plots for the Humanoid environment. Again, both SPGR+TD3 variants performed better than SPG+TD3, yielding faster reward increase early in the training, with the $\gamma = 0.99$ version having achieved very high test reward in only 2 million time steps, while the SPG+TD3 took almost 4 million time steps to start having a comparable increase. The average speed difference between SPGR+TD3 variants and SPG+TD3 is ~ 8 fps.

The last environment that was used to benchmark our algorithms is the Minotaur and its performance plots can be found on figure 7.22. In this environment, the SPG+TD3 outperformed both SPGR+TD3 variants, scoring very early in the training at around 800k time steps, higher test reward, which was followed by a declining trend till the end of training. The 0.99 variant had the worst performance, not surpassing the ceiling 1 of test reward return. The 0.95 variant exhibited an increasing trend on the test reward which is not enough to fully solve the environment.

In this round of experiments, our aim was two-fold. First, investigate if combating bias overestimation will improve the performance of SPG and second if combating the extrapolation error will improve the performance of the off-policy algorithm SPG. Regarding the first, we found out that SPG+TD3 did not performed so good as the Prioritized SPG on the same environments. Subsequently, combating bias overestimation does not have a positive effect on the SPG's performance. It may be the case that these environments do not leave much of residual TD-error to result into overestimation of the Q-value. Another explanation would be that SPG's exploration scheme is not well suited for a double-critic architectures like TD3. Regarding the second aim, we found out that combating the extrapolation error with a regularization method has a positive effect on the overall performance of SPG, which led us to the development of SPGR+TD3. This new algorithm yielded better test reward, scoring comparable returns with the state-of-the-art off-policy methods.

Parameter	Value
optimizer	Adam
learning rate	0.0001
discount(γ)	[0.95, 0.99]
regularizer constant(a)	2.5
buffer size	1000000
number of hidden layers actor	2
number of hidden units per layer	[256, 256]
number of samples per mini batch	64
number of searches	8
Noise variance σ^2	0.7
σ decay steps	6000000
activation function	ReLU
warm-up steps	50000

Table 7.6: SPGR+TD3 Hyperparameters

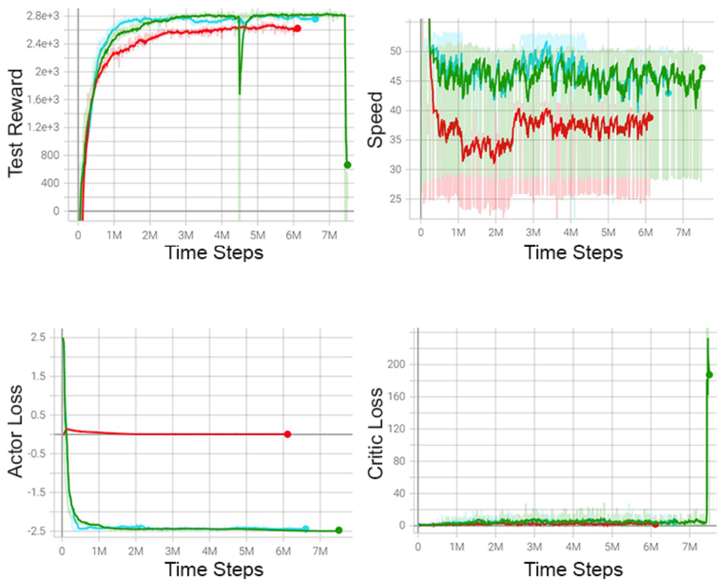


Figure 7.17: Performance plots for the Cheetah environment with ***SPG+TD3****,* ***SPGR+TD3(*** $\gamma = 0.99$***)*** *and* ***SPGR+TD3(*** $\gamma = 0.95$***)****.*

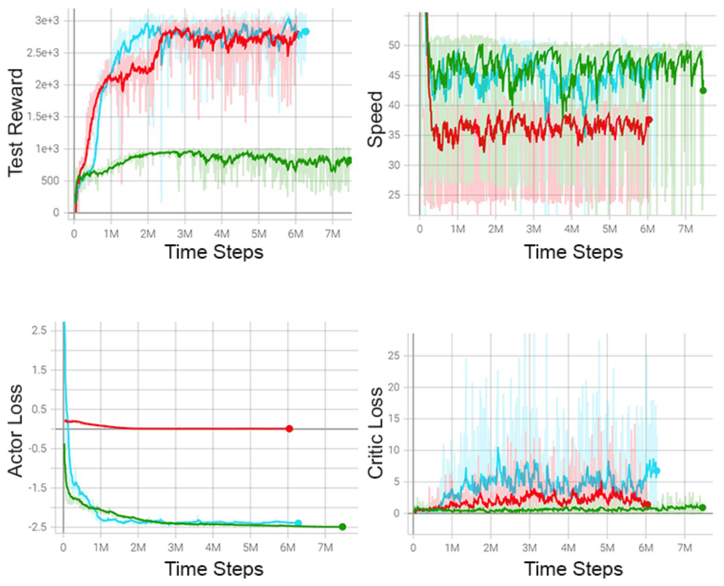


Figure 7.18: Performance plots for the Ant environment with ***SPG+TD3****,* ***SPGR+TD3(*** $\gamma = 0.99$***)*** *and* ***SPGR+TD3(*** $\gamma = 0.95$***)****.*

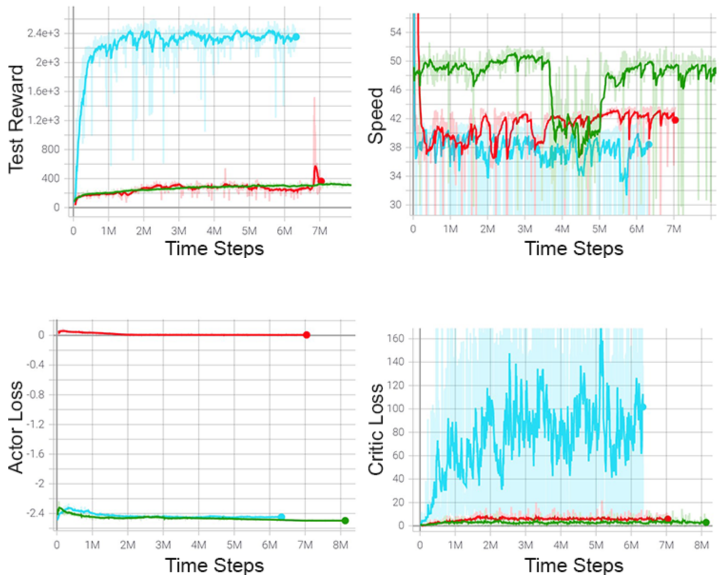


Figure 7.19: Performance plots for the Hopper environment with ***SPG+TD3****,* ***SPGR+TD3(*** $\gamma = 0.99$***)*** *and* ***SPGR+TD3(*** $\gamma = 0.95$***)****.*

