DEEP REINFORCEMENT LEARNING FOR HARD EXPLORATION PROBLEMS: LESSONS LEARNT

Stelios Tymvios

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Bachelor of Science

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

June, 2020

APPROVAL PAGE

Bachelor of Science Thesis

DEEP REINFORCEMENT LEARNING FOR HARD EXPLORATION PROBLEMS: LESSONS LEARNT

Presented by

Stelios Tymvios

Research Supervisor _____

Professor Christos Christodoulou

Committee Member _____

Dr Vassilis Vassiliades

University of Cyprus

June, 2020

ABSTRACT

In this thesis, we give a brief introduction to Reinforcement Learning (RL) with an emphasis on reward functions and their consequences, along with short overview of other issues that extend beyond maximizing a function. We give a brief, yet *pragmatic* introduction to Deep Reinforcement Learning (DRL) by reviewing the most influential *model-free, not distributed* and Deep Hierarchical Reinforcement Learning (DHRL) algorithms for both discrete and continuous domains along with some important extensions and exploration schemes. This thesis is focused on two key issues in DRL, correctness of implementation and hard exploration problems.

In DRL, implementation details go well beyond the logic and the structure of the source code. Implementation details such as weight initialization, network dimensions, value magnitudes, activation functions and weight optimization algorithms can have a profound effect on the performance and behavior of an agent. In this thesis, we identify some important silent bugs and undesired behavior that occurs when training a DRL model. In tandem we provide guidelines to assist in solving these issues along with the reasoning and intuition behind the bug and the solution.

The second focal point of this thesis is the class of *hard exploration problems*. This class of problems provides very little information to the model as to how it should perform. In turn, agents are clueless as to how to solve them and how to behave. We put emphasis on problem decomposition as it is inherent to how we learn. We provide examples of such problems from both tabular RL and DRL. We pay special attention to the domain of the Obstacle Tower (OT) as it presents a formidable yet highly decomposable benchmark for current algorithms and present our proposed solution.

Stelios Tymvios – University of Cyprus, 2020

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude towards my supervisor, Professor Christos Christodoulou, for trusting me to undertake this project and enabling me to delve deeper into my interests and his continuous support throughout the project.

I would like to express my most sincere appreciation to my co-supervisor and reviewer, Dr. Vassilis Vassiliades, for his invaluable opinion and thoughts on my ideas, for his excellent guidance, and for broadening my horizons.

Last, but not least, I would like to thank my mother. This thesis is dedicated to you, and your inexhaustible patience with me and my shenanigans over the last year.

TABLE OF CONTENTS

LIST OF FIGURES

Summary of Notation

Capital letters are used for random variables, whereas lower case letters are used for the values of random variables and for scalar functions. Quantities that are required to be real-valued vectors are written in bold and in lower case (even if random variables). Matrices are bold capitals.

$\doteq$	equality relationship that is true by definition
$\approx$	approximately equal
$\propto$	proportional to
$\Pr\{X=x\}$	probability that a random variable X takes on the value x
$X \sim p$	random variable X selected from distribution $p(x) \doteq \Pr\{X=x\}$
$\mathbb{E}[X]$	expectation of a random variable X, i.e., $\mathbb{E}[X] \doteq \sum_x p(x)x$
$\mathbb{R}$	set of real numbers
$\leftarrow$	assignment
$(a, b]$	the real interval between a and b including b but not including a
ϵ	probability of taking a random action in an ε-greedy policy
η	learning rate
γ	discount-rate parameter
λ	decay-rate parameter for Generalized Advantage Estimation and N-step returns
s, s'	states
a	an action
r	a reward

$\mathcal{S}$	set of all nonterminal states
$\mathcal{S}^+$	set of all states, including the terminal state
$\mathcal{R}$	set of all possible rewards, a finite subset of $\mathbb{R}$
$\in$	is an element of; e.g., $s \in \mathcal{S}$, $r \in \mathcal{R}$
$\lvert\mathcal{S}\rvert$	number of elements in set $\mathcal{S}$
t	discrete time step
$T, T(t)$	final time step of an episode, or of the episode including time step t
A_t	action at time t
S_t	state at time t, typically due, stochastically, to S_{t-1} and A_{t-1}
R_t	reward at time t, typically due, stochastically, to S_{t-1} and A_{t-1}
π	policy (decision-making rule)
$\pi(s)$	action taken in state s under *deterministic* policy π
$\pi(a\lvert s)$	probability of taking action a in state s under *stochastic* policy π
G_t	return following time t
h	horizon, the time step one looks up to in a forward view
$G_{t:t+n}, G_{t:h}$	n-step return from $t+1$ to $t+n$, or to h (discounted and corrected)
$p(s', r \mid s, a)$	probability of transition to state s' with reward r, from state s and action a
$p(s' \mid s, a)$	probability of transition to state s', from state s taking action a
$r(s, a)$	expected immediate reward from state s after action a
$r(s, a, s')$	expected immediate reward on transition from s to s' under action a

V, V_t	array estimates of state-value function v_π or v_*
Q, Q_t	array estimates of action-value function q_π or q_*
δ_t	temporal-difference (TD) error at t (a random variable)
$\boldsymbol{\theta}$	parameter vector of target policy
$\boldsymbol{\theta}_k$	parameter vector of target policy at the kth optimization step
$\pi(a\|s;\boldsymbol{\theta})$	probability of taking action a in state s given parameter vector $\boldsymbol{\theta}$
$\pi_{\boldsymbol{\theta}}$	policy corresponding to parameter (vector) $\boldsymbol{\theta}$
$\nabla\pi(a\|s;\boldsymbol{\theta})$	column vector of partial derivatives of $\pi(a\|s;\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$
$J_{\boldsymbol{\theta}}(\cdot)$	Objective of policy $\pi_{\boldsymbol{\theta}}$ evaluated at the input
$\nabla J_{\boldsymbol{\theta}}(\cdot)$	column vector of partial derivatives of $J_{\boldsymbol{\theta}}$ with respect to $\boldsymbol{\theta}$ evaluated at the input
$b(a\|s)$	behavior policy used to select actions while learning about target policy π

Chapter 1

Introduction

Deep Reinforcement Learning is a relatively young field that attempts to solve the curse of dimensionality, through 'deep' function approximators, or, more specifically, Neural Networks. Deep Reinforcement Learning (DRL) has taken the scientific world by storm when it exhibited proficiency in a large number of games in the Arcade Learning Environment (ALE) [1, 2, 3]. Further work showed mastery of GO, a feat thought impossible due to the combinatorial nature of the problem [4]. Since then, the popularity, academic interest, and achievements have only grown. While earlier work used domain knowledge to beat humans in combinatorial problems such as Go, further work surpassed humans not only in Go, but also Shogi, and Chess without any domain knowledge or, in the case of chess, exhaustive search like Stockfish [5]. Parallel work in problems where the rules of the problem are not given has shown performance competitive with humans in the domains of Starcraft 2 [6, 7] and Dota 2 [8], while exhibiting superhuman performance in simpler problems encountered such as those in the ALE [9, 10, 11, 12, 13, 14, 15, 16].

While DRL *works*, there are still a number of open problems holding it back. DRL has its roots in Reinforcement Learning (RL) and owes many of its successes to the theory

behind it. RL is a mathematical formalization of a process, or agent, that interacts with its environment, the environment responds with a new state and a reward, the goal of the process is to maximize the reward it receives. Learning with rewards is a very natural way to learn associations and resembles Operant Conditioning [17, 18]. RL is plagued by the poor sample efficiency, balancing exploration, and the credit assignment problem, while DRL also suffers from worse sample efficiency (relatively, DRL suffers only because the problems it tackles are multiple orders of magnitude greater than that of RL), brittleness with respect to hyper-parameters and initialization, random seeds, generalization and lack of information in imperfect information problems [17, 2, 19, 20, 9, 11, 15, 16, 12, 9, 10, 21].

In DRL, implementation details go well beyond the logic and the structure of the source code. Implementation details such as weight initialization, network dimensions, value magnitudes, activation functions and weight optimization algorithms can have a profound effect on the performance and behavior of an agent [2, 22, 23, 24, 20]. In this thesis, we identify some important, yet silent bugs and undesired behavior that occurs when training a DRL model. In tandem we provide guidelines to assist in solving these issues along with the reasoning and intuition behind the bug and the solution.

The second focal point of this thesis is *Hard Exploration Problems* and how they can be solved through decomposition and temporarily extended actions. Hard Exploration Problems tend to use *uninformative* reward functions and very delayed rewards, thus, the algorithm needs to solve both the exploration and credit assignment problems. Hierarchical and Deep Hierarchical RL (HRL, DHRL) algorithms solve problems by decomposing them into smaller and simpler ones, or by performing composite actions [25, 26, 27, 28, 29, 30]. We draw attention to the domain of the Obstacle Tower (OT) [21] as it presents a formidable benchmark for current algorithms while remaining highly

decomposable. The OT is, by design, a Herculean task for current algorithms as it draws on the current limitations of the field. We present our own potential solution to the problem based on state-of-the-art algorithms and outline several implementation details and decisions along with the reasoning behind them.

Chapter 2

Reinforcement Learning

The problems Reinforcement Learning is concerned with are those where an actor or agent needs to learn how to operate in an unknown environment with some optimal behavior with delayed feedback. To communicate our goals, we provide a reward signal to the agent, and the agent is *usually* tasked with maximizing it.

The reward hypothesis

Simply put, the reward hypothesis states:

> That all of what we mean by goals and purposes can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal (reward). Richard Sutton

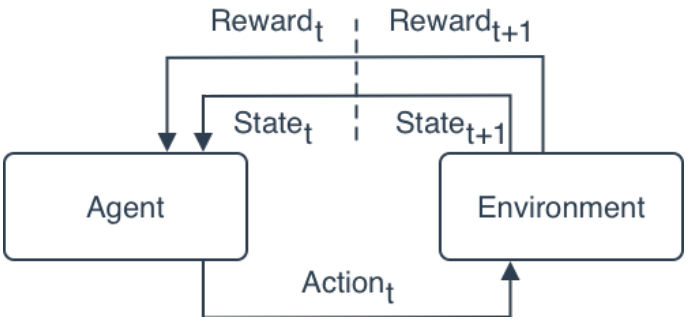

Figure 1: The general framework in Reinforcement Learning. The agent performs some action at time t conditioned on the previous state, and the environment responds with a reward and a new state.

Although a trivial observation, it can not be understated that the reward signal has high implications on the behavior of an agent. It is important that we are explicit with the goal the agent is tasked with solving and refrain from directing an agent towards particular behavior [17]. For example, suppose that we own a paper clip company and we would like to maximize our revenue. Rewarding indirect behavior such as maximizing a quantity like the number of paper clips produced is not necessarily a good proxy for the desired outcome. It is likely that the model will mindlessly create more paper clips [31] when the desired goal is to maximize profit.

It is also important to take into consideration the length, or horizon of the problem we are trying to solve. For example, certain problems are considered episodic as they have a terminal state. An example of such a problem is the game of pong. In episodic problems, the agents are concerned with maximizing their reward over the whole episode. Problems that lack a terminal state or whose terminal state is very far into the future are considered infinite horizon problems. In infinite horizon problems, unlike episodic, the agent can not consider rewards infinitely far into the future because that is infinity. Therefore, to prevent the estimation from exploding into infinity, the agents are concerned with exponentially discounted rewards. The rate of exponential discount is determined by scalar called the discount factor. Smaller values make an agent greedy and myopic with respect to up coming rewards. Larger discount factors force an agent to take into consideration long term rewards, allowing it to make short term sacrifices for long term profit.

2.1 Markov Decision Processes

Markov Decision Processes (MDPs) are the framework we use to formulate a particular problem. MDPs impose certain restrictions on the formulation that in turn, provide us with certain guarantees on convergence and optimal behavior. MDPs allow us to describe the relationship between the environment and an agent operating in it.

More formally, an MDP is the quintuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \gamma, p \rangle$, where $\mathcal{S}$ denotes the state space, or the set of all states; $\mathcal{A}$ denotes the action space, or the set of all possible actions; $\mathcal{R}$ denotes the reward space , or the set of all possible rewards; γ is the discount factor and p defines the *dynamics* of the system. The dynamics define the probability of transitioning to state s' with reward r from state s with action a'. Using the same notation as [17]:

$$p(s', r|s, a) : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$$

$$\sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) = 1, \text{ for all } a \in \mathcal{A}, s \in \mathcal{S} \tag{1}$$

Equivalently, we can define the probability of transitioning from state $s \in \mathcal{S}$ to state $s' \in \mathcal{S}$ using action $a \in \mathcal{A}$ through 1 as:

$$p(s'|s, a) : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$$

$$p(s'|s, a) = \sum_{r \in \mathcal{R}} p(s', r|s, a) \tag{2}$$

At any point t in time, an agent receives a representation $s \in \mathcal{S}$ of the environment and selects an action $a \in \mathcal{A}$, the agent then receives a reward $r \in \mathcal{R}$ and a new representation of the state s', creating a sequence, or a *trajectory*, of the following form:

$$S_1, A_1, R_2, S_2, A_2, ..., R_t, S_t \tag{3}$$

The constraint of MDPs is that the state S needs to have the Markov property. The Markov property requires that any particular state is a sufficient statistic of the future.

In other words, the state encodes enough information to fully describe the distribution of future states. Given the Markov property, we can compute the probability of any finite trajectory:

$$P(\mathcal{T};\pi) \doteq p(S_0)\prod_{i=0}^{T}\pi(A_i|S_i)p(S_{i+1}|S_i,A_i) \tag{4}$$

$$\tag{5}$$

The consequence of the Markov property is that at any point in time $t = T$, regardless of what our behavior was at $t = 0, 1, 2, .., T$, we can make an optimal decision that takes into consideration all of the future states because the current state has enough information about them, thereby reducing the problem into making *informed* sequential decisions. These informed decisions depend on whether the agent needs to find long term or short term policy. The behavior is adjusted through the discount factor $\gamma \in [0, 1]$. Defining the **return** as the sum of the rewards at time step t until the final time step T:

$$\begin{aligned} G_t &= R_t + R_1 + R_2 ... + R_T \\ &= R_t + \sum_{i=t+1}^{T} R_i \qquad (6) \\ &= R_t + G_{t+1} \qquad (7) \end{aligned}$$

We can adjust the behavior of the agent by discounting the future rewards using the discount factor γ:

$$G_t \doteq R_t + \gamma G_{t+1} = \sum_{i=t}^{T}\gamma^{i-t}R_i \tag{8}$$

which results in exponential decay of the value of future rewards. In episodic problems, agents are tasked with finding the best possible behavior for the episode and therefore they need to take into consideration the undiscounted return which is equivalent of using

$\gamma = 1$. Undiscounted rewards cannot be used in infinite horizon problems because the series diverges. Using a discount factor of $\gamma = 0$ turns the agent into a bandit because only the immediate rewards are considered.

As we mention above, the goal is maximize the discounted cumulative reward. The reason we consider discounted rewards is because in infinite horizon problems, and without discounting, adding up the rewards received from a suboptimal policy will still result in infinite rewards because the series diverges, while it converges for any γ in $[0, 1)$:

$$\sum_{i=0}^{\infty} \gamma^i = \frac{1}{1-\gamma} \tag{9}$$

Although MDPs give access to the dynamics of the environment, RL algorithms are concerned with problems that do not provide such information. Algorithms that learn the dynamics through a model are referred to as *model based* algorithms, those that do not, are referred to as *model free* algorithms. The reward function is part of the dynamics of the system, in some applications such as robotics, we may have access to it, i.e. we can query it.

We will use the term *episode* to refer to problems where the agent starts from subspace of the state space and operates until it reaches a terminal state and therefore can not operate. To bridge the gap between episodic problems and infinite problems, one can consider terminal states as those where every action transitions into the same terminal state and the observed reward is always 0.

We will use the term *policy* to refer to the behavior of an agent. More formally, the policy is the function $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that gives the probability that the agent will take action a at state s, thus $\sum_{a \in \mathcal{A}} \pi(a|s) = 1$.

We define the State Value function $V^\pi : \mathcal{S} \to \mathbb{R}$ of policy π as an estimator of the Expected discounted return [8] following the policy π from state s:

$$V^\pi(S) = \mathbb{E}_\pi \left[R|S_t = s\right] \tag{10}$$

$$= \sum_{a \in \mathcal{A}, s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a)\pi(a|s)(r + \gamma V^\pi(s')) \tag{11}$$

Where R is the return. We will use the subscript π in Expectations to denote that we 'follow' the policy π.

We define the State Action Value function $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ as an estimation of the Expected return [8] of taking action a, at state s and *then* following the policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[R|S_t = s, A_t = a\right] \tag{12}$$

$$= r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a)V^\pi(s') \tag{13}$$

$$= r(s, a) + \gamma \sum_{s' \in \mathcal{S}, a' \in \mathcal{A}} p(s'|s, a)\pi(a'|s')Q^\pi(s', a') \tag{14}$$

We also refer to $Q^\pi(s, a)$ as the Q function, and $V^\pi(s)$ as the V function.

Let us now define optimality. The optimal policy π is any policy such that it achieves the maximum discounted cumulative reward from any state. We denote $Q^*(s, a)$ and $V^*(s)$ as the optimal State Action Value and State Value functions respectively.

2.1.1 Partially Observable MDPs

Although MDPs are quite flexible, they come up short when the feedback the environment is insufficient and the Markov property is violated. Such problems are referred to as partially observable or partial information problems. To illustrate the difference, lets juxtapose between a fully observable and a partially observable problem. The former is checkers and the latter is pong. In checkers, one can make an informed decision using just

an image of the board. In pong, a single frame does not contain enough information as it is impossible to deduce the direction of the ball, however, two subsequent frames are sufficient.

Problems like Pong are referred to as *Partially* Observable because the environment does not provide a state, like in MDPs, instead it provides an *observation*. The observation is, in essence, a limited representation of the internal state.