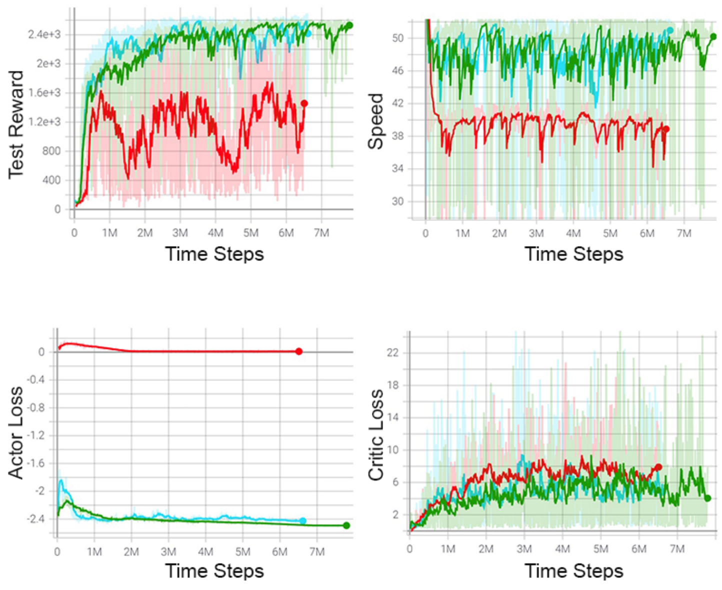


Figure 7.20: Performance plots for the Walker environment with ***SPG+TD3****,* ***SPGR+TD3(*** $\gamma = 0.99$***)*** *and* ***SPGR+TD3(*** $\gamma = 0.95$***)****.*

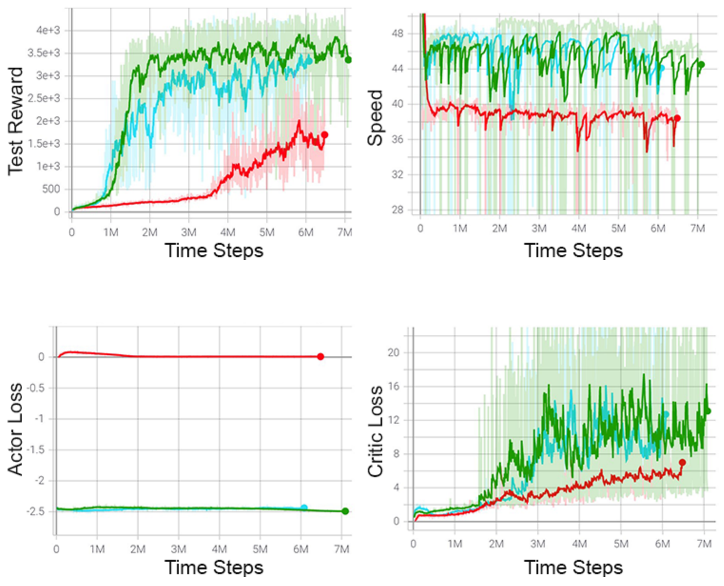


*Figure 7.21: Performance plots for the Humanoid environment with **SPG+TD3**, **SPGR+TD3(** $\gamma = 0.99$**)** and **SPGR+TD3(** $\gamma = 0.95$**)**.*

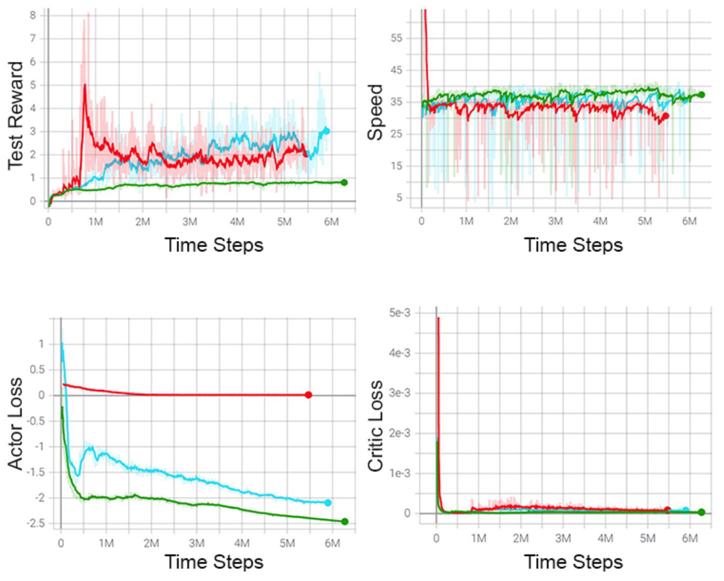


*Figure 7.22: Performance plots for the Minitaur environment with **SPG+TD3**, **SPGR+TD3(**$\gamma = 0.99$**)** and **SPGR+TD3(** $\gamma = 0.95$**)**.*

7.3.3 Performance Study

In this section, we discuss the reward statistics obtained by evaluating the final weights of each algorithm by running the policy 10 times for 1000 time steps. Two algorithms surpassed our expectations and worth mentioning; Prioritized SPG ($\sigma^2 = 0.7$) and SPGR+TD3 ($\gamma = 0.95$). The reward statistics are compared with the most recent ones for PPO and TD3 algorithms, from the work of (Raffin & Stulp, 2020) for Pybullet engine. Two environments, namely the Humanoid and Minotaur were not tested in the above work, thus we provide the results from our own implementation of the algorithms without any significant hyperparameter tuning. Also, it is important to highlight that the comparison's theme is not quantitative increase, rather qualitative improvement and serves as indicator for the potential of our algorithms. The complete list of results can be found on table 7.7. Furthermore, the cost of SPG's optimization, is highlighted on table 7.8.

Algorithm	PPO			DDPG			TD3			SPG			SPGR		
Statistics	mean	max	std	mean	max	std	mean	max	std	mean	max	std	mean	max	std
Environment															
Cheetah	2254			1841	1900	47.89	2678			2807	2854	31.07	**2861**	2873	13.60
Ant	2160			2280	2348	76.74	2865			**3101**	1354	30.47	3091	3188	118.46
Hopper	1622			2015	2076	58.24	2470			2509	2530	12.51	**2598**	2622	22.48
Walker	1238			1627	1674	36.51	2106			2303	2314	8.80	**2705**	2720	12.65
Humanoid	1819	1921	102.37	518	567	37.22	1579	1605	39.21	**4072**	4163	69.78	3975	4460	1311.9
Minitaur	19.48	22.12	2.89	1.45	2.12	0.86	0.45	0.62	0.18	**20.83**	21.553	0.45	6.882	15.29	5.63

Table 7.7: Benchmark statistics between algorithms for the PyBullet environments. We highlight with bold the highest mean reward return.

Algorithm	DDPG	SPG	Percentage Difference (%)
Environment	(fps)		
Cheetah	94.1	52.4	56.92
Ant	92.3	51.8	56.21
Hopper	108.6	58.2	60.43
Walker	104.8	58.5	56.7
Humanoid	101.6	57.9	51.95
Minitaur	67.2	40.1	50.51
Average	94.76	53.15	**55.45**

Table 7.8: Performance cost of SPG optimization for the PyBullet environments.

Starting from SPG algorithm, the results for all the environments look impressive. We focus on the mean return over the runs and can confirm the improvement over our previous experimentation result on table 7.4. We can attribute this increase in performance to the variance hyperparameter tuning that had a positive impact on the exploration process of SPG. It is also worth mentioning that the mean reward results are higher for all environments when compared with the PPO, DDPG and TD3 algorithms.

Moving to the SPGR and the final column of the table, we can observe that the mean reward return exceeded all the other algorithms except SPG, in almost all environments, excluding the Minitaur. The difference on Cheetah and Ant environment is small in comparison with the SPG. On the Hopper environment, SPGR outperforms SPG by less than 100 mean reward but in the Walker environment the difference is larger with SPGR yielding a new high return 2705, 400 mean reward more than SPG. Last, the performance on Humanoid environment is slightly lower than SPG, almost 100 mean reward less but with very large standard deviation. It is still unclear the full explanation behind this result, as the weights obtained from SPGR represented higher reward returns but in the evaluation phase, one run crumbled the statistics. The last environment, namely Minitaur was not properly solved as can be inferred from the low mean reward return.

Finally, it is important to highlight the performance cost on fps for the SPG optimization which is on average 55.45% for the 6 PyBullet environments, found on table 7.8 analytically. This cost is tolerable and justifiable when comparing with the reward return gain from the optimization.

Adding to the above discussion, a video compilation for both SPG and SPGR agents performing during various episodes has been created for progress visualization purposes, and can be found on YouTube [1] .

7.4 Safety Results

In this section we will discuss our finding when experimenting with Safety Layer and Safe-SPG. We tried to visualize the most significant results and comment in place.

7.4.1 Safety Layer

In this part, we present the results from our experiments with Safety Layer (SL) and the PointGoal environment. The purpose of these experiments is to highlight if SL can solve the constrained optimization problem, leading to a safe final policy with few constraint violation or low cost. We also explored three γ values to see if this parameter effects the final policy. The performance plots from level 1 can be found on figure 7.23 and for level 2 on figure 7.24.

Starting from the PointGoal-1 where only one constraint is active, we immediately observe that using $\gamma = 0.9$ had the worst performance, in terms of test reward and test cost, which exhibited high magnitude spiky trend. The other two values, followed similar trend regarding the test reward and the test cost. It is important to note here, that the cost may seem to decrease at first sight, a misleading observation. In practice, the agent is mastering the environment and is able to reach the goal faster, by navigating with greater speed, collecting more reward but not avoiding any obstacles. The average speed for $\gamma = [0.9, 0.95]$ is

[1] https://youtu.be/gAhfMA6SuwM

$\sim$ 62 fps, while using $\gamma = 0.99$ had an average speed of $\sim 44 fps$, a behaviour that can be explained due to other processes parallel running on PC when testing this setting.

Continuing to the second more difficult environment named PointGoal-2 and figure 7.24, it is also apparent that the $\gamma = 0.9$ leads to near zero reward return until 150k time steps where also the cost is low. After that point, a steep increase in the test reward takes place, increasing the test cost in parallel. The landscape for the other two γ values is similar, following close trends both for test reward and test cost, with the $\gamma = 0.95$ exhibiting more instances where the cost is smaller. Regarding the average speed, for $\gamma = [0.95, 0.99]$ this is $\sim 33 - 35$ fps while for $\gamma = 0.9$ the average speed decreased, starting from 65 fps and end to 30 fps.

Having conducted a lot more experiments that can be presented here, we reached the conclusion that SL learns suboptimal actions. Also, SL does not learn any safe behaviour as there is no decrease in the cost pattern during training. This method occasionally provides the agent with not feasible actions $\alpha \notin (-1, 1)$, which we are forced to clip, leading to loss of information. Furthermore, SL's neural network rarely depends on the current state and by internally examining the method, we found out that $g(s; w)$ values remain almost constant. The bottom line is that SL does not affect the policy's learning process, rather than preventing the agent to run into dangerous states during exploration. This usually results the agent stuck into hazardous object corners, not being able to pursue the goal.