More formally, a POMDP is the sextuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \gamma, p, \Omega, \mathcal{O} \rangle$, where $\mathcal{S}$ denotes the state space, or the set of all states; $\mathcal{A}$ denotes the action space, or the set of all possible actions; $\mathcal{R}$ denotes the reward space , or the set of all possible rewards; γ is the discount factor and p defines the dynamics of the system, just as before, Ω defines the observation space $o \sim \Omega$ and $\mathcal{O}$ is a distribution of the observations conditioned on the current state [32].

A common approach to solving POMDPs is by incorporating a history. This can be done by keeping a history of the observations but this comes at the cost of increased dimensions. Modern algorithms incorporate neural networks with memory cells such as LSTMs and GRUs [33, 34, 35, 32, 12, 36].

2.1.2 Goal conditioned MDPs

Goal conditioned MDPs, or Universal MDPs (UMDP) are a simple extension to vanilla MDPs that reformulates the problem from the finding a policy that maximizes the discounted reward function $r(s, a)$, to finding a policy that maximizes the discounted reward function $r(s, a, s_g)$ over a distribution of goals $s_g \sim \mathcal{G}$ [37, 29, 38, 39]. Therefore a Goal conditioned MDP is the sextuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \gamma, p, \mathcal{G} \rangle$, where $\mathcal{G}$ is the goal space. In consequence, the goal of the agent becomes to find the optimal policy according to:

$$\pi* = \max_{\pi} \mathbb{E}_{\pi} \left[\mathbb{E}_{s_g \sim \mathcal{G}} \left[\sum_{i=0}^{T} \gamma^i r(s_i, a_i, s_g) \right] \right] \tag{15}$$

2.1.3 Options

An important extension to the MDP framework is the idea of high level temporarily extended actions and one of the common formulations of this idea is known as Options[25]. A Markovian Option is the triplet $\langle I_o, \pi_o, \beta_o \rangle$, where I_o is the initiation set, i.e. a set of states from which an option may be used, π_o is a particular option's policy, and β_o is the termination condition [26, 25]. Primitive actions can be generalized to options, where the initiation set is every state, the policy selects the corresponding action, and the termination condition is always true after performing said action [25]. An example of 'options' can be found in figure 2.

2.2 Exploration vs Exploitation

The Exploration Exploitation dilemma is simply asking whether to exploit prior knowledge, or to continue exploration in an attempt to find something better. Ideally, we would like to commit just enough exploration that we reach an optimal policy. In non trivial problems, it is rarely the case that agents have committed sufficient exploration and not fall into a locally optimal policy [40]. In addition, exploration based on successful policies is a source of bias as algorithms converge to a set of solutions [41] and therefore influence the distribution of future scenarios.

Agents that perform sufficient exploration of either the policy can come up with novel strategies that were unknown to humans such as *move 37* in match 2 of AlphaGo [4] vs

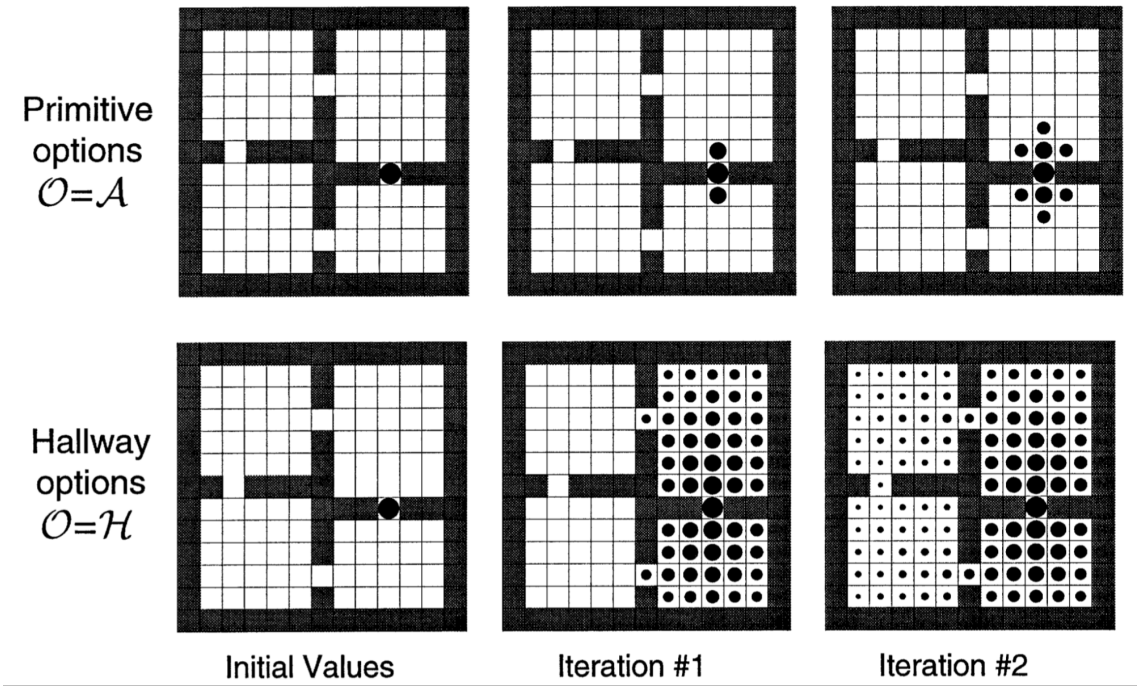

Figure 2: An example of the options framework in the four rooms domain. The agent can perform primitive actions, i.e. move North, South, East and West like in the top figure, or, it can perform a high level action and move to the corridors, like in the bottom figure [25].

Lee Sedol, exploit bugs in environments such as Qbert [42] and Sonic [43], and even exhibit emerging behavior in multi agent systems [41].

To illustrate the necessity for exploration, lets take the simple case of a binary bandit that uses the first pull of each action to estimate the reward, i.e. is biased. The bandit has the following expected rewards for arms $k = 1$ and $k = 2$, where $p(R|k = x)$ denotes the probability of receiving reward R given the selected arm is x. The first arm receives reward 1000 with probability 0.5 and 0 with probability 0.5, the second arm receives reward 1 with probability 1.

$$\begin{aligned}
\mathbb{E}_{k=1} &= 1000 * p(1000|k = 1) + 0 * p(0|k = 1) \\
&= 1000 * 0.5 \\
&= 500 \\
\mathbb{E}_{k=2} &= 1 * p(1|k = 2) \\
&= 1
\end{aligned}$$

Therefore the optimal policy should pick $k = 1$ with probability 1. For a bandit without any form of exploration, the first pull is enough to cause it to always pick $k = 2$.

2.2.1 $\epsilon - greedy$ Exploration

A common exploration strategy is to use an $\epsilon greedy$ behavior policy. An $\epsilon - greedy$ policy acts greedily by selecting $a = \pi(s)$ from a deterministic policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ with probability $1 - \epsilon$, and explores with probability ϵ by sampling $a \sim U(\mathcal{A})$. In $\epsilon - greedy$ policies, the greedy action $\pi(s)$ is selected $1 - \frac{\epsilon(|\mathcal{A}|-1)}{|\mathcal{A}|}$ often and the rest of the actions are selected $\frac{\epsilon}{|\mathcal{A}|}$ often. [17]

Therefore, an $\epsilon - greedy$ policy is:

$$p(\cdot|s) = \begin{cases} \pi(s) & \text{with probability } 1 - \epsilon \\ U(\mathcal{A}) & \text{with probability } \epsilon \end{cases}$$

In practice, $\epsilon - greedy$ policies often begin with a large value for ϵ and slowly anneal to a smaller value. Starting with large exploration and annealing allows a model to build a less biased estimation because it does not bootstrap on lucky prior observations [17].

2.2.2 Softmax Boltzman Distribution

An alternative method of performing both exploration and exploitation is using a stochastic policy $p : \mathcal{S} \times \mathcal{A} \to [0, 1]$ that draws samples from a Boltzman Distribution [17]. Under such policy the probability of selecting an action is as follows:

$$p(a|s; t) = \frac{\exp \frac{q(a|s)}{t}}{\sum_{a'}^{\mathcal{A}} \exp \frac{q(a'|s)}{t}}$$

Where t is the temperature. The temperature performs an adjustment on the values and directly alters the entropy of the policy. Small temperature values cause values to be exaggerated which results in increased probabilities, and thus, lower entropy. Whereas larger temperatures repress the values and thus, the probability of selecting actions with lower values increases.

2.3 Reward Maximization

An agent's behavior, by definition, can only be as good as the reward function allows as its goal is to maximize said reward function. In consequence, achieving optimality with respect to some reward function does not necessarily achieve optimality with respect to

our goals. The misalignment is a consequence of poorly defined reward functions. Besides misalignment, another important issue is poor performance that occurs in *hard exploration problems*, where the signal is sporadic and thus, results in an agent that wonders around aimlessly. Moreover, sporadic reward functions introduce the problem of attributing the reward to particular actions, this is commonly referred to as the reward assignment problem.

Uninformative reward functions present the greatest obstacle in teaching agents how to solve particular problems. Depending on the underlying algorithm, the same reward function can have radically different results based on the information they provide. We would like to put emphasis on the term uninformative. We often consider empty, or zero valued reward signals as uninformative, but that is not always the case. The same reward function can be informative for an agent that performs very frequent updates, or completely useless for an agent that performs sporadic updates [17].

2.4 Beyond Reward Maximization

Although the goal in Reinforcement Learning is to learn the optimal behavior through a reward signal, the process of learning hides many intricacies and nuances that make it difficult to diagnose the exact short-comings of a particular algorithm. This issue is exacerbated in Deep Reinforcement Learning as the algorithms are more volatile and incorporate a lot of moving parts. Regardless, we can decompose and understand an agent by evaluating it in a variety of different environments that allow us to deconstruct its behavior into the following performance evaluations:

- Noise

- Scale
- Memory
- Generalization
- Exploration
- Credit Assignment

It is important to separate the facets of performance in DRL as, unlike other fields of Machine Learning, an agent's behavior and performance dictates the future states it will observe. In consequence, an agent is responsible for its own dataset, this creates an amplifying feedback loop that can cause an agent to fall into behavioral local minima, to forget prior experience, and even to never learn.

Noise

Noise refers to an agent's robustness to noisy reward signals [19]. Noisy reward signals can result in varying behavior depending on the underlying algorithm and require different approaches when optimizing the loss function.

Scale

Scale refers to the robustness of an algorithm in terms of the size of the provided rewards. Although tabular algorithms can learn with rewards in varying scales, DRL algorithms are not as robust due to the methods that we use to train them and because the changes are *not local*. Differences in magnitude can result in very large gradients and gradient updates that cause catastrophic forgetting [44] and numerical instability [2], they

can mess with optimizers such as Adam [35] that keep track of magnitude and last but not least, different optimization algorithms like Adam and SGD can exhibit wildly different results depending on the scale [19].

Memory

Classic RL algorithms assume that the problems they observe is Markov, thus the state provides a sufficient representation of the future. This is in contrast to real world problems as they are usually partially observable problems. We can transform a POMDP into an MDP by constructing a state representation with sufficient information [36, 17]. In classic algorithms, this was done by increasing the state space to account for more observations, in contrast, DRL algorithms usually incorporate a memory augmented neural network that keeps an internal state that is separate from the received observation [36].

Generalization

Generalization refers to the ability to generalize behavior to unseen inputs [19] and is the reason DRL algorithms use neural networks to learn a policy. Tabular methods do not generalize without some compression of the state space that can result in suboptimal behavior [17]. An agent with high generalization capabilities is, by definition, more sample efficient than other agents that achieve the same final performance. In certain cases such as [45], an agent with lower performance on seen examples may be able to generalize better than other agents that showcase better performance on seen examples.

Generalization in RL comes by introducing randomness in the behavior of the model and the observations of the environment; batch normalization; and regularization [46]. Specifically, we make an agent more stochastic by making it $\epsilon - greedy$ which overrides

the actions of an agent, use dropout layers that introduce stochasticity at runtime by disabling neurons, L2 regularization to avoid over-fitting just like in Deep Learning [35, 46], data augmentation to modify the incoming observations [46] and induced invariance by introducing random filters through randomly initialized Convolutional filters [45].

Exploration

Exploration is paramount to learning optimal behavior. Lack of sufficient exploration of the state space may result in agents that may never experience states with high rewards that occur behind states with some penalties [17, 19]. This is very similar to how we humans learn and behave, we encounter the same dilemma in our lives in the form of choosing between entertainment and discipline.

It is important that we separate Exploration and Generalization. The former refers to observing semantically different states, whereas the latter refers to semantically similar states. This is an important distinction that distinguishes some classes of agents with others. Soft agents [47], i.e. agents that use entropy maximization techniques as intrinsic reward, attempt to perform exploration by receiving novel observations. Such agents can fall victim to the Noisy-TV problem, where the agent receives highly entropic observations and thus high intrinsic reward and ignores the extrinsic reward [48]. We can circumvent this limitation through other schemes such as structured exploration [9], observation distillation [48], noisy networks [49], or action noise [50, 51].

Credit Assignment

Credit Assignment refers to the ability to assign credit to the correct action at the correct time [19]. Reinforcement Learning is an extension to contextual bandits in that the

agent's actions *directly* affect the distribution of future states, hence, the significance and emphasis on obtaining the Markov property. Another issue with Credit Assignment is the existence delayed rewards which are even more exacerbated in sparse reward environments [17, 21, 38].

Chapter 3

Deep Learning

3.1 Optimization

While 'learning' is often thought of as synonymous to optimization, the reality is far from it. Optimization is used to achieve learning in a neural network or other *differentiable* methods [35]. We can think of neural networks as a function $F(X, \boldsymbol{\theta})$, where the neural network defines the operations in F, and the weights, $\boldsymbol{\theta}$, are the *learnable* parameters of the function. The goal of 'learning', is to minimize a differentiable error function that includes F, and a training dataset X, and hoping that the model generalizes to a new dataset X' that is sampled from the same distribution as X [35].

3.1.1 Gradient Descent

Gradient Descent is a very simple optimization procedure that iteratively updates the parameters of a differentiable function. In our case, gradient descent simply computes the gradient of a weight vector, usually denoted with $\boldsymbol{\theta}$, with respect to some cost function that needs to be minimized. The gradient of a function gives us the direction of the steepest ascent, by taking the negative of that function, we go towards the opposite direction [35].

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} E(F(X; \boldsymbol{\theta}), Y) \tag{16}$$

Where $\boldsymbol{\theta}$ are the parameters of the differentiable function, $E(Y, \hat{Y})$ is a *differentiable* cost function between the predicted value $\hat{Y}$ and the true value Y that needs to be minimized, and η scales the size of the step.

We take the liberty of treating the weights of a neural network as *a single vector* since we can select individual elements from the weight vector through matrix multiplication. This approach provides a very useful abstraction that hides the complexity of the underlying model and focuses on the important content [35].

3.1.2 SGD

It is often physically impossible to compute the gradient with respect to the whole dataset due to hardware, and time limitations, an alternative approach is to compute the gradient for a single example and update the weights with respect to that, this method is called Stochastic Gradient Descent (SGD) [35].

Due to the high variance of using a single example to compute the gradient, SGD may perform steps that are destructive. The first remedy is to use a minibatch, i.e. more than one example. Minibatches essentially compute the gradient for all examples and then perform a reduction. The reduction operation can be either summation, or averaging. In the case of summation, we need to reduce the step size accordingly to avoid too large steps. The second remedy is to use ‘momentum‘. The momentum is, in essence, an exponential recency moving average of the gradients [35].

$$V_0 = \mathbf{0}$$

$$V_t = \beta V_{t-1} + (1-\beta)\nabla_{\boldsymbol{\theta}} E(F(X;\boldsymbol{\theta}), Y) \tag{17}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta V_t \tag{18}$$

Where V_t is the momentum at time t, and β adjusts the speed of the updates.

3.1.3 RMSProp

Root Mean Square Propagation (RMSProp) is a modification on the momentum. Momentum estimates the first moment of the gradient, RMSProp is a second moment method. It computes of the magnitude of the weights, by keeping track of the squared gradients [35].

$$V_0 = \mathbf{0}$$

$$V_t = \beta V_{t-1} + (1-\beta)(\nabla_{\boldsymbol{\theta}} E(F(X;\boldsymbol{\theta}), Y))^2 \tag{19}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\eta}{\sqrt{V_t}} \odot \nabla_{\boldsymbol{\theta}} E(F(X;\boldsymbol{\theta}), Y) \tag{20}$$

We perform an element wise multiplication that results in weights mostly around 1, then we account for that through the learning rate. The effect of RMSProp is that we gain more robust updates because the weights are neither too small or too large and by multiplying with the gradient vector, we know the correct direction to change the weights [35].

3.1.4 Adam

Adaptive Moment Estimation (Adam) combines RMSProp and Momentum, along with a de-biasing mechanism for the early stages. De-biasing is used because in the early stages

are by definition biased because we have very poor estimate of the actual moments [52, 35].

$$
\begin{aligned}
M_0 &= \mathbf{0} \\
M_t &= \frac{\beta M_{t-1} + (1-\beta)\nabla_{\boldsymbol{\theta}} E(F(X;\boldsymbol{\theta}), Y)}{1-b_1} \\
V_0 &= \mathbf{0} \\
V_t &= \frac{\beta V_{t-1} + (1-\beta)(\nabla_{\boldsymbol{\theta}} E(F(X;\boldsymbol{\theta}), Y))^2}{1-b_2} \\
\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \frac{\eta}{\sqrt{V_t}} \odot M_t
\end{aligned} \tag{21}
$$

where b_1, b_2 are the de-biasing factors, M denotes the momentum from SGD, and V is the second moment from RMSProp [52, 35].

3.2 Training Techniques

3.2.1 Stochastic Computation Graphs

A common pattern when training neural networks and DRL models in particular is that of estimating a distribution. This arises in two forms, Probability Density Functions (PDF) and Probability Mass Functions (PMF). The issue that arises is that sampling is *not a differentiable* operation and thus breaks end-to-end training. We can go around this issue through two ways, the reparameterization trick [53], and estimating probabilities.

The reparameterization trick is used when we compute a PDF. The idea is that we can separate the sampling process from the estimation of the distribution by

1. Sampling z from $\mathcal{N}(0, I)$, and
2. Skewing z

This two step process allows us to estimate the distribution parameters, and then map the noise to that distribution [53, 35].

Estimating probabilities is done by computing values for the different outputs of the model, apply the softmax operator, and then sample from the resulting categorical distribution [53, 35].

3.2.2 Penalty Annealing

A common occurrence in DL is to train for multiple, or composite objective functions. One of the issues that arise with composite objectives is that an auxiliary task may end up dominating the loss function and in consequence destroy cause the network to collapse or cause numerical instability that results in infinities.

A common approach is to slowly introduce the penalties through annealing, i.e. multiplying the penalty with a factor that grows from 0 up to 1. This approach is particularly useful when training a VAE network as it allows the decoder model to learn to produce good reconstructions before imposing the D_{KL} penalty from eq. 24 [54, 55].

3.2.3 Weight Regularization

Weight Regularization penalizes the weights the network based on the norm of its weights, and in consequence constrains the network to smaller values. The penalty is usually in terms of L1 and L2 norms. Suppose that we have some arbitrary cost function $J(\cdot;\boldsymbol{\theta})$, the new cost function becomes:

$$J_{reg}(\cdot;\boldsymbol{\theta}) = J(\cdot;\boldsymbol{\theta}) + \lambda\|\boldsymbol{\theta}\|_n^n \tag{22}$$

Where n is the norm, i.e. L1, L2.

3.2.4 Independent and Identically Distributed

When training neural network, we train on a training set X and hope that the model generalizes to a different set, X' that is sampled using the same process as X.

We generally assume that our samples are identically distributed, this means that any two examples have the same probability of occurring. Identical distribution is important because it provides a gateway towards generalization. If the sampling process does not result in identically distributed samples, then the probability of finding a subset of samples is greater than the rest. In consequence, there is an incentive to learn how to solve inputs that occur more often.

In addition, we want the items from the batch that we use to train the model are independent, this means that the probability of observing a particular input is dependent on the probability of observing a different input. This, again, hinders generalization.

When a dataset has both Independent and Identically Distributed (I.I.D) elements, we say that it has the I.I.D property.

3.3 Self-Supervised Learning

Unsupervised Learning as a field of ML is concerned with learning the underlying structure of the data using models to help researchers understand patterns in the data, or perform some form of dimensionality reduction. In modern literature, Unsupervised Learning serves auxiliary tasks for models, and agents that help in optimizing the main task. In this context, Unsupervised models are referred to as *Self-Supervised* models. In DRL, Self-Supervised Learning is used to learn observation representations through AutoEncoder[35] models, state representations through autoregressive models, and world dynamics through forward predictive models.

3.3.1 AutoEncoders

An AutoEncoder (AE) [35, 53] learns an identity function $F : R^n \rightarrow R^n$. Identity functions on their own are not interesting, however, the goal of an AE is to learn an identity function through an information bottleneck. We define an AE as the composite function $F(x; \boldsymbol{\theta}, \boldsymbol{\phi}) : R^n \rightarrow R^n = D \circ E$, where $E(x; \boldsymbol{\theta}) : R^n \rightarrow R^z$, $D(z; \phi) : R^z \rightarrow R^n$ and $R^z \neq R^n$. The intuition behind autoencoders is that we can learn a mapping or an encoding function, $E(\cdot; \boldsymbol{\theta})$ that transforms a higher dimensional space R^n to a lower dimensional space R^z, and another mapping or in this case a decoding function, $D(\cdot; \phi)$ that transforms the lower dimensional space R^z back to the higher dimensional space R^n. Learning these mappings is possible because we store the information of the transformations into the parameters $\boldsymbol{\theta}$ and ϕ. In short, an AE is a composite function $F(x; \boldsymbol{\theta}, \phi)$ that learns to minimize:

$$\begin{aligned} J(x; \boldsymbol{\theta}, \phi) &= \mathbb{E}\left[(X - F(X))^2\right] \\ &= \mathbb{E}\left[(X - D(E(X; \boldsymbol{\theta}); \phi))^2\right] \end{aligned} \tag{23}$$

which results in learning two individual functions, $D(\cdot; \phi)$ and $E(\cdot; \boldsymbol{\theta})$. In general, AEs are used to construct an informative representation of the input and perform some sort of dimensionality reduction.

3.3.2 Variational AutoEncoders

Variational AutoEncoders (VAEs) [35, 53, 56] provide a Bayesian approach to learning an informative latent variable by assuming that the input is a transformation of a Multivariate Gaussian distribution that is sampled from and that the dataset exhibits the iid (independent and identically distributed) property. Similarly to AEs, VAEs learn an encoder and a decoder, however, VAEs are stochastic models whereas AEs are not.

The goal of a VAE is to learn to produce the original input with high probability by encoding it into a distribution instead of a latent variable [53, 56]. This allows us to impose certain restrictions on the distribution which we can not do in a sensible way on regular AEs. VAEs work by assuming that the input is the result of sampling from a Multivariate Gaussian distribution, the prior $p(x)$. The encoder model learns to map the input into the distribution $q(x)$ and is penalized based on the KL Divergence between the prior and the learnt distributions [56, 53]:

$$L_{distr} = D_{KL}(q(x)||p(x)) \tag{24}$$

The decoder model works by sampling from $q(x)$ and reconstructing an image with the goal to maximize the probability that the given input will be created:

$$L_{rec} = \mathbb{E}_{x\sim data}\left[\mathbb{E}_{z\sim q(x)} \log p(x|z)\right] \tag{25}$$

which results in the following objective function:

$$\begin{aligned} J &= -\beta L_{distr} + L_{rec} \\ &= -\beta D_{KL}(q(x)||p(x)) + \mathbb{E}_{x\sim data}\mathbb{E}_{z\sim q(x)} \log d(x|z) \end{aligned} \tag{26}$$

Where $\beta \in (0, 1]$ is a scaling factor, z is some latent sampled from the learnt distribution $q(x)$ and $p(x)$ is the prior that defines the underlying distribution which is assumed to be a Multivariate Gaussian and $d(x)$ is the decoder function. It is important to note that sampling in of itself is a *non-differentiable* operation. In consequence, we can not sample directly from $q(x)$ however, we use the *reparameterization* trick to handle this issue (see 3.2.1). An extension on VAEs, the β-VAE [56] suggests that extending the scaling factor, β, to penalties greater than 1 produces disentangled representations that are easier to interpret over the representations created by a VAE or an AE.

3.3.3 MMD-VAE

Maximum Mean Discrepancy Variational AutoEncoder (MMD-VAE) builds on Variational AutoEncoders by identifying and solving two issues, a) the latent variable that is forwarded from the encoder model is not used during decoding, b) the model overestimates the variance of latent code and pushes it to infinity. MMD fixes these issues by penalizing the distance between the transformation of a random variable and two kernels, one constructed by the learnt variable and a gaussian one [57, 58].

Chapter 4

DQN and Friends

4.1 DQN

Deep Q-Networks(DQN) [2] not only showed that Neural Networks are capable of solving high dimensional problems through raw pixels, it also established a set of core practices and created the era of *Deep Reinforcement Learning*. DQN was tested on the Atari benchmark [1] suite where it managed to learn from raw pixels how to play classic atari games.

DQN is a spiritual successor to the ubiquitous Q-learning [17], however, it provides some significant changes that enable learning how to behave in high dimensional environments using neural networks as function approximators.

The main ingredients of DQN's success are the usage of an off-policy algorithm, the usage of an Experience Replay (ER) buffer that was sampled periodically, the usage of target weights, and the usage of offline updates.

4.1.1 Experience Replay

The usage of Experience Replay is not new, however, it is only possible because DQN is an off-policy algorithm. Off-policy algorithms can use experience from other agents in order to optimize their own behavior, just like Q-learning. The Experience Replay is a buffer that contains the N most recent experiences and is sampled uniformly and periodically.

The inclusion of ER allows the agent to sample transitions that it would have otherwise 'forgot'. Forgetting transitions occurs because in RL the distribution of future states is dependent on the policy. Thus, as the policy of an agent improves, the distribution of observed states changes. Uniform sampling allows the agent to avoid catastrophic forgetting by ensuring that all states have a non zero sampling probability. In addition, it de-correlates the updates.

4.1.2 Offline updates

Due to the nature of neural networks, updating on every step, like in Q-learning is expensive, susceptible to noise, and catastrophic forgetting. In [2], DQN is updated every X number of steps with a number of transitions. The increased interval between updates allows the network to remain stable and avoid catastrophic updates to the policy. In addition, larger number of transitions allow for better and more stable estimations of the gradient, thus the neural network is able to learn.

4.1.3 Target Weights

DQN keeps track of two sets of weights, one set is the policy weights $\boldsymbol{\theta}$, and the second is the target weights $\boldsymbol{\theta}'$. DQN periodically copies the policy weights to the target weights.

The target weights are used to estimate the target values for the policy network. The inclusion of target weights avoids the network bootstrapping off its own overestimations for the state-action values. This change removes bias from the state-action value estimation and keeps the policy stable.

4.1.4 Optimization Procedure

DQN works by treating the learning process as a regression problem, where given the current state s the network needs to estimate a target that is created using the target weights. More specifically, the optimization objective in DQN is:

$$\delta_t = r + (1-d)\gamma \max_{a'} Q(s_{t+1}, a'; \boldsymbol{\theta}') - Q(s_t, a_t; \boldsymbol{\theta})$$
$$J_{dqn}(\boldsymbol{\theta}, \boldsymbol{\theta}') = \mathbb{E}[\delta_t^2] \tag{27}$$

and the overall algorithm is available in 4.1.4.

The optimization objective forces the policy weights to learn to estimate the reward in addition to the estimation of the next state value, which enables, the network to reduce the δ_t error.

4.2 Double DQN

Double Q-learning uses two Q-functions a means of reducing the overestimation that occurs in particular scenarios [17, 59]. Double DQN is an adaptation of the Double Q-learning approach specifically for DQN. More specifically, it adapts the optimization objective to select an action using the policy weights, the value of the action is estimated

Algorithm 1 DQN

Initialize $\mathcal{D} \leftarrow \{\}$
Initialize weights $\boldsymbol{\theta}$
Initialize $\epsilon - greedy$ policy $\pi_{\boldsymbol{\theta}}$
$\boldsymbol{\theta'} \leftarrow \boldsymbol{\theta}$
$s \sim \mathcal{S}^0$
for step in $0..T$ **do**
    $a \sim \pi(s)$
    take action a
    Observe s', r, d ▷ next state, reward, episode termination
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle s, a, r, s', d \rangle\}$
    $s \leftarrow s'$
    if $d == True$ **then**
        $s \sim \mathcal{S}^0$
    end if
    if is time to update weights **then**
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla_{\boldsymbol{\theta}} J_{dqn}(\boldsymbol{\theta}, \boldsymbol{\theta'})$
    end if
    if is time to update target weights **then**
        $\boldsymbol{\theta'} \leftarrow \boldsymbol{\theta}$
    end if
end for

using the target weights [59]. It alters the computation of the δ_t to:

$$\delta_t = r + (1-d)\gamma Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \boldsymbol{\theta}); \boldsymbol{\theta'}) - Q(s_t, a_t; \boldsymbol{\theta}) \tag{28}$$

$$J_{dqn-double}(\boldsymbol{\theta}, \boldsymbol{\theta'}) = \mathbb{E}[\delta_t^2] \tag{29}$$

4.3 Dueling DQN

Dueling DQN introduces a dueling head architecture that enables a network to decompose the $Q(s, a)$ estimation to

$$Q(s, a) = V(s) + (A(s, a) - \max_{a'} A(s, a')) \tag{30}$$

and

$$Q(s, a) = V(s) + (A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s, a')) \tag{31}$$

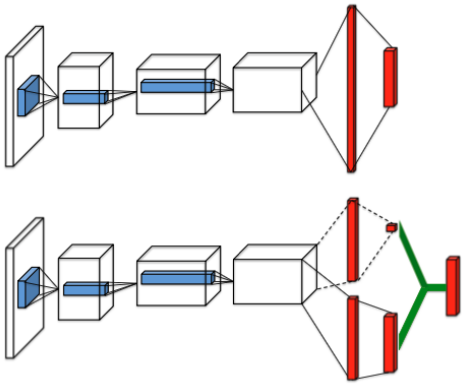

Figure 3: The DQN architecture (**top**) and the Dueling DQN architecture(**bottom**) [60]. Both networks estimate $Q^{\pi}(s, a)$ at the final layer. The green component implements one of [30], [31]. Source [60].

The reasoning behind this estimation is that the neural networks that we use to estimate these values produce an $|\mathcal{A}|$ dimensional vector with the state-action values for every action for a particular input state. Decomposing $Q(s, a)$ like above shows us that $V(s)$ is estimated for every action, thus, we can create a module with two internal output heads, one to estimate $V(s)$ and one to estimate $A(s, a)$. When we sum the outputs of the modules, $V(s)$ is projected, thus, the value is added to every action. This change enables the networks to learn significantly faster because $V(s)$ does not need to be learned over and over again for every state-action pair [59].[1]

4.4 DRQN

Deep Recurrent Q-Network (DRQN) is a first attempt in using RNNs, more specifically LSTMs [33], in DQN. A number of problems are not fully observable. Such problems are referred to as 'Partially Observable' problems and need a means of transforming the observations into informative states before they can be solved. DRQN accomplishes this

[1]The keen eye will notice that neither function is the 'correct' advantage function. The explanation is listed in the appendix.

transformation by incorporating an LSTM cell before the fully connected part of the neural network. The introduction of the LSTM cell allows the network to remember information across different time-steps.

The introduction of the LSTM layer allows an agent to learn a proper mapping of sequences of observations into the state-space and enables the agent to solve some POMDPs. In [36] the authors also identify the issue of initializing the hidden state when training a recurrent neural network and provide two distinct mechanisms to perform the Bellman backup updates with back-propagation through time.

Besides a significant improvement in POMDPs, DRQN exhibits better performance in some fully observable problems, however, it also exhibited significant differences depending on the location of the LSTM layer. Both DRQN and DQN were tested on Arcade Learning Environment (ALE) [1]. ALE provides observations in the form of images, thus, the agents require a Convolutional Neural Network [35]. Both agents used the same network architecture, except DRQN used an LSTM layer in-place of the first fully connected layer right after the last convolution layer. In [36], DRQN exhibits severe performance degradation depending on the location of the LSTM layer, and the activation function that follows.

4.4.1 Bootstrapped Sequential Updates

The first proposed update mechanism to the initialization problem is to sample episodes from an Experience Replay. The updates happen from the beginning of an episode up to the end. This allows the RNN to retain the hidden/context state throughout the episode and the sample algorithm is available in 4.4.1.

Algorithm 2 Bootstrapped Sequential Updates

Require: $\mathcal{D}$ ▷ Trajectory Experience Replay
Require: $N > 1$ ▷ Number of trajectories to sample
Require: $\boldsymbol{\theta}$ ▷ Weights
Require: $\boldsymbol{\theta}'$ ▷ Target weights
Require: $\lambda > 0$ ▷ learning rate

$BellmanError \leftarrow 0$
for i in 0..N **do**
    h = **0**
    h' = **0**
    trajectory $\sim D$
    for (s, a, r, s') in trajectory **do**
        $q, h \leftarrow Q(s, a, h; \boldsymbol{\theta})$
        $y, h' \leftarrow \max_{a'} Q(s', a', h'; \boldsymbol{\theta}')$
        $y \leftarrow \gamma(1 - d(s'))y + r$
        $BellmanError \leftarrow BellmanError + (y - q)^2$
    end for
end for$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \nabla_{\boldsymbol{\theta}} BellmanError$

4.4.2 Bootstrapped Random Updates

The second proposed mechanism is to select a random starting point in an episode and update for a number of *iterations*. This proposal requires zeroing the internal state of the RNN before the updates.

4.5 R2D2

Recurrent Replay Distributed DQN (R2D2) is a distributed, DQN based agent that manages to achieve great performance in multiple domains. We consider the algorithm in this section for the introduction of Burn-In-Phases that improve the training results of Recurrent Agents [12].

Algorithm 3 Bootstrapped Random Updates

Require:
Require: $\mathcal{D}$ ▷ Trajectory Experience Replay
Require: $N >= 1$ ▷ Number of trajectories to sample
Require: K ▷ BPPT Steps
Require: $\boldsymbol{\theta}$ ▷ Weights
Require: $\boldsymbol{\theta}'$ ▷ target weights
Require: λ ▷ learning rate

$BellmanError \leftarrow 0$
for i in 0..N **do**
    h = **0**
    h' = **0**
    trajectory $\sim D$
    j $\sim [0 : |trajectory| - k]$
    trajectory $\leftarrow trajectory[j :]$
    for (s, a, r, s') in trajectory **do**
        $q, h \leftarrow Q(s, a, h; \boldsymbol{\theta})$
        $y, h' \leftarrow \arg\max_{a'} Q(s', a', h'; \boldsymbol{\theta}')$
        $y \leftarrow \gamma(1 - d(s'))y + r$
        $BellmanError \leftarrow BellmanError + (y - q)^2$
    end for
end for$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \nabla_{\boldsymbol{\theta}} BellmanError$

4.5.1 Burn-In-Phase

The *Burn-in-phase* is similar to Bootstrapped Sequential Updates in the sense that the whole trajectory is replayed, however, instead of updating from the beginning of the trajectory, the network is allowed to go through a series of steps in the trajectory in order to build the context/hidden state before adjusting the weights. The algorithm is available in 4.5.1.