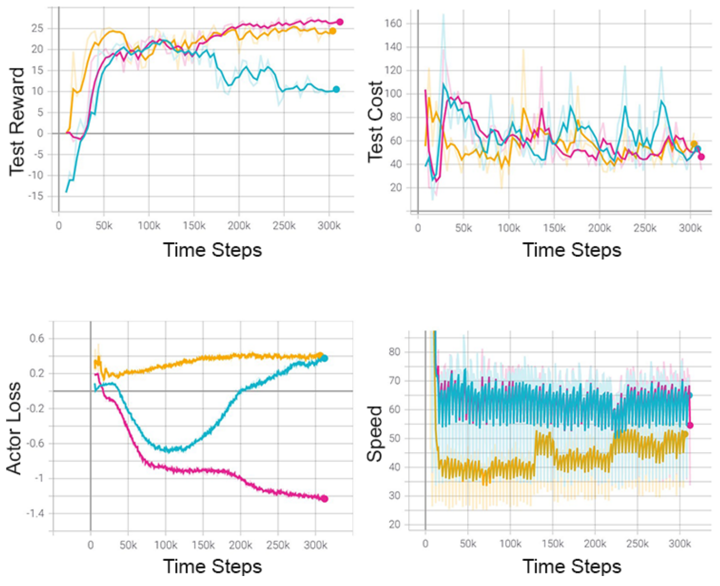


*Figure 7.23: Performance plots for the PointGoal-1 environment with **SL(**$\gamma = 0.99$**)**, **SL(** $\gamma = 0.95$**)** and **SL(** $\gamma = 0.9$**)** for 300k time steps.*

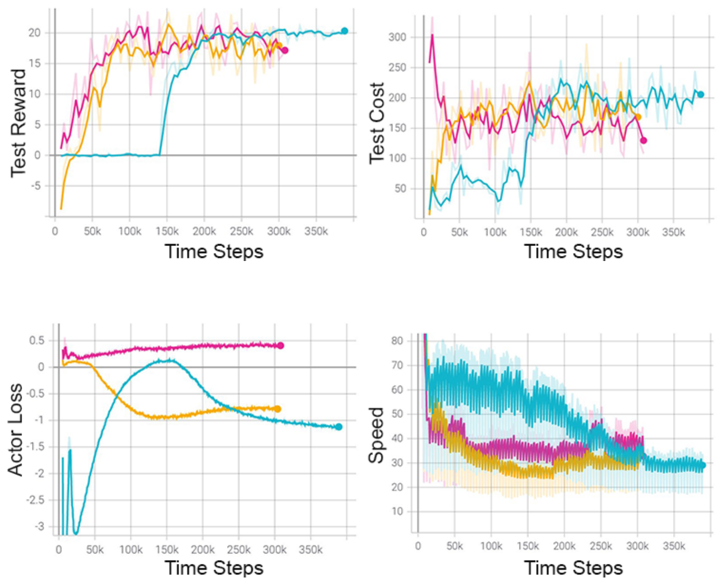


Figure 7.24: Performance plots for the PointGoal-2 environment with ***SL(*** $\gamma = 0.99$***)****,* ***SL(*** $\gamma = 0.95$***)*** *and* ***SL(*** $\gamma = 0.9$***)*** *for 300-400k time steps.*

7.4.2 Safe RL through Bounded Transfer Learning

In this part we present the result from the experiments of PointGoal-1,2 environment, utilizing Safe-SPG as presented on 5.4.2.1 and following the Transfer Learning scheme realized on 5.4.2.2. The number of searches performed is $M = 16$. Three sets of parameters were tested for each stage; for Stage 1 $\gamma_c = [0.4, 0.55, 0.65]$, $\delta_1 = 0.05$ and κ starts from 0.9 and decays to 0.5 through the course of training. For Stage 2, we keep the same γ_c values and the same κ decay, but increase $\delta_1 = 0.09$. We will describe the stages one by one.

In the first stage, we train a classic DDPG agent with pure Q-learning thus the performance plots follow a nominal trend and omitted, reaching the target test reward set at 25, in 500k time steps. This run yields an agile moving agent.

The second stage's performance plots can be found on figure 7.25. For this stage, we also plot the test reward and test cost returns for a Baseline-SPG that does not use Safe Exploration, to highlight the effect of the latter. By inspecting the test reward plot, we spot that all SPG versions exhibit an initial steep decrease in their return, until ∼50k time steps where they recover, following different trends. Only Safe-SPG($\gamma_c = 0.55$) does not recover, exhibiting negative test reward returns until 340k time steps. Safe-SPG($\gamma_c = 0.4$) outperforms the other two variants, reaching high test reward return very early in the training. The same trend is also displayed by the Baseline-SPG. Moving to test cost plot, we immediately spot that Baseline-SPG's Offline Gaussian Exploration does not offer any cost reduction, a natural result in contrast with the Offline Gaussian Safe Exploration of Safe-SPG that decreases the test cost return for all its variants. More specific, Safe-SPG($\gamma_c = 0.4$) performs far better than the other two variants, yielding better final test reward with low test cost. Moving on the last plot from first row of the plot, we observe the speed performance of Safe-SPG which is very analogous between the three γ_c variants. We indicate the average speed performance per variant to be for $\gamma_c = [0.4, 0.55, 0.65]$ 42, 43 and 46 fps respectively. On the second row of the figure we examine the actor's loss, which also reflects the increase in the Q value on the next plot. Lastly, we can observe the Q^{cost}-value plot over the course of training, that confirms the effect of γ_c value. In this stage, we start saving the weights of both actor and cost critic after 400k time steps. This concludes the presentation of the stage 1.

The third stage's performance plots can be found on figure 7.26. First, we immediately notice that Safe-SPG($\gamma_c = 0.55$) suffers a great test reward decrease early in training until 150k time steps. A similar trend follows Safe-SPG($\gamma_c = 0.4$) that recovers few time steps earlier. The Safe-SPG($\gamma = 0.65$) exhibits a more stable test reward trend till the end of training. We believe that this drop stems from non-smooth adaptation of the agent's weights from stage 1 to new experience gathered at stage 2 and was also present in the previous stage. Next, observing the test cost plot which reflects the effect of Offline Gaussian Safe Exploration, we can confirm that Safe-SPG($\gamma = 0.65$) yields the lowest test cost return followed by Safe-SPG($\gamma_c = 0.4$) with a decreasing trend. It is important to highlight that both test reward and test cost are decoupled, signaling the effectiveness of Safe Exploration. In other experimentation runs, we observed that these two values followed similar trend, that is when the reward increased, the cost also increased because the agent have learned to reach faster the target goal without caring for the costs. In final right plot and first row of the same figure, we inspect the speed plots from this training run, where Safe-SPG γ_c-variants performed similarly, except for the $\gamma_c = 0.65$ version that exhibited a small increase in speed after 400k time steps which is attributed to other parallel processes finished in PC during that time. The average speed for all three variants is ~ 26 fps which is a big degradation in comparison with the previous stage. On the bottom row we look at the actor's loss exhibiting a decreasing trend while the Q-value on the next plot increases. The final plot informs us about the magnitude of the Q^{cost}-value that is higher for the Safe-SPG($\gamma_c = 0.65$) reaching a value of ~ 0.3 at the end of the training run. Safe-SPG($\gamma_c = 0.55$) exhibits a bit higher but similar trend with Safe-SPG($\gamma_c = 0.4$). The lowest value is observed for Safe-SPG($\gamma = 0.4$), which is natural. It is important to note that we save actor's weights after 400k time steps, giving enough time for the agent to meet the constraints placed. In general, we suggest saving the final safe policy's weights, after at least 80$\%$ of the predefined time steps length. This concludes the presentation of the stage 2.