Intuitively, Bootstrapped Sequential Updates violate the I.I.D property but, unlike DQNs, the input is not as correlated because RNN architectures have a hidden state and LSTMs have both a hidden state and a cell state, thus, while the input states are not

Algorithm 4 Burn-in-phase

Require: Trajectory Experience Replay $\mathcal{D}$
Require: number of trajectories to sample N
Require: Burn in steps K
Require: BackProp through time steps M
Require: weights $\boldsymbol{\theta}$
Require: target weights $\boldsymbol{\theta'}$
Require: λ learning rate
  $BellmanError \leftarrow 0$
  for i in 0..N **do**
    h = **0**
    h' = **0**
    trajectory $\sim D$
    j $\sim [0 : |trajectory| - K - M]$
    burn-in-trajectory $\leftarrow trajectory[j : j + K]$
    for (s, a, r, s') in burn-in-trajectory **do**
      $\_, h \leftarrow Q(s, a, h; \boldsymbol{\theta})$
      $\_, h' \leftarrow \arg\max_{a'} Q(s', a', h'; \boldsymbol{\theta'})$
    end for
    bptt-trajectory $\leftarrow trajectory[j + K : j + K + M]$
    for (s, a, r, s') in bptt-trajectory **do**
      $q, h \leftarrow Q(s, a, h; \boldsymbol{\theta})$
      $y, h' \leftarrow \arg\max_{a'} Q(s', a', h'; \boldsymbol{\theta'})$
      $y \leftarrow \gamma(1 - d(s'))y + r$
      $BellmanError \leftarrow BellmanError + (y - q)^2$
    end for
  end for
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \nabla_{\boldsymbol{\theta}} BellmanError$

I.I.D catastrophic interference may be avoided. Bootstrapped Random Updates require zeroing the internal state of the RNN and makes it harder for the Network to learn to use the internal state but the update is not as correlated. According to [36], both approaches exhibit the same behavior. Last but not least, the Burn-in-phase appears to facilitate better representation learning as the Recurrent Network achieved greater performance than the feed forward equivalent in Pong. The performance discrepancy suggests that RNNs may not only serve as memory modules but also assist in better feature extraction that results in better representation learning by considering the context [12], this observation is in line with [36].

4.6 C51

C51 is a distributional agent. Instead of learning the expected value, it learns a distribution. Learning distributions has the additional benefit of allowing us to estimate the variance between the estimated rewards of the actions. Thus, given two actions that have the same mean, a risk averse agent would prefer the one with the lowest variance. In addition, variance *could* be a proxy for the amount of exploration performed with particular state-action pairs, this however is not a guaranteed. Further on, estimating distributions makes the process of optimization easier since the targets are bound within some range, and thus, do not need to estimate values of different magnitudes [61].

4.6.1 Control

To perform control with C51, we compute the support for each atom, then estimate the mean by multiplying with each individual weight. In essence, we create a histogram, and then estimate the mean. Each atom (or bin) of the histogram has a corresponding

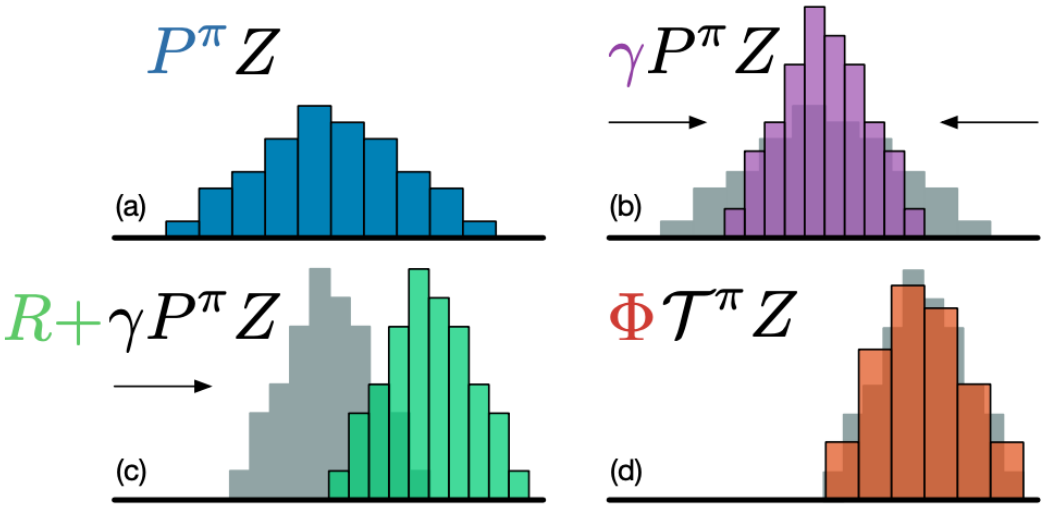

Figure 4: a) The distribution as estimated by the Q-Network with emerging policy π. b) applying the discount factor results in shrinking the distribution towards 0. c) Adding the reward shifts the distribution. d) The distribution is projected onto the target distribution. Source [61]

value, i.e. weight, we then use the weight to estimate the mean of the histogram. We will denote the support for atom i as $p_i(s, a)$, and its weight with z_i.

Algorithm 5 Q-value Estimation

Require: $s \in \mathcal{S}$, $a \in \mathcal{A}$
$q = \sum_i^N z_i p_i(s, a)$
return q

4.6.2 Optimization Procedure

In C51, we estimate the distribution of the next state, and discount the values by γ and add the reward r. Then we readjust the target distribution in 51 bins. The step of readjusting is necessary because we shift and squeeze the distribution, thus it is possible that it ends up out of bounds. The readjustment process ensures that estimations over the bounds are put into the final or first bin if the estimation is over or under the bin.

The error is reduced through CrossEntropyLoss [61]. A more thorough explanation on C51 exists in A.2.

Algorithm 6 C51 Optimization Step

Require: $s \in \mathcal{S}, a \in \mathcal{A}, \gamma, s' \in \mathcal{S}$
$a' \leftarrow \arg\max_{a'}$ Q-VALUE-ESTIMATION(s', a') ▷ Compute the best action
$b_i = (\gamma z_i) p_i(s', a') + r$ ▷ Create Projection
$m \leftarrow adjust\ bins(b, 51)$
return $-\sum_i^N m_i \log p_i(s, a)$

4.7 Rainbow

Rainbow is the current *published* state-of-the-art Q-learning method. It combines DQN with

- Distributional value estimation as in C51
- Dueling Value estimation modules as in Dueling networks
- Double Q-learning as in Double DQN

With Prioritized Experience Replay and Noisy Networks and N-step Return [2, 61, 60, 62, 49, 59, 63]. Further work incorporates Implicit Quantile Networks with Rainbow to achieve even greater performance. [2]

4.8 Off-Policy Correction

Although DQN and derivatives are off-policy algorithms and by definition, can use experience collected from other policies, they can only converge to an optimal policy iff that the source (behavior) policy and the current policy are not too far apart [64, 65].

[2] The work has not been published in a paper form, but is available here `https://github.com/medipixel/rl_algorithms`. We last accessed the page on the 29th of May, 2020. An archive for the repository exists in the WayBackMachine here `https://web.archive.org/web/20200529154653/https://github.com/medipixel/rl_algorithms`.

Not accounting for differences between policies when considering experience can lead to stability issues [64, 65].

Another concern choosing between algorithms that consider the whole return, i.e. Monte Carlo methods instead of single or few step methods. Monte Carlo methods allow the swift propagation of future rewards at the cost of requiring additional steps and increased variance, whereas few step methods can use the readily available experience at the cost of increased bias [17, 64]. We can bridge the gap using n-step return updates:

$$Q(s,a) \leftarrow Q(s,a) + \mathbb{E}_{\mu}[\sum_{t\geq 0}^{n} \gamma^t (\Pi_{x=1}^{t} c_x)(r_t + \gamma \mathbb{E}_{\pi} Q(s_{t+1}, \cdot) - Q(s_t, a_t))] \tag{32}$$

4.8.1 Importance Sampling

Importance Sampling (IS) is a technique that attempts to account for the bias introduced when sampling from a set that is generated from a different distribution. In RL, IS can be used to adjust for the *off-policy-ness* of transitions by accounting for the differences in the probability of selecting the same action between the two policies [17, 64, 66]. IS alters c in equation [32] to be the fraction of the probability of selecting that specific action using the current policy over the policy that 'experienced' the transition

$$c_x = \frac{\pi(a|s_x)}{\mu(a|s_x)} \tag{33}$$

where π is the learnt policy, i.e. current policy and μ is the behavior policy that generates transitions and a is the performed action. An issue with IS is that it exhibits high variance due to the fraction term. When the policy μ selects actions that are very likely in π, the ratio c becomes enormous and in the opposite case, results in small values. Thus, IS can result in variance explosions [64]. Consider for example an $\epsilon - greedy$ policy μ that anneals ϵ and a deterministic policy π, both using the same DQN. We optimize π and

select actions using μ. Let us also consider a transition selected in the early stages where the epsilon was large, with action a, that is also the greedy action. The consequence of using IS here is that $\pi(a|s_x)$ is guaranteed to be larger than $\mu(a|s_x)$ simply because ϵ was annealed to a smaller value, thus the greedy action is more probable and, thus $c_x > 1$. Since equation [32] uses a product, a series of selecting the greedy action is guaranteed to explode. This introduces instability issues that can cause a Neural Network to collapse [64].

4.8.2 Q(λ)

Q(λ) solves the variance of IS by using a constant value, $c = \lambda$. Given the following definition of 'off-policy-ness' $\epsilon = \max_s |\pi(s, \cdot) - \mu(s, \cdot)|$, which is the maximum difference of the action probabilities between the two policies. Q(λ) can converge to the Q^* with a worst case of $\lambda \leq \frac{1-\gamma}{2\gamma}$. This however, results in very small values. Due to the product operator, the small values quickly becomes useless. When attempting to learn Q^μ, we can converge when $\lambda < \frac{1-\gamma}{\gamma\epsilon}$ [64, 65]. Thus, Q(λ) learns quickly, *given* the two policies are not very different which requires knowing ε which is intractable [64]. Thus, Q(λ) works well in the on-policy scenario, but falls short in the off-policy case.

4.8.3 Tree Backup(λ)

On the other side of this, Tree Backup(λ) (TB(λ)) [67] sets the value c to

$$c_x = \lambda\pi(a_x|s_x) \tag{34}$$

This is useful when we consider radically different policies, it *wastes* experience on the on-policy case [64].

4.8.4 Retrace(λ)

Retrace(λ) strikes a balance between the three aforementioned techniques by setting the value c_x to

$$c_x = \lambda \min(1, \frac{\pi(a_x|s_x)}{\mu(a_x|s_x)})$$

Retrace(λ) accounts for the degrees of 'off-policy-ness' through a truncated IS, while preventing variance explosions by clamping the ratio and does not waste experience in the on policy case since

$$c = \begin{cases} \min(1, \frac{\pi(a_x|s_x)}{\mu(a_x|s_x)}) \geq \pi(a_x|s_x) & \text{in the on policy case} \\ \min(1, \frac{\pi(a_x|s_x)}{\mu(a_x|s_x)}) \leq 1 & \text{in the off-policy case} \end{cases} \tag{35}$$

Moreover, Retrace(λ) can use increasingly greedier policies, i.e. reducing ϵ in $\epsilon - greedy$, or decreasing entropy when using a Boltzman distribution, and can converge to the optimal policy *without* requiring infinite exploration in the limit[3] [64, 17].

Unfortunately, the λ term is another hyperparameter that needs to be tuned and is dependent on the underlying algorithm, the particular problem, and the γ. However, understanding the behavior of the correction as we outline above can give hints on how to choose it. For example, in the case of TB(λ), we can get away with a larger lambda function if we use $\epsilon - greedy$ with small ϵ.

[3] We suggest reading about Greedy In the Limit with Infinite Exploration (GLIE) in [17]

Chapter 5

Policy Gradients

Policy Gradient algorithms optimize directly the target policy. Value Function Approximation algorithms attempt to estimate the value of each action at a state and create a deterministic policy. Policy gradient methods differ in that they directly optimize the policy by increasing the probability of good trajectories to occur. Policy Gradient algorithms exhibit several advantages over Value-Function Approximations. First, they are able to learn stochastic policies; second, they are resistant to small changes whereas DQN and derivatives are not [2, 60, 59]; Policy Gradient methods are proven to at least converge to a locally optimal policy [68, 17], and last, but not least, they can learn how to solve continuous action problems where DQN and derivatives fall short.

The general idea is that if we sample enough trajectories, we can build an unbiased estimation of the performance of the policy and adjust it towards actions that create trajectories with high performance. In essence, Policy Gradients are Monte-Carlo Control through Policy Optimization.

Actor-Critic Methods improve upon plain Policy Gradient by incorporating two modules, an actor and a critic. The critic is a value function approximator and the actor is

a policy network. Actor-Critic methods alternate between optimizing the critic and the actor [51, 50, 47]

Policy Gradient methods are able to solve both discrete and continuous action problems. For discrete problems, they estimate a discrete probability distribution and sample from it. For continuous problems, the usual approach is estimating a multivariate Gaussian distribution by estimating the mean and the logarithm of the standard deviation. For discrete problems, we create a probability mass function using Softmax and sample from it.

5.1 Vanilla Policy Gradient

We will begin the introduction to On-Policy Policy Gradient Algorithms with REINFORCE [69, 17] and slowly incorporate improvements to build up to the full Vanilla Policy Gradient to justify introducing other algorithms and the reasoning behind their choices.

REINFORCE provides the basis for policy gradient algorithms. In REINFORCE, an agent consists of just a stochastic policy $\pi(s)$ that outputs a conditional probability distribution that is used to sample the action for the current state. The policy is adjusted such that trajectories that resulted in good returns are more likely to occur. The probability for a trajectory $\mathcal{T}$ to occur is

$$P(\mathcal{T};\boldsymbol{\theta}) = p(S_0)\prod_{t=1}^{T} p(S_t|S_{t-1},A_{t-1})\pi_{\boldsymbol{\theta}}(A_{t-1}|S_{t-1}) \tag{36}$$

$$\nabla_{\boldsymbol{\theta}} \log P(\mathcal{T};\boldsymbol{\theta}) = \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t|S_t) \tag{37}$$

Where $p(S_0)$ is the probability of the state S_0 to be the first state, $\pi(A_i|S_i)$ is the probability of selecting the action A_i at time i under policy π on state S_i. Since we are trying

to maximize the expected return and we would like to increase the probability of encountering trajectories that result in high returns, the reward function $J(\boldsymbol{\theta})$ we are trying to maximize is:

$$G_{\mathcal{T}} = \sum_{t=1}^{T} \gamma^{t-1} R_t^{\mathcal{T}}$$

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} [G_{\mathcal{T}}] \tag{38}$$

Where $G_{\mathcal{T}}$ are the rewards for the trajectory $\mathcal{T}$ and $R_t^{\mathcal{T}}$ is the reward at time t for trajectory $\mathcal{T}$. Since the goal is to maximize the function, we need to calculate the gradient.

$$\nabla_{\boldsymbol{\theta}} \hat{J}(\mathcal{D}; \boldsymbol{\theta}) = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{T} \in \mathcal{D}} \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t | S_t) \gamma^t G_{\mathcal{T}} \tag{39}$$

Where $\mathcal{D}$ is a collection of trajectories which are used to estimate the gradient, $G_{\mathcal{T}}$ is the return of the trajectory $\mathcal{T}$. In REINFORCE, the actions taken sampled from a distribution $\pi(\cdot|S_t)$, therefore we can not take the gradient with respect to a single instance, instead we use an estimation of the gradient by sampling a large number of trajectories. Because the estimation of the gradient is dependent on the probability of selecting a particular action at a particular state, REINFORCE can only be trained on-policy through trajectories collected using the same set of weights as those that were used to collect the roll-out.

5.1.1 Ignoring the Past

An initial observation on the update scheme of the weights is that good actions may be punished for mistakes that occurred earlier. A Natural idea is to implement a scheme similar to Monte Carlo Control (MMC) [17]. MCC uses the truncated return to adjust estimation of a particular action by considering only the future events [17]. This idea is often referred to as *rewards to go*. As a result of this modification, gradient becomes:

$$G^{\mathcal{T}}_{t:T} = \sum_{i=t}^{T} \gamma^{i-t} R^{\mathcal{T}}_i \tag{40}$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{T} \in \mathcal{D}} \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t|S_t) G^{\mathcal{T}}_{t:T} \tag{41}$$

Where $R^{\mathcal{T}}_i$ is the reward from trajectory $\mathcal{T}$ at time-step i, and $G^{\mathcal{T}}_{t:T}$ is the return from time-step t to time-step T for the trajectory $\mathcal{T}$.

While REINFORCE does work, and can learn a policy to solve particular problems, the algorithm is usually unstable and that is due to the high variance introduced when we use the return. In addition, REINFORCE does not consider the dynamics of the environment directly, i.e. it doesn't consider the new state, which is one of the sources of variance. Instead, it simply improves the probability of selecting an action based on just the return.

5.2 Advantage Actor Critic

To alleviate the susceptibility of REINFORCE to variance and noise, we can use a function that reduces the variance. The general policy gradient form

$$\nabla_{\boldsymbol{\theta}} J(\mathcal{D}; \boldsymbol{\theta}) = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{T} \in \mathcal{D}} \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t|S_t) \Psi_t \tag{42}$$

allows Ψ_t to be any of

- $G_{t:T}$
- $\sum_{t'=t}^{T} \gamma^{t'-t} R_{t'} - b(s_{t'})$[1]
- $V^{\pi}(s)$

[1] Any substitute of $b(s_t)$ is a valid baseline function

- $Q^{\pi}(s,a)$
- $A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s')$
- $r_t + \gamma V^{\pi}_{\boldsymbol{\theta}}(s') - V^{\pi}_{\boldsymbol{\theta}}(s)$

where all of the above functions are γ-just, and can be used in place of Ψ_t [70]. γ-just functions are functions that can approximate the advantage function.

Advantage Actor Critic (A2C) is the simplest improvement on REINFORCE. In the generalized policy gradient equation, it sets $\Psi_t = r_t + \gamma V^{\pi}_{\boldsymbol{\theta}}(s') - V^{\pi}_{\boldsymbol{\theta}}(s)$, i.e. the Advantage function. We select Ψ_t to be the advantage function because it has the lowest variance, for more details, see B.1,[70]. A2C is often referred to as 'Vanilla Policy Gradient'[2] , and, like REINFORCE, is an on policy Policy Gradient algorithm and the code is available in 5.2. Unlike REINFORCE, A2C uses a number of parallel workers to collect roll-outs. The workers are used to collect trajectories in order to get a better estimate of the true gradient. The return of the trajectory by definition has the greatest variance. We can reduce the

Algorithm 7 A2C

Initializestochastic policy $\pi_{\boldsymbol{\theta}}$
Initializecritic V_{ψ}
Initializelearning rate η
for K=1.. **do**
    Sample $\mathcal{D}$ trajectories
    Compute Rewards to Go
    Compute Advantage using any γ-just function
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}}(\mathcal{D})$ ▷ Perform gradient ascent
    $\psi \leftarrow \psi - \eta \nabla_{\psi}(V_{\psi}(s_t) - G_t)^2$ ▷ for all steps in all trajectories
end for

variance by considering just the state-value function $\Psi = V_{\boldsymbol{\theta}}(s)$. $V_{\boldsymbol{\theta}}(s)$ is less noisy simply because it provides an expectation of the return, instead of the discounted sum of the rewards. We can further reduce the variance by setting $\Psi_t = r_t + \gamma V^{\pi}(s_{t+1}) - V(s_t)$, i.e.

[2]Some people refer to REINFORCE as the Vanilla Policy Gradient, others to A2C

an estimation of the Advantage. Setting Ψ_t as the estimate of the Advantage function further reduces the variance because the value of $\gamma V_{\boldsymbol{\theta}}^{\pi}(s_{t+1})$ is in a included in $V(s)$, except that it is weighted. In addition, this estimation of the Advantage function takes into consideration the current state, the next state, and the observed reward, with the latter being the only real source of noise. Thus, the Advantage function also improves the policy based on the new state. The consequence is that we make actions that result in good new states more probable, instead of blindly following the return.

5.3 Generalized Advantage Estimation

Taking the idea of using an estimation of the advantage a step further, we can consider the Advantage across the steps in the episode [70] as that will give a better estimate of the *true* advantage function. Recall that we do not have access to the advantage function, only an estimate.

$$\hat{A}_t^{GAE(\gamma,\lambda)} \doteq \sum_{t'=0}^{T} (\gamma\lambda)^{t'} (\hat{A}(s_{t+t'}, a_{t+t'})_{t+t'}) \tag{43}$$

Where $\hat{A}(s,a)$ is any $\gamma - just$ estimator of the Advantage [70].

$\hat{A}_t^{GAE(\gamma,\lambda)}$ provides a significantly better and more stable estimate than $-V(s) + r + V(s')$ and is almost always the preferred method of estimating the advantage[70]. Although $\hat{A}_t^{GAE(\gamma,\lambda)}$ appears very similar to $TD(\lambda)$, $\hat{A}_t^{GAE(\gamma,\lambda)}$ is concerned with the advantage, where $TD(\lambda)$ is concerned with the return [70].

5.4 Trust Region Policy Optimization

Trust Region Policy Optimization (TRPO) [68] improves upon A2C 5.2 by solving the constrained optimization problem: Maximize the advantage of the newer policy over the

old policy subject to a constrain on the KL_D between the old and the new policy.

$$\boldsymbol{\theta}_{k+1} = \max_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_k, \boldsymbol{\theta}) \tag{44}$$

$$\text{subject to } KL_D(\boldsymbol{\theta}||\boldsymbol{\theta}_k) \leq \delta$$

$$L(\boldsymbol{\theta}_k, \boldsymbol{\theta}) = \mathbb{E}[\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)} A(s,a)] \tag{45}$$

Where $\boldsymbol{\theta}$ are the network weights before the optimization steps, and $\boldsymbol{\theta}_k$ are the network weights after k optimization steps on the current trajectories. Note that while we can perform multiple optimization steps, we do not use older experience like in DQN and derivatives.

The TRPO's optimization problem allows it to update the policy such that it makes monotonic improvements. TRPO uses a second order optimization algorithm (Conjugate Gradient) to update the policy, while the critic is updated using any gradient descent algorithm. The main issue of TRPO is that second order optimization methods are expensive to compute. In addition, it can get stuck on local optima [68], this is especially prominent when the batch size is small.

5.5 Proximal Policy Optimization

Proximal Policy Optimization [22] (PPO) improves upon TRPO [68] by removing all together the constrained optimization problem. Instead, it explicitly limits how large the difference between the old policy and the newer policy can be using a clipped objective.

$$J_{PPO}(s,a,\boldsymbol{\theta}_k,\boldsymbol{\theta}) = \min(\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)} \hat{A}_t^{GAE(\gamma,\lambda)}, Clip(\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)}, 1-\epsilon, 1+\epsilon)\hat{A}_t^{GAE(\gamma,\lambda)}) \tag{46}$$

Where $\boldsymbol{\theta}_k$ are the weights of the policy after k updates onto $\boldsymbol{\theta}$. To understand why this objective works and how the gradients flow, we need to understand how autograd works, mainly, how the Clip and min operations work.

First of all, the Clip operation sets a bound onto the value, so in [46], the Clip operation limits the upper bound that the value can have to $1 + \epsilon$ and the lower bound to $1 - \epsilon$, so any value out of the bounds is squashed into those. Second, the min operation can be easily decomposed in two operatios, first is an element wise multiplication, and the second, a linear combination of values. The element-wise multiplication takes each row and multiplies every element by 0, except the one that has the lowest value, that element is multiplied by 1. The linear combination is a multiplication with $\mathbf{1}$. This linear combination simply adds the columns together. Since every element except 1 was multiplied by 0, then the matrix has the smallest values of each row.

The element-wise clip operation results in gradients of the following form:

$$\nabla Clip(x, lower, upper) = \begin{cases} 1 & \text{if } lower \leq x \leq upper \\ 0 & \text{otherwise} \end{cases} \tag{47}$$

Thus, the clip operation does not propagate gradients when the probability of selecting an action becomes too large or too little. Thus, when we use the min operator with a clipped and an unclipped value, by definition, we select the clipped value. Since the gradient for the clipped values is 0, we disallow them from flowing backwards. This combination of operations effectively reduces the training batch to fewer values.

The ratio objective allows PPO to a) perform more than 1 optimization step, b) constrain the change of the policy and by proxy limiting the $D_K L$. The objective function increases the probability of selecting an action when its advantage is positive, and reduces it when its advantage is negative, in addition, it clips this change, such that the policy is

not significantly altered, much like TRPO. In addition, since PPO uses the probabilities of the older policy, it can perform many optimization steps whereas other Policy Gradient algorithms cannot since the probability of experiencing a trajectory after optimization is different.

Algorithm 8 PPO

Initializestochastic policy $\pi_{\boldsymbol{\theta}}$
Initializecritic V_ψ
Initializelearning rate η
for N=1.. **do**
    Sample $\mathcal{D}$ trajectories
    Compute Rewards to Go
    Compute Advantage using GAE.
    $\boldsymbol{\theta}_k \leftarrow \boldsymbol{\theta}$
    for K=1.. **do**
        $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + \eta \nabla_{\boldsymbol{\theta}_k} J_{PPO}$ ▷ Perform gradient ascent
    end for
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_k$
    $\psi \leftarrow \psi - \eta \nabla_\psi (V_\psi(s_t) - G_t)^2$ ▷ for all steps in all trajectories
end for

5.6 The Deterministic Policy Gradient

The equivalence between the deterministic policy gradient and the stochastic policy gradient has been proven in [71]. Since then, Deterministic policy gradients have taken robotics tasks by storm. We will not derive the equivalence, as [71] includes a thorough and comprehensive derivation. Instead we will build the intuition behind it.

Consider a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ where $\mathcal{A}$ is a continuous space. We can estimate the quality of this using a Q-network. The DQN Q-networks are a function in the form of $Q : \mathcal{S} \rightarrow R^{|\mathcal{A}|}$ because they only handle discrete actions. We can alter the Q-network to be a function of the form $Q : \mathcal{S} \times \mathcal{A} \rightarrow 1$, where we concatenate the state and the action spaces [39]. From here, we can treat the Q-network as a function that needs to be maximized.

$$J_{\boldsymbol{\theta}}(s) = Q_{\psi}(s, \pi_{\boldsymbol{\theta}}(s)) \tag{48}$$

$$\nabla_{\boldsymbol{\theta}} J_{\boldsymbol{\theta}}(s) = \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} Q_{\psi}(s, \pi_{\boldsymbol{\theta}}(s)) \tag{49}$$

In essence, we learn a differentiable function Q, as a surrogate to the actual problem that we are trying to solve. Then, we use this function to derive the policy.

5.7 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) [51] can be thought of as the spiritual successor of DQN, but for continuous action spaces. DDPG, like DQN, uses target networks to create more stable updates and avoid feedback loops. Unlike DQN though, DDPG uses a Polyak average update of the networks:

$$\psi_{target} \leftarrow \tau \boldsymbol{\theta}_{target} + (1 - \tau)\psi \tag{50}$$

$$\boldsymbol{\theta}_{target} \leftarrow \tau \boldsymbol{\theta}_{target} + (1 - \tau)\boldsymbol{\theta} \tag{51}$$

Where ψ denotes the weights of the critic, and $\boldsymbol{\theta}$ denotes the weights of the actor.

In order to update the critic, we use the two target networks to create the target value, and then update in the same fashion as DQN:

$$y = r + \gamma(1 - d) Q_{\psi_{target}}(s', \pi_{\psi_{target}}(s')) \tag{52}$$

$$\delta_t = y - Q_{\psi}(s, a) \tag{53}$$

$$J_{DDPG_{\psi}}(\mathcal{D}) = \mathbb{E}_{\langle s,a,r,d,s' \rangle \sim \mathcal{D}}[\delta_t^2] \tag{54}$$

In order to update the actor, we use the critic network directly:

$$J_{DDPG_{\boldsymbol{\theta}}}(\mathcal{D}) = \mathbb{E}_{\langle s,a,r,d,s' \rangle \sim \mathcal{D}}[Q_{\psi}(s, \pi_{\boldsymbol{\theta}}(s))] \tag{55}$$

An issue with the deterministic Policy Gradient is the lack of exploration. In [51], this is handled through action noise generated from an Ornstein-Uhlenberg process, alternatively, action noise sampled from a multivariate Gaussian or a factored multivariate Gaussian can also be used. We provide the algorithm for DDPG in 5.7.

Algorithm 9 DDPG

Initialize$\boldsymbol{\theta}$ ▷ Policy weights
Initializeψ ▷ Critic weights
$\boldsymbol{\theta}_{target} \leftarrow \boldsymbol{\theta}$
$\psi_{target} \leftarrow \psi$
$\mathcal{D} \leftarrow \{\}$
Initializenoise process AN
$s \sim \mathcal{S}^0$
for step 1.. **do**
  $n \sim AN$
  $a = clip(\pi(s) + n, -1, 1)$ ▷ Normalized Actions not included in [51]
  $a' = unscale(a)$ ▷ Unscale the clipped action to fit the action space
  take action a' ▷ Take the unscaled action
  Observe s', r, d
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle s, a, r, s', d\rangle\}$ ▷ Store the normalized action
  $s \leftarrow s'$
  if $d == True$ **then**
    Initialize noise process AN
  end if
  if is time to update **then**
    for however many updates **do**
      $\psi = \psi - \eta\nabla_{\psi} J_{DDPG_{\psi}}(\mathcal{D})$ ▷ Update Critic
      $\boldsymbol{\theta} = \boldsymbol{\theta} + \eta\nabla_{\boldsymbol{\theta}} J_{DDPG_{\boldsymbol{\theta}}}(\mathcal{D})$ ▷ Update Actor, gradient ascent
      $\psi_{target} \leftarrow \tau\boldsymbol{\theta}_{target} + (1-\tau)\psi$ ▷ Update target critic
      $\boldsymbol{\theta}_{target} \leftarrow \tau\boldsymbol{\theta}_{target} + (1-\tau)\boldsymbol{\theta}$ ▷ Update target actor
    end for
  end if
end for

The original DDPG algorithm did not include action scaling, however, it improves performance when the action bounds are outside $[-1, 1]$, this is likely because NNs perform poorly when values get too large. Scaling can be achieved through the hyperbolic tangent function.

5.8 Twin Delayed DDPG

DDPG suffers from a number of instability issues and poor performance. First of all, since we are treating Q as a function that is to be optimized, we need to be aware that NNs are not smooth approximators which means that when we optimizing against them, we may fall into local minimum/maximum in the input space that minimizes/maximizes the neural network, but not the true objective, i.e. the policy may exploit shortcomings in the critic network. Second, the critic in DDPG suffers from the same over-estimation issues that plague DQN. Twin Delayed DDPG(TD3) identified and addressed the above issues [50].

Twin

To address the issue of over-estimation, TD3 uses a pair, hence twin, Q-networks when computing the target values and selects the minimum of the two. Thus, the objective for the critic becomes:

$$y = r + \gamma(1-d) \min_{i\in\{1,2\}} Q_{i_{\psi_{target}}}(s', \pi_{\psi_{target}}(s')) \tag{56}$$

$$\delta_{t_i} = y - Q_{\psi_i}(s,a)$$

$$J_{TD3_\psi}(\mathcal{D}) = \mathbb{E}_{\langle s,a,r,d,s'\rangle\sim\mathcal{D}}[\sum_{i\in\{1,2\}} \delta_{t_i}^2] \tag{57}$$

It needs to be noted and emphasized that y is a target value thus that $\nabla_\psi y = \mathbf{0}$. TD3 uses Q_1 to update the actor:

$$J_{TD3_{\boldsymbol{\theta}}}(\mathcal{D}) = \mathbb{E}_{\langle s,a,r,d,s'\rangle\sim\mathcal{D}}[Q_{1_\psi}(s, \pi_{\boldsymbol{\theta}}(s)) \tag{58}$$

Delayed

To further improve stability, TD3 delays the rate of update of the policy and the target network to a fraction of the critic. The observation is that the policy should change slower than the critic, so as to preserve the performance and avoid rapid changes.

Smoothing

One important observation in DDPG is that the policy was likely exploit *bugs* in the critic. To avoid these bugs, TD3 performs policy smoothing. When computing the target value, the output of the target policy network is smoothed by adding noise from a Gaussian distribution.

$$a' = Clip(\pi(s') + Clip(\epsilon, -c, c), -1, 1), \epsilon \sim \mathcal{N}(0, \sigma) \tag{59}$$

$$y = r + \gamma(1 - d) \min_{i \in \{1,2\}} Q_{i_{\psi_{target}}}(s', \pi_{\psi_{target}}(s')) \tag{60}$$

$$\delta_{t_i} = y - Q_{\psi_i}(s, a)$$

$$J_{TD3_\psi}(\mathcal{D}) = \mathbb{E}_{\langle s,a,r,d,s' \rangle \sim \mathcal{D}}[\sum_{i \in \{1,2\}} \delta_{t_i}^2] \tag{61}$$

The overall algorithm, available in 5.8, is mostly the same as DDPG except for the improvements we mention above.

5.9 Soft Actor Critic

Soft Actor Critic (SAC) relaxes the basic goal of RL to also include entropy [47]. In RL, the main goal is to learn a policy that results in the highest expected return. SAC alters that goal to learning a policy with the highest possible expected return and expected

Algorithm 10 TD3

Initialize$\boldsymbol{\theta}$ ▷ Policy weights
Initializeψ ▷ Critic weights
Initializenoise process AN
Initializedelay ▷ TD3 sets the delay to 2
$\boldsymbol{\theta}_{target} \leftarrow \boldsymbol{\theta}$
$\psi_{target} \leftarrow \psi$
$\mathcal{D} \leftarrow \{\}$
$s \sim \mathcal{S}^0$
for step 1.. **do**
  $n \sim AN$
  $a = clip(\pi(s) + n, -1, 1)$
  $a' = unscale(a)$ ▷ Unscale the clipped action to fit the action space
  take action a' ▷ Take the unscaled action
  Observe s', r, d
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle s, a, r, s', d \rangle\}$ ▷ Store the normalized action
  if is time to update **then**
    for i in 1..however many updates **do**
      $\psi = \psi - \eta \nabla_{\psi} J_{TD3_{\psi}}(\mathcal{D})$ ▷ Update Critic
      if $i \equiv 0 (\text{mod delay})$ **then**
        $\boldsymbol{\theta} = \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} J_{TD3_{\boldsymbol{\theta}}}(\mathcal{D})$ ▷ Update Actor, gradient ascent
        $\psi_{target} \leftarrow \tau \boldsymbol{\theta}_{target} + (1 - \tau)\psi$ ▷ Update target critic
        $\boldsymbol{\theta}_{target} \leftarrow \tau \boldsymbol{\theta}_{target} + (1 - \tau)\boldsymbol{\theta}$ ▷ Update target actor
      end if
    end for
  end if
end for

entropy

$$\begin{cases} \sum_t^T \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t)] & \text{classic RL} \\ \sum_t^T \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))] & \text{maximum entropy RL} \end{cases}$$

where $(s_t, a_t) \sim \rho_\pi$ are the state-action marginals from following the policy π at timestep t, and $\mathcal{H}$ is the entropy of the policy's action distribution at s_t [47].