Lastly, we present a lengthier version of Stage 2 training for 1 million time steps and make a qualitative comparison with the results from (Ray et al., 2019) for the PointGoal-2 environment on figure 7.27. We inspect the test reward and test cost returns from stage 2 on the top row that do not follow the same trend mainly due to different γ_c value. As we discussed in the previous paragraph, this value plays a key role in the magnitude of Q^{cost}, thus having a big effect on Safe Exploration process 5.11 and the bounds we enforce. Also, it was stated earlier that level 2 environments are more cost-dense subsequently leading to more cost signals in a batch of trajectories. Because of that we found that γ_c needs to have a small value, preferably less than 0.5. On the bottom row we look at the results from Ray et al. and focus mainly on CPO method that exhibits similar cost trend with our Safe-SPG. The training run time from the paper authors is lengthier and the cost measure slightly different making quantitative comparison not possible. With a closer look, we can confirm that Safe-SPG matches and surpasses CPO in terms of test reward and test cost return in only 1 million time steps. The only method that is able to partially solve this environment and satisfy the constraints during training is TRPO-Lagrangian marked with pink colour.

This concludes the Safe-SPG's results that illustrated the ability of Safe Exploration to reduce the cost during training, subsequently the constraint violations, while increasing the reward return. The method as a whole, with the addition of Bounded Transfer Learning scheme was not optimized to the full extend and not all available environments were tested because of the abrupt loss of dr. Wiering that halted the study.

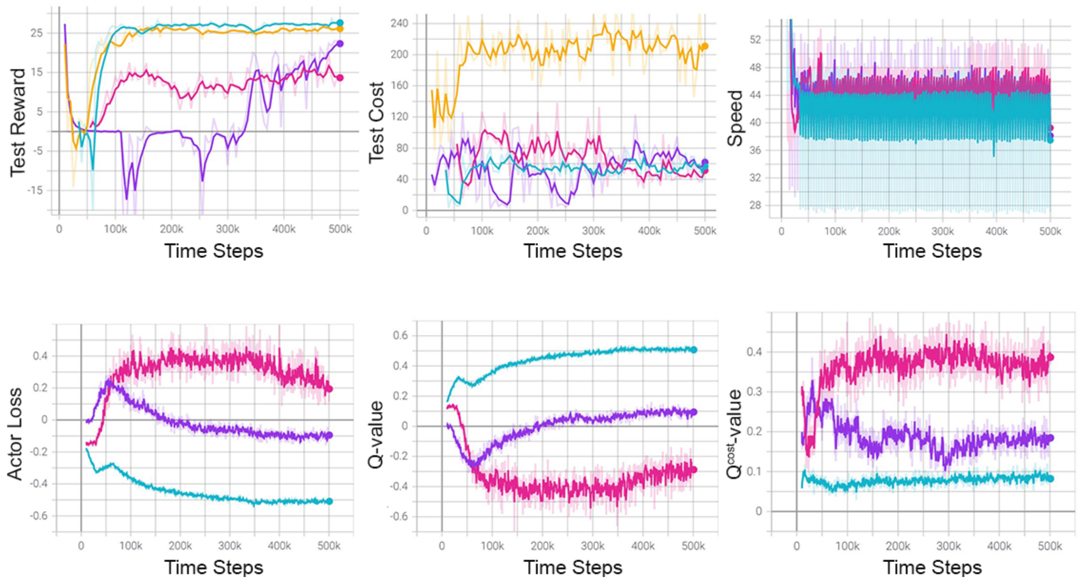


Figure 7.25: Performance plots for the PointGoal-1 environment with ***Safe-SPG($\gamma_c = 0.65$)***, ***Safe-SPG($\gamma_c = 0.55$)*** *and* ***Safe-SPG($\gamma_c = 0.4$)*** *for 500k time steps. We additionally plot a* ***Baseline-SPG*** *version.*

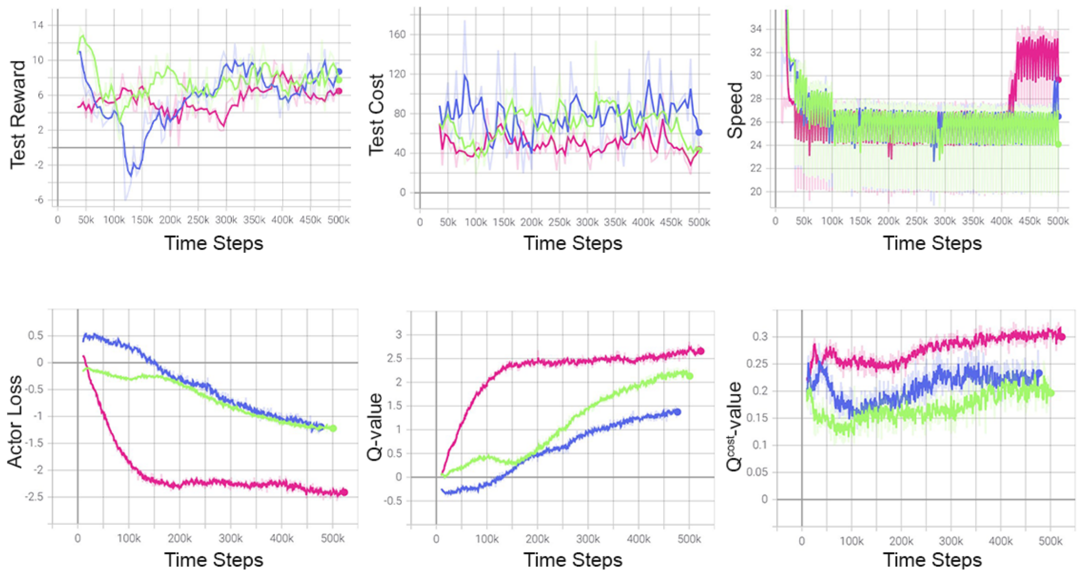


Figure 7.26: Performance plots for the PointGoal-2 environment with ***Safe-SPG($\gamma_c = 0.65$)***, ***Safe-SPG($\gamma_c = 0.4$)*** *and* ***Safe-SPG($\gamma_c = 0.55$)*** *for 500k time steps.*