The conventional RL objective can be achieved by turning the temperature parameter α to 0, the temperature is a tradeoff coefficient that allow us to adjust how much we value entropy in the policy to be [47]. We can also incorporate discounts but, we need to discount both the reward and the entropy. We can retrieve $V^\pi(s)$ and $Q^\pi(s,a)$ through:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi}[r(s,a)] + \alpha\mathcal{H}(\pi(\cdot,s)) + \mathbb{E}_{s'\sim\rho_\pi(s)}[\gamma V(s')] \tag{62}$$

$$Q^\pi(s,a) = r(s,a) + \mathbb{E}_{s'\sim\rho_\pi(s)}[\gamma V(s')] \tag{63}$$

$$Q^\pi(s,a) = \mathbb{E}_{a,s\sim\rho_\pi(s)}[r(s,a) + \gamma(Q^\pi(s',a') + \mathcal{H}(\pi(\cdot|s')))] \tag{64}$$

Where $\rho_\pi(s')$ are the state marginals from following policy π at state s', α is the temperature.

SAC builds ontop of TD3 [50] and uses a pair of twin Q-Networks that are updated with the Squared Bellman Error of their prediction and the target value. The target value is the minimum Q-value of two Q-networks that are updated using Polyak updates, as in TD3. While SAC is off-policy, the target Q-values [64] and the state entropy are computed using actions sampled from the current policy. SAC also includes automatic temperature adjustment:

$$J_{SACC_\alpha}(\mathcal{D}) = \mathbb{E}_{a_t\sim\pi, s_t\sim\mathcal{D}}[-\alpha(\log\pi(a_t|s_t) + \overline{H})] \tag{65}$$

where $\overline{H}$ is a constant vector equal to the hyper parameter of the target entropy.

5.9.1 Continuous Soft Actor Critic

Similarly to TD3, the continuous policy outputs values in the $[-1, 1]$ range using hyperbolic tangent activation, however, the policy is stochastic whereas the TD3 policy is not. The SAC policy network outputs a factored Gaussian probability distribution using the reparameterization trick

$$a_{\boldsymbol{\theta}}(s) = tanh(\mu_{\boldsymbol{\theta}}(s) + \epsilon \odot \sigma_{\boldsymbol{\theta}}(s)) \qquad \epsilon \sim N(0, I) \tag{66}$$

where $\mu_{\boldsymbol{\theta}}(s)$ and $\sigma_{\boldsymbol{\theta}}(s)$ are outputs of the network and ϵ is factored Gaussian noise. The output is then squashed using the hyperbolic tangent activation. This combination allows SAC to *sample bounded* actions whose probability (and entropy) can be computed in closed form [47]. Furthermore, the sampling removes the need to use policy smoothing from TD3, however, the algorithm is mostly the same and is available in 5.9.1.

The update for the Q-functions becomes:

$$y = r + \gamma(1 - d) \min_{i \in \{1,2\}} [Q_{i_{\psi_{target}}}(s', a') + \mathcal{H}(\pi(\cdot|s'))] \qquad a' \sim a_{\boldsymbol{\theta}}(s) \tag{67}$$

$$\delta_{t_i} = y - Q_{\psi_i}(s, a)$$

$$J_{SACC_\psi}(\mathcal{D}) = \mathbb{E}_{\langle s,a,r,d,s' \rangle \sim \mathcal{D}}[\sum_{i \in \{1,2\}} \delta_{t_i}^2] \tag{68}$$

The policy update in SAC uses the minimum estimation of the Q function in order to update the policy:

$$J_{SACC_{\boldsymbol{\theta}}}(\mathcal{D}) = \mathbb{E}_{s \sim \mathcal{D}} \left[\min_{i \in \{1,2\}} [Q_i(s, a) + H(\pi_{\boldsymbol{\theta}}(\cdot|s))] \right] \qquad a \sim a_{\boldsymbol{\theta}}(s) \tag{69}$$

The above equation is the temperature objective for the continuous version, hence J_{SACC}, we include the objective for the temperature for the discrete case later.

Algorithm 11 SAC Continuous

Initialize$\boldsymbol{\theta}$ ▷ Policy weights
Initializeψ ▷ Critic weights
Initialize$H$ ▷ Target entropy
$\boldsymbol{\theta}_{target} \leftarrow \boldsymbol{\theta}$
$\psi_{target} \leftarrow \psi$
$\mathcal{D} \leftarrow \{\}$
$s \sim \mathcal{S}^0$
for step 1.. **do**
  $a \sim a_{\boldsymbol{\theta}}$
  $a' = unscale(a)$ ▷ Unscale the clipped action to fit the action space
  take action a' ▷ Take the unscaled action
  Observe s', r, d
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle s, a, r, s', d \rangle\}$ ▷ Store the normalized action
  if is time to update **then**
    for i in 1..however many updates **do**
      $\psi = \psi - \eta \nabla_{\psi} J_{SACC_{\psi}}(\mathcal{D})$ ▷ Update Critic with gradient descent
      $\boldsymbol{\theta} = \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} J_{SACC_{\boldsymbol{\theta}}}(\mathcal{D})$ ▷ Update Actor with gradient ascent
      $\psi_{target} \leftarrow \tau \psi_{target} + (1 - \tau)\psi$ ▷ Update target Critic
      $\alpha = \alpha + \eta \nabla_{\alpha} J_{SACC_{\alpha}}(\mathcal{D})$ ▷ Update temperature
    end for
  end if
end for

5.9.2 Discrete Soft Actor Critic

Discrete SAC trains a discrete actor critic agent in an off-policy way. This enables greater sample efficiency over on-policy policy gradient methods as the agent can reuse experience collected in the experience replay. At the same time, discrete SAC can learn faster since it does not need to rely on $\epsilon - greedy$ approaches to explore the state-action space.

In Continuous SAC (and TD3, DDPG), the Q-Networks are a function of the form $Q(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, we will use DQN style Q-Networks $Q(s, a) : \mathcal{S} \rightarrow \mathbb{R}^{|\mathcal{A}|}$, which allows

us to compute

$$V(s_t) = \pi(s_t)^T [Q(s_t) - \alpha \log(\pi(s_t))] \tag{70}$$

$$J_{SACD_\alpha}(\mathcal{D}) = \mathbb{E}_{s_t \sim D}\left[\pi(s_t)^T \left[-\alpha(\log(\pi(s_t)) + \overline{H})\right]\right] \tag{71}$$

$$J_{SACD_\pi}(\mathcal{D}) = \mathbb{E}_{s_t \sim D}\left[\pi(s_t)^T \left[\min_{i \in \{1,2\}} Q_{\Psi_i}(s_t) - a \log(\pi_{\boldsymbol{\theta}}(s_t))\right]\right] \tag{72}$$

$$J_{SACD_\Psi}(\mathcal{D}) = \mathbb{E}_{s \sim \mathcal{D}}\left[\sum_{i \in \{1,2\}} [Q_{\Psi_i}(s,a) - [r(s,a) + \gamma(1-d)V(s_{t+1})]]^2\right] \tag{73}$$

where α is the temperature and $\log(\pi(s_t))$ is the log probability of sampling each action from the policy. By computing the dot product of the log probability and the probability $\pi(s_t)^T \log(\pi(s_t))$, we can retrieve the entropy of the action distribution for a Categorical Distribution. Furthermore, because we know the action probabilities for a particular state, we do not need to obtain a Monte Carlo estimate (i.e. sample), we can instead compute $V(s)$ directly. Note that in [73], we are still using the minimum of the two QNetworks to estimate the action values. The algorithm is available in 5.9.2 and remains unchanged except for the objective functions.

5.10 Conclusion

In this chapter, we introduced the most common approaches to solving DRL problems using policy gradient methods. We introduced two classes of policy gradients, on-policy and off-policy methods. Policy gradient algorithms separate the policy from its evaluation and train two neural neural networks simultaneously. One can think of this as a form of Generalized Policy Improvement [17].

On-policy actor critic methods usually use an estimation of the advantage function in order to reduce the large variance of relying introduced as they rely on return. In contrast, the off-policy methods learn a Q function and optimize against it.

Algorithm 12 SAC Discrete

Initialize$\boldsymbol{\theta}$ ▷ Policy weights
Initializeψ ▷ Critic weights
Initialize$H$ ▷ Target entropy
$\boldsymbol{\theta}_{target} \leftarrow \boldsymbol{\theta}$
$\psi_{target} \leftarrow \psi$
$\mathcal{D} \leftarrow \{\}$
$s \sim \mathcal{S}^0$
for step 1.. **do**
  $a \sim \pi_{\boldsymbol{\theta}}$
  take action a
  Observe s', r, d
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle s, a, r, s', d\rangle\}$
  if is time to update **then**
    for i in 1..however many updates **do**
      $\psi = \psi - \eta\nabla_{\psi} J_{SACD_{\psi}}(\mathcal{D})$ ▷ Update Critic with gradient descent
      $\boldsymbol{\theta} = \boldsymbol{\theta} + \eta\nabla_{\boldsymbol{\theta}} J_{SACD_{\boldsymbol{\theta}}}(\mathcal{D})$ ▷ Update Actor with gradient ascent
      $\psi_{target} \leftarrow \tau\psi_{target} + (1-\tau)\psi$ ▷ Update target Critic
      $\alpha = \alpha - \eta\nabla_{\alpha} J_{SACD_{\alpha}}(\mathcal{D})$ ▷ Update temperature
    end for
  end if
end for

Off-policy methods are more sample efficient, however, they suffer from convergence issues. In contrast, On-policy methods exhibit more stable performance while they remain less sample efficient. The On-Policy methods that we outlined can run in parallel environments, this enables them to collect more experience in less time, thus they have better Wall-Time compared to Off-policy methods. For tasks such as robotics, the collection of trajectories is expensive, thus off-policy methods are preferred, in contrast, emulator tasks are usually very cheap to run, so On-Policy methods are used.

5.10.1 KL Divergence

KL Divergence is used as a regularization term in PPO and TRPO [22, 68] in order to prevent the gradients from performing significant changes in the policy. In essence, it is a measure of the difference between two distributions of selected action. This constraint

is particularly important on robotics tasks. Because the changes in the weights are not local, simply due to the gradients, even theoretically small changes can have catastrophic results in the policy. This is particularly important in robotics. Given a handcrafted policy, we want to be frugal with its modifications in order to avoid dangerous states that may damage the robot.

Chapter 6

Tools, Exploration methods and Hierarchies

6.1 Prioritized Experience Replay

An important shortcoming of DQN is that the ER buffer is sampled uniformly and thus, we waste optimization steps in values that do not improve the policy. A natural solution to selecting good transitions is using the [66], δ_t error. In [62], this is done in two ways through a priority value. The probability of selecting a transition is given through the priority value:

$$P(i) = \frac{p(i)/a}{\sum_j p(j)/a} \tag{74}$$

where the priority p for δ_t error is computed using its absolute value and is adjusted with the temperature parameter a.

The natural way to give priority is using the absolute δ_t error.

$$p(i) = |\delta_{ti}| + \varepsilon \tag{75}$$

This approach is referred to as proportional, and is vulnerable to outliers [66]. An alternative approach is to use a rank based priority:

$$p(i) = rank(\delta_{ti})^{-1} \tag{76}$$

Where $rank(i)$ is the rank of δ_{ti} after sorting the transitions using their absolute δ_t values. The rank based version is more robust to outliers and follows a power law distribution. While the two approaches guarantee a non zero probability of sampling each action, they do not guarantee that transitions with low initial δ_t error are seen often enough to ensure that their priority is correct at the current time step. The reason this is an issue is because as the policy changes, some transitions become more useful than others [66]. Note that new observations are inserted with the maximum priority.

Recall that Q-learning methods rely on expectations, thus, when we alter the sampling probabilities, we introduce a certain bias into the update. In order to account for the introduced bias, we use Importance Sampling to correct for the bias and reduce the effect of outliers.

$$w_i = (\frac{1}{N \cdot P(i)})^{\beta} \tag{77}$$

Where β dictates how aggressively we account for bias. Q-learning methods are by definition biased, thus, we can allow them to be biased while the model learns and anneal β towards the end of the training process [66].

The objective function remains almost the same as Double DQN, except for the bias correction weight, the algorithm remains the same for the most part is available in 6.1.

$$\nabla_{\boldsymbol{\theta}} J_{PrioritizedQ}(\boldsymbol{\theta}, \boldsymbol{\theta}') = \frac{1}{N} \sum_{s,a,r,s',w \sim \mathcal{D}} w \cdot \delta_t \nabla_{\boldsymbol{\theta}} Q(s, a; \boldsymbol{\theta}) \tag{78}$$

Algorithm 13 Prioritized Experience Replay DQN

Initialize empty Prioritized Experience Replay $\mathcal{D}$ with capacity N
Initialize random Q-network Q
Initialize learning rate λ
Initialize PER bias compensation β
Initialize policy $\pi_{\boldsymbol{\theta}}$ with weights $\boldsymbol{\theta}$
$\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$
$s \sim U($ initial states$)$
for T Steps **do**
    $a \sim \pi_{\boldsymbol{\theta}}(a|s)$
    $r, s' \sim p(s_{t+1}|s, a)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s, a, r, s', priority = max)$
    $s \leftarrow s'$
    if is time to update **then**
        $batch \sim \mathcal{D}$
        compute δ_t for batch
        compute $p(\cdot)$ for batch using β
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J_{PrioritizedQ}(\boldsymbol{\theta}, \boldsymbol{\theta}')$
        update priority for batch in $\mathcal{D}$
    end if
    if is time to update target weights **then**
        $\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$
    end if
    anneal β
end for

6.2 Hindsight Experience Replay

In UMDPs, an agent is given a task to complete over the duration of an episode. The reward the agent receives is based on the completion of the task. Usually the agent is given a reward of 1 on completion and no reward in-between. In the DRL setting, the state and the goal, as given by the environment, are transformed into an input through concatenation, the vector is then given to the agent. Because the reward function is usually uninformative, this approach results in an agent that wonders aimlessly until it stumbles on the desired goal. Hindsight Experience Replay [38] (HER) provides a mechanism that allows an agent to make efficient use of experience collected from a UMDP and learn significantly faster by re-purposing experience.

HER, just like ER from DQN stores the collected experience in a construct and samples them during training. In contrast to ER though, HER also creates experiences with *Hindsight*. These hindsight experiences instead of being the tuple $\langle s_i, g_i, a, r(g_i, s_{i+1}), s_{i+1} \rangle$, for state, goal, action, reward, and next state, they are the tuple $\langle s_i, s_j, a, r(s_i, s_j), s_{i+1} \rangle$, where g_i is replaced by an observed state using a sampling policy G. This change allows the agent to generalize better across states because it sees more rewards, has more experiences, and thus gains a better understanding of the state space [38]. In [38] the authors propose three sampling methods:

- Final: The final state of the episode
- Episode: A random state sampled from all the states observed in an episode
- Future: A future state from the same trajectory

with Final and Future achieving the best results. The introduction of HER effectively allows an agent to consider multiple goals in the state space. HER consists the basis for

a number of Hierarchical algorithms that are considered later in the thesis. Since HER is a general algorithm orthogonal to others, we provide a general approach in 14.

Algorithm 14 Store With Hindsight

for all $\langle s_i, g_i, a, s_{i+1}\rangle \in \mathcal{T}$ **do** ▷ For all transitions in the trajectory.
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle s_i, g_i, a, r(s_i, g_i, a_i), s_{i+1}\rangle\}$
    for all $g' \sim G(S)$ **do**
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle s_i, g', a, r(s_i, g', a_i), s_{i+1}\rangle\}$ß
    end for
end for

Note that HER is usually used in Robotics tasks where the reward function is usually provided by the environment and thus we can use it. In cases where the reward function is not provided, alternative functions are negative cosine similarity: $\frac{-s \cdot g}{\|g\|_\mu^2 \|s\|_\mu^2}$, or the norm of the difference e.g. $L1, L2$.

6.3 Noisy Linear Layers

A particularly important issue in RL is the lack of exploration schemes. This can cause agents to fall into locally optimal behavior [49, 20, 51]. This is particularly problematic when we consider deterministic methods such as DQN. Many algorithms address this issue by using an $\epsilon - greedy$ exploration policy, or by introducing noise in the action space or through entropy maximization methods [49, 51, 50, 47].

An alternative approach is to introduce noise in the weight space. Consider a Linear function

$$L(x; W, b) = xW + b$$

Where W is the weight matrix and b is the bias vector.

We can make the function stochastic by sampling noise from a Gaussian distribution for both terms using the reparameterization [53] trick

$$L(x; W_\mu, W_\sigma, b_{mu}, b_\sigma) = x(\epsilon \odot W_\sigma + W_\mu) + (\xi \odot b_\sigma + b_\mu) \tag{79}$$

where W_μ, W_σ correspond to the mean and the standard deviation of the noisy weight matrix; b_μ, b_σ are the mean and the standard deviation of the bias vector; and ϵ, ξ are sampled Gaussian noises from the same space as W_μ and b_μ respectively, with mean 0 and $\sigma = 1$.

Thus, for off-policy methods, we can use the noisy linear layers instead of linear layers to introduce noise in the parameter space for better exploration. Noisy linear layers in *On-policy* Policy Gradient methods are quite problematic as the optimization procedure will use experience that was collected with different weights [49]. To handle this, the agent should retain the same noisy weights through the optimization step as those that collected the transitions regardless of the number of the collected rollouts.

6.4 (Pseudo-)Count Based Exploration

Exploration of the world is paramount to learning good behavior. Curious agents learn to explore the world through intrinsic motivation. In its core, intrinsic motivation provides guidance to an agent as to what and where is should learn more of [72, 48, 73]

Pseudo-Count Based Exploration works by learning a density model of the states based on the sequence of observed states. In its core, Pseudo-Count Based Exploration learns a ‘coding‘ of the input state and the observation history, and gives back to the agent the probability of coding the state [72, 73]. An always novel agent will always observe a

positive reward as the probability of novel states is approximately 0 [72, 73]. This is done through a generative model.

Extending the above idea, DQN-PixelCnn [73] combines a PixelCnn[74] as the density model along with Mixed Monte Carlo (MMC) return:

$$Q(s,a) = Q(s,a) + \alpha[(1-\beta)\delta_t(s,a) + \beta(\delta_t MC(s,a))] \tag{80}$$

and $Retrace(\lambda)$[64]. Mixed δ_t rewards allow for better propagation of intrinsic reward back to previous states which facilitates better exploration and improved performance [73].

6.5 Random Network Distillation

Auto-regressive intrinsic reward[75, 76] models fall short to the noisy-tv problem [48]. In short, the Noisy-TV problem occurs when an agent has no incentive to explore because the environment itself provides novel observations even when the agent does not work towards the goal. Under this circumstances, such agents do not perform any actions as that will decrease the intrinsic reward they receive.

Random Network Distillation (RND) solves the noisy tv problem by using a randomly initialized network. The random network receives the observation and outputs a lower dimensional vector, i.e. a compression of the observation. Then, another network tries to predict the output of the random network. In RND the intrinsic reward is given by the difference between the predicted output of the random network and the actual output of the network. This change causes the network to be more robust to noise and entropy over the previous approaches [48].

$$J_{RND}(\boldsymbol{\theta}) = ||\hat{f}(x) - f_{\boldsymbol{\theta}}(x)||_2^2 \tag{81}$$

where $\hat{f}(x)$ is the random network, and $f_{\boldsymbol{\theta}}(x)$ is the network that is part of the agent.

6.6 Bootstrapped DQN

Bootstrapped DQN [20] aims to improve the exploration properties of DQN. In [2], DQN uses an $\epsilon - greedy$ policy and derivative works and implementations use an annealing ε. $\epsilon - greedy$ strategies are *dithering*, they are not deep, nor targeted. While $\epsilon - greedy$ strategies work for bandits and contextual bandits, these problems consider only single steps. In contrast, RL considers trajectories and indefinite amount of steps, thus dithering strategies do not provide an incentive to perform long sequences of exploration.

Bootstrapped DQN consists of N independent linear heads and a set of shared weights. Each head is trained with its own subset of the experience collected from the ER and provides as estimate of the Q value. In addition, each head has its own equivalent target head.

Training in Bootstrapped dqn consists of selecting a random head at the start of each episode and storing the collected experience in the ER. When storing an experience tuple, the tuple is marked using a probability distribution M. The mark signifies which heads can use that tuple. Thus, each head is not trained with the experience it collects, but rather, with a sample of all the collected experience. This allows different heads to encounter state-action pairs that they would not have been encountered otherwise. In essence, Bootstrapped DQN exploits the uncertainty through networks that 'lack experience' to drive exploration.

Bootstrapped DQN alters the DQN Objective function in the following manner:

- The backup equation remains mostly the same as the Double DQN objective.
- The gradient is filtered using the mask m_t to prevent the gradient from propagating to the wrong heads.

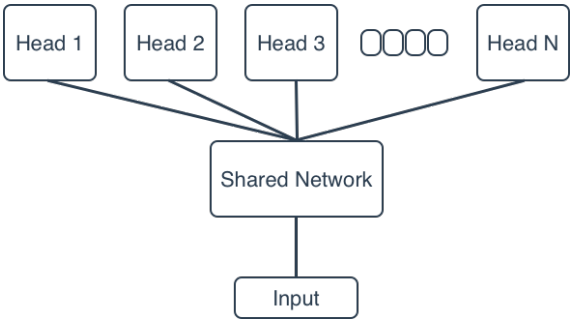

Figure 5: The network architecture for Bootstrapped DQN. The network share a collection of weights, in [20] the shared network is a CNN with the same convolution layers as [2].

$$
\begin{aligned}
y &= r + \gamma(1 - d(s'))Q_k(s, \arg\max_{a'} Q_k(s, a'; \boldsymbol{\theta}'); \boldsymbol{\theta}) \\
\delta_t &= y - Q(s, a) \\
\nabla_{\boldsymbol{\theta}} J_{bsQ}(\boldsymbol{\theta}, \boldsymbol{\theta}') &= \frac{1}{N} \sum_{s,a,r,s',m \sim \mathcal{D}} \sum_{i=1}^{k} m_k \cdot \delta_t \nabla_{\boldsymbol{\theta}} Q_k(s, a; \boldsymbol{\theta})
\end{aligned} \tag{82}
$$

Where $Q_k(s, a; \boldsymbol{\theta})$ is the Q value for state, action and head s, a, k respectively, m_i is a mask that denotes whether the experience sampled can be used to adjust the head i.

6.7 h-DQN

h-DQN [27] builds on the original DQN algorithm and extends it in order to learn temporal abstractions. h-DQN uses three components, two neural networks and a critic. The two neural networks are the meta-controller and the controller. The meta-controller outputs goals and the controller attempts to achieve them, the critic decides whether the goal was achieved or not.

The meta-controller operates in a coarser granularity than the controller and outputs goals from the observation space by creating a binary mask of an area. The meta-controller is a DQN agent that is trained through the extrinsic reward[27].

Algorithm 15 Bootstrapped DQN

Initialize empty Experience Replay $\mathcal{D}$ with capacity N
Initialize random Q-network Q with weights $\boldsymbol{\theta}$ and K heads
Initialize learning rate λ
Initialize policy $\pi_{\boldsymbol{\theta},h}$ with weights $\boldsymbol{\theta}$ and head h $\sim U(\{1..K\})$
Initialize distribution M to create bootstrap masks
$\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$
$s \sim U($ initial states$)$
for T Steps **do**
    $a \sim \pi_{\boldsymbol{\theta},h}(a|s)$
    $r, s' \sim p(s_{t+1}, r|s, a)$
    $m_t \sim M$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s, a, r, s', m_t)\}$
    $s \leftarrow s'$
    if is time to update and can sample C transitions from $\mathcal{D}$ **then**
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \lambda \nabla_{\boldsymbol{\theta}} J_{bsQ}(\boldsymbol{\theta}, \boldsymbol{\theta}')$
    end if
    if is time to update target weights **then**
        $\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$
    end if
    if is time to change head **then**
        $h' \sim U(1..K)$
        $\pi_{\boldsymbol{\theta},h} \leftarrow \pi_{\boldsymbol{\theta},h'}$
    end if
end for

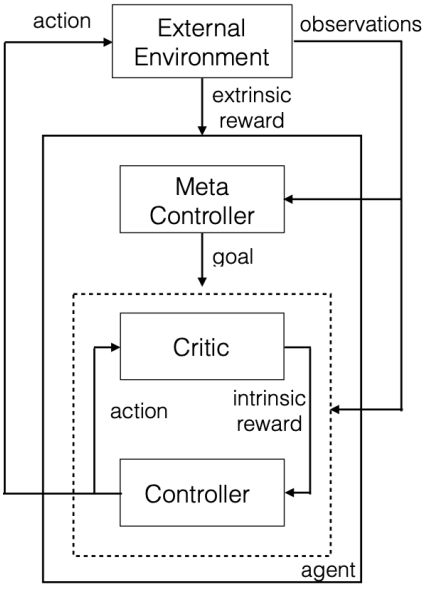

Figure 6: The schematic of the h-DQN architecture. Source [27].

The critic is an arbitrary function that decides whether the goal was achieved and provides the intrinsic reward for the controller. Because the critic is arbitrary, it needs to be tuned to the particular task at hand. The critic gives a 0 reward in every step and a reward of 1 when the goal is achieved[27].

The controller is a DQN agent that learns to achieve the goals of the meta-controller. The controller uses an $\epsilon - greedy$ behavior where the ε is annealed from 1 to 0.1 depending on the rate with which the controller achieves goals. If the success rate is equal or greater than 90% then $\varepsilon = .1$, otherwise it is linearly annealed[27].

h-DQN suffers frm two issues. The first issue is the lack of control with respect to how long the lower level policy operates. This is due to the way the 'critic' function is defined. h-DQN allows the lower level policy to operate until the critic function decides the goal is achieved, or the episode terminates. The second issue is that the model is not end-to-end differentiable.

6.8 Feudal Networks

FeUdal Networks[77] (FUNs) are based on the principle of separation of concerns introduced in Feudal Learning [78]. FUNs learn a meta-policy, the manager, and a primitive policy, the worker. The manager produces goals in a latent space common to both policies and the worker works towards materializing the goal. In order to learn, the manager uses the extrinsic reward that depends on the agent's actions. The worker learns by receiving an intrinsic reward that is based on the distance of the goal and a latent representation of the state, the closer the latent state to the goal, the higher the reward.

In essence, the worker is a meta-learner, it receives an input and an embedding of the task and learns to solve them. This observation allows us to use techniques from meta-learning to insert more information to the network. More specifically, the goal is embedded using a multiplicative relationship as it prevents the worker from ignoring it [77]. It should be noted that the goal as created from the manager is detached from the computation graph, this is done to prevent the loss of the worker from interfering with the loss of the manager [77]. The manager can be trained using a policy gradient algorithm since we require continuous actions, the worker can be trained with on-policy or off-policy discrete methods, however, [77] uses asynchronous A2C for both and joins the losses.

FUNs were used to solve Montezuma's revenge [1, 77], thus, they use a CNN as a vision module, however, for other problems, the vision module can be a multi-layer neural network. The meta-policy sets goals in the latent space of the vision module, i.e. the space that the vision module compresses the observation. Since Montezuma's Revenge is partially observable, both agents use LSTMs[33]. Because the manager needs to operate in coarser granularity than the worker, there needs to be a way to make the operation

coarse. FUNs introduced the idea of dilated LSTMs that operate similarly to clockwork RNNs [79], except instead of just ticking at a lower rate, they observe all the inputs from the previous up to the current tick. We can see the general architecture of FUNs in figure 7.

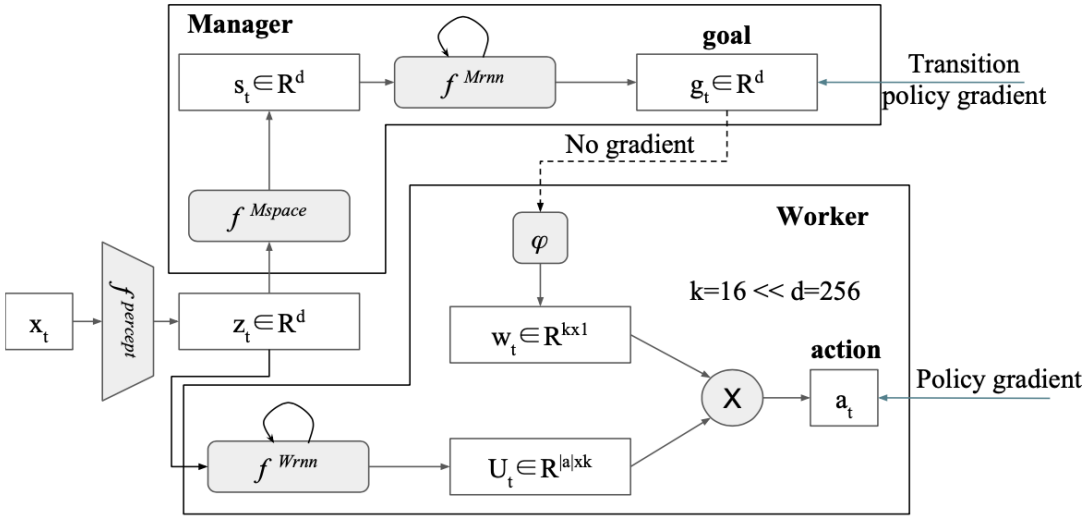

Figure 7: The general architecture of the FUNs. $f^{percept}$ is the vision module, f^{Mrnn}, f^{Wrnn} are the RNNs for the manager and worker respectively, f^{Mspace} is a linear layer followed by a ReLu activation that creates the implicit state of the model. The goal of the worker is embedded using ϕ into a lower dimension, then, the dot product of the embedding and the result of the neural network is taken to create the log probability for the actions [1, 77]. Figure source [77].

6.8.1 Feudal Learning

Feudal Learning [78] uses a managerial approach to constructing hierarchies. In Feudal Learning, a high level policy takes an observation and provides to the lower level policy a task to solve. The manager observes the environment at a temporal scale that depends on its level in the hierarchy. Higher level policies operate at a coarser temporal scale and the lowest/primitive policy operates at every timestep. This separation of concerns allows

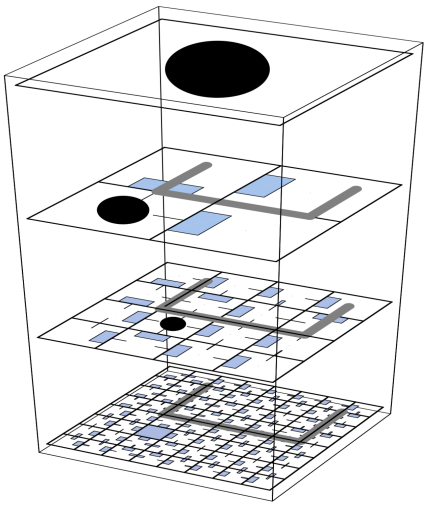

Figure 8: An example of a feudal system in a maze task. Source [78].

an agent to remain agnostic on how the lower level policies operate [78]. In addition, it allows the policies to remain agnostic to the rewards received by the higher level policies as their rewards depend on whether the goal was achieved. An example of such a policy can be found in figure 8.

While in principle feudal learning should work, it is rather limited in that the hierarchy needs to be crafted specifically for the task at hand and it's observation space. This prevents us from building a general algorithm that can solve decomposable problems, however it provides the basis for other algorithms.

6.9 Competitive Ensembles of Information-Constrained Primitives

Many Hierarchical algorithms attempt to solve problems by learning a meta-policy that triggers specific actions to occur in a lower level policy. Algorithms that learn both a meta-policy and a primitive policy require learning and extracting features from all states [80]. In [80], it is argued that such policies learn slowly because they need to learn both a temporal or spatial structure and the meta-policy needs to learn to produce actions for

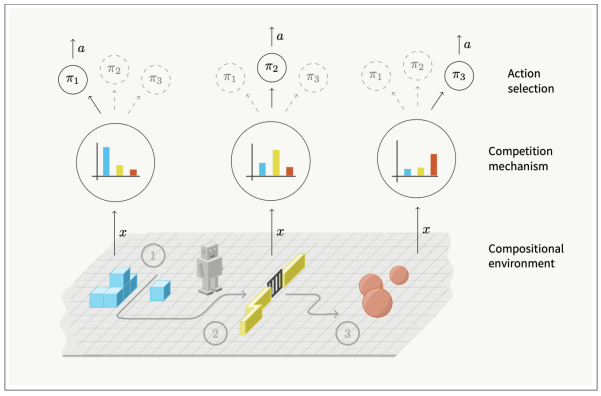

Figure 9: An illustration of the behavior of CEICP. To the left, we see that composite environments activate different primitive policies of the agent. To the right, we see how an action is selected. The agent creates K latent variables, the latent variables are used by the decoders to create k different action distributions for each state, distributions are selected by sampling based on the information usage of the primitive, and the action is then sampled from the selected distribution. Source [80]

all observations. Furthermore, the meta-policy usually operates at a different granularity, thus, there needs to be a mechanism to stop the primitive policy when it achieves the goal [30]. While such a mechanism is possible in robotic tasks, it becomes problematic in video game environments as the input is usually high dimensional. Other work simply ignores goal checking [32].

Competitive Ensembles of Information-Constrained Primitives (CEICP) [80], uses an ensemble of primitive policies and selects an action based on information. Each policy learns an encoder $p_{enc}(z|s)$ and a decoder $p_{dec}(a|z)$ that creates primitive actions

$$\pi^k_{\boldsymbol{\theta}}(a|s) = \int_z p_{enc}(z_k|s) p_{dec}(a|z_k) dz_k \tag{83}$$

this results in an overall policy that is a mixture of experts, each specializing in a sub-task

$$\pi_{\boldsymbol{\theta}}(a|s) = \sum_k c_k \pi^k_{\boldsymbol{\theta}}(a|s) \tag{84}$$

where $c_k = \delta_{kk'}, k' \sim p(k|s)$ where $a_k(s) \doteq p(k|s)$ is the probability of selecting primitive k in state s. The encoder constructs a latent representation of the state z_k that is used to

by the decoder to construct a distribution over the actions. The encoder is penalized for the distance between the latent z_k and a Gaussian prior $p(z)$. The penalty is given by the KL divergence between the two latent variables, i.e. the variational bottleneck objective [81]:

$$\mathcal{L}_k = D_{KL}(z_k||p(z)) = D_{KL}(p_{enc}(z_k|s)||p(z)) \tag{85}$$

The penalty forces the policies to constrain their knowledge over the input state. In essence, the more information the encoder puts in the latent variable, the greater the distance to the prior. In order to select a primitive, we use 85 in a Boltzman distribution and sample from it

$$a_k = \frac{\exp(\beta\mathcal{L}_k)}{\sum_i \exp(\beta\mathcal{L}_i)} \tag{86}$$

In order to transfer information about the reward, [80] uses A2C [82], but other Policy gradient algorithms may be used as well. This is done by weighing the reward for each particular primitive by a_k: $r^k = \sum_{t=0}^{T} r_t \gamma^t a_k$. Agents with high probability of acting at particular states are rewarded more and penalized more. This creates an incentive to act in particular subspaces of the observation space and ignore others.

At the moment, the different policies may be overlapping. In order to promote diversity, we include another regularization loss:

$$\mathcal{L}_{reg} = \sum_k a_k \mathcal{L}_k = -\mathcal{H}(a) + \log(\sum_k \exp(\mathcal{L}_k)) \tag{87}$$

Where $\mathcal{H}(a)$ is the entropy of the action distribution. The proof for 87 can be found in [80] and the final objective is:

$$\mathcal{J}^k_{\boldsymbol{\theta}} = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[r^k] - \beta_{ind}\mathcal{L}_k - \beta_{reg}\mathcal{L}_{reg} \tag{88}$$

where β_{ind} and β_{reg} are weight parameters for the respective terms.