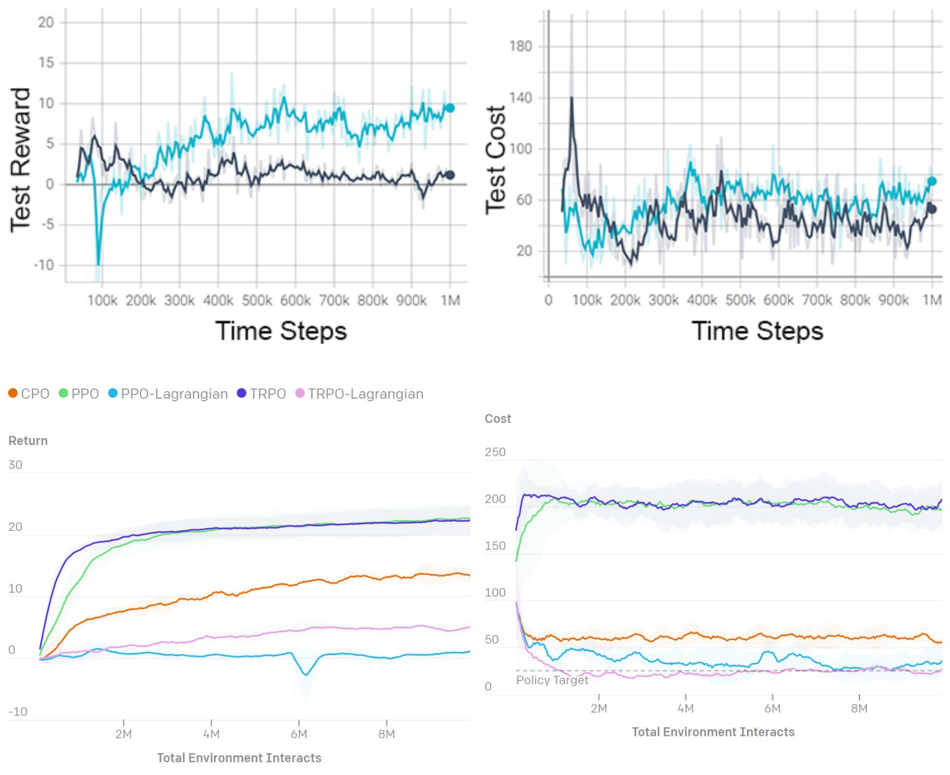


Figure 7.27: Performance plots for the PointGoal-2 environment with ***Safe-SPG($\gamma_c = 0.4$)*** *and* ***Safe-SPG($\gamma_c = 0.65$)*** *for 1 million time steps. The second row illustrates the performance plots from OpenAI's benchmark for the same environment.*

8 Discussion and Concluding Remarks

With chapter 7 we conclude the exploration of Deep Reinforcement Learning for continuous action control. In this chapter, first we answer the research question as stated in section 1.2. Then, future work directions are provided for both policy optimization and constrained policy optimization. Finally, we wrap up this thesis with some concluding remarks.

8.1 Answer to Research Questions

We refresh our research questions that guided this study and provide brief and descriptive answers to them.

1. **How can we increase the performance of deterministic algorithms like SPG?**
   The most effective method stem from our study is prioritized sampling. This had the biggest impact in increasing the final performance of the off-policy SPG as discussed on section 7.2. The second most important factor was the number of searches we performed along with the batch size of trajectories we train our networks on, as presented on section 7.1.

2. **What performance cost must be paid and how hard is it to solve the policy optimization problem?**
   There is always a performance bottleneck when there is optimization involved. SPG and in general off-policy actor-critic algorithms are not the exclusion. The performance cost is mainly coupled with the environment and more specific the engine or suite available for benchmark. The implementation does not play the most important role, although we implemented everything on tensor level, using the most out of a GPU (except parallel gradient calculations). More precise SPG's optimization had on average 55.45% performance cost when compared to DDPG, highlighted on table 7.8.

3. **Does bias overestimation hurts the performance and convergence of SPG?**
   From our experiments, and more specific from section 7.3.2, we found out that combating bias overestimation does not have any positive effect on the performance and convergence properties of SPG. Despite that, we confirmed that combating the extrapolation error and moving closer to the data-generating distribution by Regularizing SPG, leading to SPGR; yielded positive performance increase in most of the environments tested.

4. **How can we achieve Safe RL for continuous action control?**
   There is no universal and unique answer to this question, rather than various schools of thought. One school is working with on-policy methods like PPO, TRPO, CPO and variants that require solving quadratic programs by using linear programming to enforce the constraints. One other school is working with off-policy methods but with stochastic algorithms, thus estimating Gaussian and other distributions. There is also a school of thought that works on Q-functions and how to decouple them from just the reward signal, generalizing their approximation properties to other signals like cost. And finally

what I found particular interesting, was a school of thought that tries to combine Model Predictive Control with off-policy methods, having a hybrid model in mind.

5. **Can we adapt SPG algorithm to incorporate safety constraints? If we can, how competitive is its performance?**
   Yes, we can modify the SPG's exploration to incorporate safety constraints 5.4.2.1, which led us to the development of Safe-SPG that performs Offline Gaussian Safe Exploration to find the best actions in terms of Q-value that satisfy the constraints at that moment. Regarding the second part of the question, it is partially answered as we did not have the time to build other model than Safety Layer for comparison. Still, from the results we had, the performance of Safe-SPG in terms of test reward and test cost return, is far better than Safety Layer and comparable with CPO in just 1 million time steps as can be realized from figure 7.27, while both measures exhibit a promising trend. We note that for comparative and conclusive results, our algorithm should run for at least 4-5 million time steps, matching the exploration time of the other algorithms.

6. **Can we optimize our policies to achieve minimum number of constraint violations? If we can, what performance cost must be paid and how hard is to solve the constrained policy optimization problem?**
   It has been proven in practice mainly from (Ray et al., 2019) who also developed a benchmark to test their hypothesis, that with constrained policy optimization we can achieve minimum number of constraint violations, utilizing the Constrained RL framework and performing Safe Exploration. The cost that needs to be paid is naturally higher than performing a simple policy optimization. This is mostly due to the insertion of constraints that require more diverse and meticulously designed exploration to be implemented in order to yield a safe final policy. From section 7.4.2 and speed results on the figures for both environment levels, comparing them with the speed plots on section 7.1 and quantitative on table 7.1, it becomes clear the magnitude of the performance cost that must be paid. More specific, performing 16 searches on Mujoco environments had an average speed of 73.1 fps, versus the Safety Gym's level 1 of 45 fps and level 2 of 26 average fps per batch of 64 trajectories. So we confirm, solving the constrained policy optimization problem, pose a real challenge both from implementation and computation side. In this work, we tried to solve it by approximating the cost function and then bounding it in a region by setting the appropriate threshold hyperparameters. Although we cannot state that the problem was solved at its full, we provided empirical experimentation results with a novel Transfer Learning framework, utilizing Safe-SPG's exploration.

8.2 Future work

In this section we first give future directions for the off-policy actor-critic algorithm SPG that was in the epicenter of this study. Then we conceptualize the future steps that should be followed in order to extend and validate the safety part of this research and more specific Safe-SPG.

One direction for future work would be to extend and scale the SPG's Offline Gaussian Exploration by increasing the number of searches and batch size. This will require broad experimentation and great computational power. Also it would be beneficial to try incorporate the TD3's exploration mechanism into the SPG's exploration and try to understand why the double Q-learning critic approach did not provide any significant improvements.

Another future work direction would be state normalization in the replay buffer in order to provide the agent with more smooth observations. This idea stems from my research where I noticed that current state-of-the-art algorithms for policy optimization are greatly benefited from it (Raffin & Stulp, 2020). This approach is not computationally expensive nor difficult to implement, thus gets a spot in the todo list.

Adding on all that, a possible future direction would be to perform hyperparameter tuning on SPGR and more specific on the scalar (a) 7.3.2 which was not optimized for our environments, rather was taken from the literature. It would be informative to investigate more the extrapolation error and methods to bridge the gap between the behaviour policy and the learned policy.

Moving to the Safety part, we first need to adapt and get used to work in the Constrained Markov Decision Process framework which is the anchor of Safe RL. Two main branches of algorithms exist in the literature, one transforming the optimization criterion and the other modifying the exploration process. We firmly believe that a successful approach must consider both of these branches and yield a hybrid solution, optimizing both.

As far as transforming the optimization criterion, methods based either on Lyapunov functions (Chow et al., 2018) or on function barriers are the best candidates. Regarding the exploration process modification, a very interesting approach would be the prioritized level replay (Jiang et al., 2021) that also matches the level-based environments available on Safety Gym.

Another direction for future work would be to use reward shaping techniques and try to incorporate the *auxilary* cost signal. This could be done with appropriate adaptive normalization of the cost signal in order to have a meaningful proportion to reward signal. We should note here that signal design is an art on the Reinforcement Learning and comes with many loopholes like *sparse reward* that should be avoided. On the same syllogism, the Q^{cost} function could be fused with the Q-function on the actor's training part with appropriate operators, so as to better direct the safe policy produced by the actor.

A final direction for future work would be the further improvement and optimization of the Bounded Transfer Learning scheme with an addition of an extra layer to both actor and critic, in order to transition better the weights to the next stage, avoiding the steep decrease in test reward return early in the training as discussed on section 7.4.2. A useful addition would be the transfer of the gradients themselves from the previous stage which would require transferring the optimizers weights.

8.3 Conclusion

In this thesis, Deep Reinforcement Learning for continuous action control was explored. During this exploration, experiments were conducted mostly with deterministic off-policy actor-critic algorithms, extending and providing empirical results of the Sample Policy Gradient algorithm whose main idea is to explore the action space more globally, avoiding local optima. The main conclusion of our investigation is that off-policy actor-critic algorithms can increase their performance with Prioritize Sampling and Regularization.

We have further investigated a big issue that demands attention; exploring and developing methods to make it acceptably Safe to embed Reinforcement Agents into physical environments. Towards that end, we introduced the Constrained Markov Decision Process framework, the notion of safety, the constraints and eventually the Constrained Reinforcement Learning with function approximation. In the same theme, we developed Safe-SPG that performs Offline Gaussian Safe Exploration and a novel Bounded Transfer Learning scheme to solve the constrained environments at hand.

Bibliography

Achiam, J. (2018). Openai spinning up. https://spinningup.openai.com/en/latest/user/introduction.html

Achiam, J., Held, D., Tamar, A., & Abbeel, P. (2017). Constrained policy optimization. *International Conference on Machine Learning*, 22–31.

Altman, E. (1999). *Constrained markov decision processes* (Vol. 7). CRC Press.

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., & Mané, D. (2016). Concrete problems in ai safety.

Bastani, H., Drakopoulos, K., Gupta, V., Vlachogiannis, J., Hadjicristodoulou, C., Lagiou, P., Magiorkinis, G., Paraskevis, D., & Tsiodras, S. (2021). Efficient and targeted covid-19 border testing via reinforcement learning. *Nature*, 1–11.

Bellman, R. (1957). *Dynamic programming* (1st ed.). Princeton University Press.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *CoRR*, *abs/1606.01540*. http://arxiv.org/abs/1606.01540

Chow, Y., Nachum, O., Duenez-Guzman, E., & Ghavamzadeh, M. (2018). A lyapunov-based approach to safe reinforcement learning. *arXiv preprint arXiv:1805.07708*.

Christiano, P., Leike, J., Brown, T. B., Martic, M., Legg, S., & Amodei, D. (2017). Deep reinforcement learning from human preferences.

Christiano, P., Shlegeris, B., & Amodei, D. (2018). Supervising strong learners by amplifying weak experts.

Coumans, E., & Bai, Y. (2016–2021). Pybullet, a python module for physics simulation for games, robotics and machine learning.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, *2*(4), 303–314.

Dalal, G., Dvijotham, K., Vecerik, M., Hester, T., Paduraru, C., & Tassa, Y. (2018). Safe exploration in continuous action spaces.

Degris, T., White, M., & Sutton, R. S. (2013). Off-policy actor-critic.

Dulac-Arnold, G., Mankowitz, D., & Hester, T. (2019). Challenges of real-world reinforcement learning.

Eysenbach, B., Gu, S., Ibarz, J., & Levine, S. (2017). Leave no trace: Learning to reset for safe and autonomous reinforcement learning.

Fujimoto, S., & Gu, S. S. (2021). A minimalist approach to offline reinforcement learning.

Fujimoto, S., van Hoof, H., & Meger, D. (2018). Addressing function approximation error in actor-critic methods. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th international conference on machine learning* (pp. 1587–1596). PMLR. https://proceedings.mlr.press/v80/fujimoto18a.html

García, J., Fern, & o Fernández. (2015). A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, *16*(42), 1437–1480. http://jmlr.org/papers/v16/garcia15a.html

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning* [http://www.deeplearningbook.org]. MIT Press.

Gros, S., Zanon, M., & Bemporad, A. (2020). Safe reinforcement learning via projection on a safe set: How to achieve optimality? [21st IFAC World Congress]. *IFAC-PapersOnLine*, *53*(2), 8076–8081. https://doi.org/https://doi.org/10.1016/j.ifacol.2020.12.2276

Hadfield-Menell, D., Russell, S. J., Abbeel, P., & Dragan, A. (2016). Cooperative inverse reinforcement learning. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, & R. Garnett (Eds.), *Advances in neural information processing systems*. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2016/file/c3395dd46c34fa7fd8d729d8cf88b7a8-Paper.pdf

Hans, A., Schneegaß, D., Schäfer, A. M., & Udluft, S. (2008). Safe exploration for reinforcement learning. *ESANN*.

Hasselt, H. (2010). Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, & A. Culotta (Eds.), *Advances in neural information processing systems*. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf

Holubar, M. S., & Wiering, M. A. (2020). Continuous-action reinforcement learning for playing racing games: Comparing spg to ppo. *arXiv preprint arXiv:2001.05270*.

Jiang, M., Grefenstette, E., & Rocktäschel, T. (2021). Prioritized level replay. *International Conference on Machine Learning*, 4940–4950.

Joshua Achiam, A. R.
bibinitperiod D. A. (n.d.). *Safety gym*. https://openai.com/blog/safety-gym/

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Konda, V. R., & Tsitsiklis, J. N. (2000). Actor-critic algorithms. *Advances in neural information processing systems*, 1008–1014.

Leike, J., Krueger, D., Everitt, T., Martic, M., Maini, V., & Legg, S. (2018). Scalable agent alignment via reward modeling: A research direction.

Levine, S., Kumar, A., Tucker, G., & Fu, J. (2020). Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2019). Continuous control with deep reinforcement learning.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, *8*(3-4), 293–321.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Moldovan, T. M., & Abbeel, P. (2012). Safe exploration in markov decision processes.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Pecka, M., & Svoboda, T. (2014). Safe exploration techniques for reinforcement learning – an overview. In J. Hodicky (Ed.), *Modelling and simulation for autonomous systems* (pp. 357–375). Springer International Publishing.

Precup, D. (2000). Eligibility traces for off-policy policy evaluation. *Computer Science Department Faculty Publication Series*, 80.

Raffin, A., & Stulp, F. (2020). Generalized state-dependent exploration for deep reinforcement learning in robotics. *CoRR*, *abs/2005.05719*. https://arxiv.org/abs/2005.05719

Ray, A., Achiam, J., & Amodei, D. (2019). Benchmarking safe exploration in deep reinforcement learning. *arXiv preprint arXiv:1910.01708*, *7*.

Robbins, H., & Monro, S. (1951). A stochastic approximation method. *The annals of mathematical statistics*, 400–407.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533–536.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2017). Trust region policy optimization.

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2018). High-dimensional continuous control using generalized advantage estimation.

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

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). Deterministic policy gradient algorithms. *International conference on machine learning*, 387–395.

Silver, D., Singh, S., Precup, D., & Sutton, R. S. (2021). Reward is enough. *Artificial Intelligence*, 103535. https://doi.org/https://doi.org/10.1016/j.artint.2021.103535

Sutton, R. R. (2019). The bitter lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Todorov, E., Erez, T., & Tassa, Y. (2012). Mujoco: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5026–5033. https://doi.org/10.1109/IROS.2012.6386109

Tsantekidis, A., Passalis, N., Toufa, A.-S., Saitas-Zarkias, K., Chairistanidis, S., & Tefas, A. (2021). Price trailing for financial trading using deep reinforcement learning. *IEEE Transactions on Neural Networks and Learning Systems*, *32*(7), 2837–2846. https://doi.org/10.1109/TNNLS.2020.2997523

Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual*. CreateSpace.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Wiehe, A. O., Ansó, N. S., Drugan, M. M., & Wiering, M. A. (2018). Sampled policy gradient for learning to play the game agar. io. *arXiv preprint arXiv:1809.05763*.

Wiering, M., & Van Otterlo, M. (2012). Reinforcement learning. *Adaptation, learning, and optimization*, *12*(3).

Wu, Y., Tucker, G., & Nachum, O. (2019). Behavior regularized offline reinforcement learning. *arXiv preprint arXiv:1911.11361*.

Yann LeCun, I. M. (2021). Self-supervised learning: The dark matter of intelligence. https://ai.facebook.com/blog/self-supervised-learning-the-dark-matter-of-intelligence

Zahavy, T., Haroush, M., Merlis, N., Mankowitz, D. J., & Mannor, S. (2019). Learn what not to learn: Action elimination with deep reinforcement learning.