While Competitive-Ensembles can successfully learn skills and can generalize, it does not provide a mechanism for temporal abstraction. The lack of temporal abstraction does not assist in solving the credit assignment problem. The second issue of Competitive-Ensembles is the introduction of yet another hyperparameter that needs task dependent tuning.

6.10 Hierarchical Actor Critic

Hierarchical Actor Critic (HAC) is based on the observation that agents that operate in a Feudal Learning [78] hierarchy are effectively solving a goal-based problem [30]. Due to the goal based nature of Feudal Learning, we can exploit HER [38, 30], to teach an agent how to solve its particular problem through Hindsight experience.

The issue with the above approach is that it assumes that each level in the hierarchy works independently of the others. This is false. If we treat each agent in isolation, then any non primitive agent assumes that the underlying MDP is stationary and thus it can solve its problem. This assumption does not hold because for each non primitive agent (i.e. any agent that produces goals and not primitive actions), the agent below alters its perceived dynamics because it learns. This results in a non stationary MDP which gravely hinders convergence and performance [30]. HAC solves the non-stationary dynamics by treating each sub-policy as if it was already optimal and thus allows an agent to assume that the dynamics do not change [30]. HAC defines 'optimal' as achieving the goal state in *as few steps as* possible [30]. The second form of non-stationary dynamics involves exploration. In order to learn an optimal policy, exploration is necessary, and thus we need to perform it. Moreover, even if exploration does not assist in altering

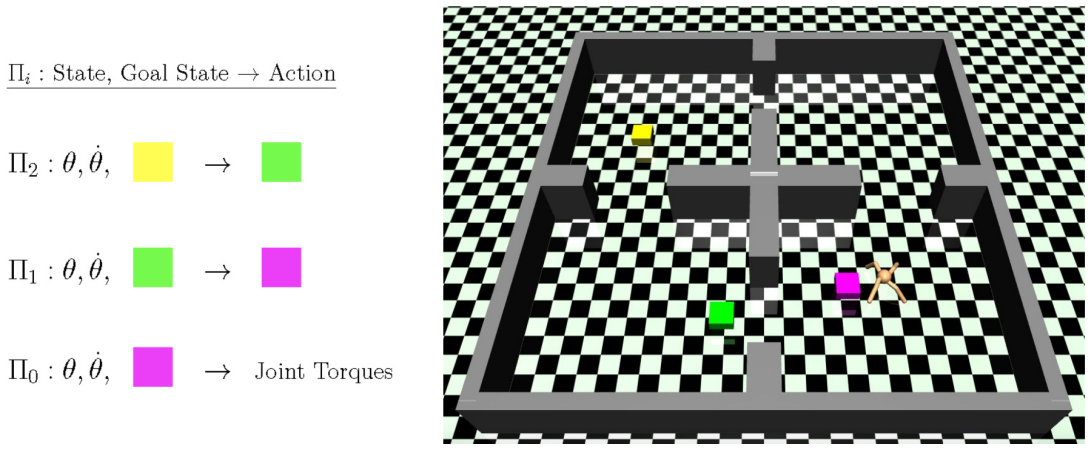

Figure 10: An ant moving in a continuous four rooms domain problem. The agent consists of three hierarchies, Π_0, Π_1, Π_2, Π_0 performs primitive actions and has a goal provided by Π_1 (pink), Π_1 perms a high level action, i.e. sets the goal in pink for Π_0 based on the goal provided by Π_2 (green). Π_3 produces the green goals for the Π_2 policy based on the yellow goal provided by the environment as the current task. Source [30].

the underlying policy, the dynamics remain non-stationary, because exploration is usually annealed over the learning process.

HAC is based on DDPG [51]. DDPG is an off-policy method that uses an Experience Replay in order to improve sample efficiency. HAC uses N DDPG agents in a hierarchy and each agent, except the primitive, produces actions that are in the same space as the observations. An example of this is visible in figure 10 the green and pink are actions produced by the high level policies, while yellow is the goal provided by the environment. Because each agent operates in a goal based environment, we can use HER to generalize the goals as usual, *and* we can also use future states as actions that were performed even if they were not. Thus, the fact that the goal space, the action space, and the observation space for all non primitive agents is the same allows us to treat lower policies as already optimal.

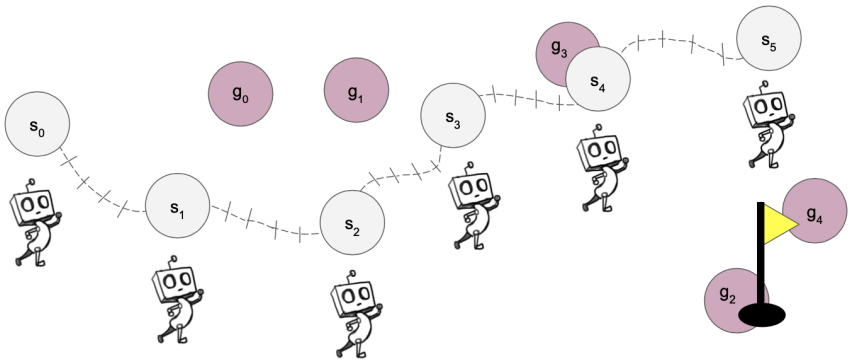

Figure 11: A trajectory in a toy problem. The flagpole indicates the goal of the top level policy, the pink circles indicate the actions of the top level policy and goals of the primitive policy. The gray circles indicate the achieved goals and the dashes indicate the primitive actions. Source [30].

HAC runs each agent for Hsteps, at which point, control returns back to the higher level policy, the higher level policy introduces a new goal and the process repeats until termination, an example trajectory can be found in figure 11. In order to learn, HAC uses three types of action transitions:

1. Hindsight Action Transitions. These transitions treat the current state as a selected action. For example, in figure 11, the agent is at state s_0 and sets as action g_0. The robot ends up in s_1, thus, the higher level policy can store $\langle old-state = s_0, new{-}state = s_1, action = s_1, reward = -1, goal = yellow{-}flag, done = False \rangle$. In its transitions. The higher level policy stores reward of -1 because it assumes the lower level policy is optimal and thus it can reach s_1 in a single step. When the goal is achieved, e.g. the robot reaches the flag, then it uses a reward of 0, to indicate that the goal is reached and $done = True$ to indicate end of episode.

2. Hindsight Goal Transitions. These are the same transitions used in HER [38]. Hindsight Goal Transitions add both the experienced transition, from the previous example: $\langle old - state = s_0, new - state = s_1, action = g_0, reward = -1, goal = yellow - flag, done = False \rangle$, and a transition of the form $\langle old - state = s_t, new - state = s_{t+1}, action = g_t, reward = X, goal = Y, done = Z \rangle$, where the hindsight goal Y is selected as from one of the future states, X is 0 if the hindsight goal is the next state, otherwise -1 and Z is true if the next state is the hindsight goal, false otherwise.

3. Goal Testing Transitions. The previous two forms of transitions limit the policies to learning how to achieve short goals, this is due to the fact that they alter the done signal. This results in myopic agents that do not know how to react to longer policies. This is also a shortcoming of the need for exploration, noisy policies may be unable to achieve the proposed goal. To handle this issue, HAC uses Goal Testing Transitions with probability λ. These transitions tell the agent below to be deterministic and to attempt to reach the goal. If the goal is achieved, then the agent uses a reward of 0, if the policy below fails, then the agent receives a reward of -H and sets the done signal as True. This teaches the agent about distant goals as well.

The major short-coming of HAC is that it only works in robotics. This is a consequence of HAC's definition of optimality and that the goals are *tied* to the state and ignores extrinsic rewards. In robotic tasks, the goals need to be achieved as fast as possible, the intermediate rewards are 0, and when the agent reaches the goal, it receives a positive reward. Thus, there is no notion of optimality with respect to the collected rewards because there are not any rewards.

Furthermore, the agents in HAC produce actions in the observation space. This is problematic in POMDPs because the same observation can be produced by an indefinite amount of internal states. An alternative solution is to use actions in a latent space produced by an RNN. This is however makes the problem non-stationary because the latent variable is produced through a function that changes with training, if we stop the gradients from flowing in the RNN then the representation of the state will probably be poor and informative.

Last but not least, HAC requires a notion of 'closeness' to the goal. Since the higher level policy performs a check on whether the goal was achieved, the distance needs to be quantified in some form or another. This can be done through any distance functions such as cosine or euclidean, the issue is that these functions are meaningless in environments with image inputs. A potential solution is to perform extreme max pooling of the pixels akin to how go-explore differentiates states [9], but this operation significantly reduces the information provided by the images. Even if there was a way to measure closeness in images, the effectiveness is limited. Video game benchmarks often use 'rooms', thus, in order to have a proper distance measurement, a policy has to put goals confined within the specific 'room'.

6.11 Conclusion

In this chapter we summarized a number of useful utilities that help alleviate some issues of exploration, temporal abstraction and by extension the credit assignment problem. Unfortunately, all the three problems remain open. In particular, exploration schemes are not *purposeful*. By that we mean that they do not explicitly attempt to explore in order to learn good policies as they also attempt to balance exploration with sample efficiency.

We also drew attention to a number of useful extensions to the experience replay. While the list is far from exhaustive, we believe that it is a good stepping stone before delving deeper into topic.

Last but not least, we introduced a few hierarchical algorithms in order to help build some intuition on the topic and as to how we understand it. We paid special attention to HAC as it introduces temporal abstraction in robotics which assists in solving the credit assignment problem. We also paid attention to Competitive Ensembles as it focuses on task decomposition, one of the main focal points of the thesis.

Chapter 7

Problem Decomposition and the Obstacle Tower

Through the previous chapters we introduced a number of *flat*[1] algorithms that are able to solve a variety of tasks ranging from Robotics to video games. While they exhibit impressive results and often super human performance [63, 13, 62, 48, 22, 83, 9], they are still unable to solve hard exploration problems with sparse rewards despite focused effort. In particular, [84] argues that bonus based exploration [48, 73] are not sufficient to properly explore the state action space, and instead are over optimized to the specific problem they were tasked with solving. The most common Hard Exploration benchmark is Montezuma's Revenge (MR) from the Arcade Learning Environment (ALE) [84, 1, 48].

The problem of exploration goes hand in hand with the credit assignment problem. In short, the credit assignment problem asks the agent to attribute the rewards it received to particular actions. These actions can be arbitrarily far from the time the agent received the reward. A modern example of this is in the DeepMind Behaviour Suite [19]. In the Behavior Suite, one of the tasks determines the reward on the first action taken, but, the reward is given a few hundred steps later.

[1] Flat is used to refer to non hierarchical agents.

7.1 Decomposition

A different approach to tackling both exploration and credit assignment is decomposing a task into sub tasks. Sub tasks are a very natural way of decomposing problems because it is similar to how we, humans, handle problems.

An intuitive way to think of sub tasks is to consider how *we* solve a particular problem. For example, if the task is to make a sandwich, we need to get the bread, cut it, place the condiments and the cheese, grill it, and then add tomato. All these are sub tasks or problems that need to be solved before solving the actual problem. These problems can be further decomposed, for example getting the bread requires moving our muscles. Thus, decomposition gives us a variable granularity over high level actions. This enables easier chaining of high, and low level actions, localized exploration, e.g. learn to optimally solve different sub tasks instead of solving the whole task, and remedies the credit assignment problem.

Sub tasks can be thought of in a more mathematical fashion that we believe helps in bridging the gap between intuition and practice. We can think of sub tasks as distinct regions of the state space that need to be observed sequentially, or sub spaces of the state space that are locally similar but reside in different regions. The first way to think of it resembles a sequence of high level actions. In the example above, the sub tasks that we outlined are these regions of space. The second way allows us to share experience and knowledge across these high level actions. In the above example, an example of shared knowledge is how to hold the utensils in order to cut the bread and the tomato, as well as how to spread mayonnaise with a knife. All these sub tasks share a common structure,

holding and using a knife and thus, all of them can be thought of as generalizations over that.

7.2 Goal based Problems

Many problems in robotics are defined in terms of goals. These goals reside in the observation space and when the agent observes the goal, the episode is considered a success. The most common approach to providing goals to a DRL agent is by concatenating the observation with the goal. An alternative for discrete observation problems is to expand the observation space to include the goal. The latter approach is most prominent in discrete problems where the observation space is finite. This approach results in increasing the observation space by a factor equal to the number of goals.

Problems that concatenate the goal and the observation effectively create a new observation space where the state is locally similar, but, due to the inclusion of the goals, they reside in different regions. HER effectively bridges the gap between the locally similar states and allows agents to generalize to new goals [30].

7.3 Mountain Car

In the Mountain Car problem, the agent controls a car that is positioned between two hills. The agent receives a negative reward when it expends energy to move the car. The agent receives a positive reward when it reaches the top of the right hill. The engine of the car is too weak for it to drive to the top of the hill. The only way to reach the top is by building momentum. Momentum is built by driving back and forth. Thus, the agent needs to expend energy to reach a high enough momentum and then ascend the hill using that momentum [85].

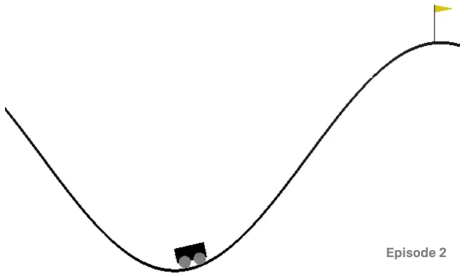

Figure 12: A rendering of the Mountain Car problem from OpenAI gym. The reward is given upon reaching the flag to the right.

The Mountain Car problem is one of the simplest and toughest exploration problems because it requires *directed* exploration. Dithering [20] exploration methods like $\epsilon - greedy$, or sampling from a $softmax/Boltzman$ [17] distribution are very unlikely to solve it. By directed exploration, we refer to attempting to exploring a subspace of the state action space with the purpose of finding good policies. Dithering techniques try to minimize the *negative* impact of selecting actions at random by selecting mostly optimal actions [17]. This however makes it very unlikely that a policy will perform the correct sequence of actions in order to explore a new subspace. This is particularly problematic in single step methods as they are highly biased [17], thus, they are unlikely to make a short term sacrifice for long term rewards [17, 20].

The Mountain Car in the OpenAI gym [3] has two versions, one with a discrete action space, and one with continuous action space. In the case of continuous action space, the problem can be solved with directed perturbations (noise) introduced in the actions, the most common approach to this, is through an Ornstein Uhlenbeck [86] process that generates noise that resembles Brownian motion [51]. Further on, a common approach for

single step algorithms is to delay the policy update until after collecting a large number of transitions [51, 50].

Many of the 'high performing' agents in OpenAI gym's leaderboard on the problem alter the reward function in order to solve the problem. We believe this is an unfair approach as it embeds knowledge and significantly alters the MDP.

An alternative approach that goes a long way in solving the problem is by decreasing the temporal granularity of the agents. This is done by repeating the action of the agent for multiple steps. This decrease in granularity makes it significantly easier to *stumble* by accident onto the solution.

The Mountain Car problem is easy to decompose into multiple steps, at the highest level, the decomposition is A) gain enough momentum, and B) ascend. From here, A can be further decomposed into A_1) partially ascend the right hill, and A_2) partially ascend the left hill. This decomposition is an example of 'nested' sub tasks, however, it can be flattened into A) partially ascend the right hill, B) partially ascend the left hill, and finally C) fully ascend the right hill.

7.4 The Taxi Problem

The Taxi problem [87] is a tabular example of nested hierarchies. In the Taxi problem there are 4 locations from which the agent | which operates a taxi | picks up a passenger and takes them to the destination location. The agent receives a -1 reward at every step, in the case of illegal pick ups and drop offs, the agent receives a -10 reward. The Taxi problem provides a state that encodes the location of the agent, the location of the passenger and the goal destination. The agent moves in a grid with walls, when the agent

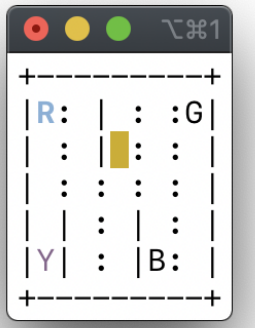

Figure 13: An example state of the taxi problem, the agent controls the car, orange, and needs to pick up the passenger from R, to Y.

attempts to drive towards a wall, it stays on the same location and takes a -1 penalty. Upon completing the trajectory, the agent receives a +20 reward.

The Taxi problem is effectively a goal problem, thus, there is a shared structure in how to operate. The agent does not need to relearn how to navigate to different spots in the grid when the passenger is in a different location, or the passenger needs to be dropped to a different location. The decomposition of the Taxi problem is trivial to decompose, a) reach the passenger, b) pick up the passenger, c) move passenger to goal, d) drop off the passenger.

7.5 Four Rooms

The Four Rooms domain is a classic goal based, tabular problem and provides the basis for many other procedurally generated environments. In short, the problem consists of 4 rooms each connected to two others via a hall. The agent can move north, south,

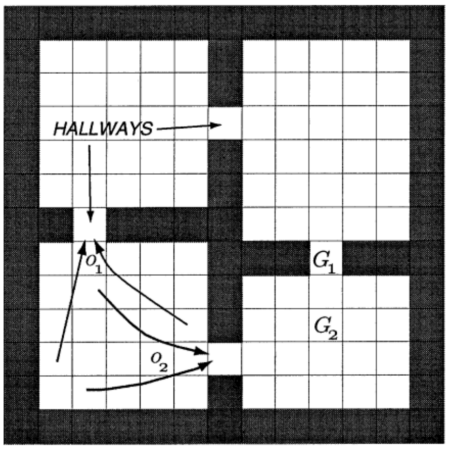

Figure 14: The four rooms domain. G_1 and G_2 are potential goals and O_1, O_2 are high level actions that take the agent to the respective hallway, source [88]

east and west. The actions are stochastic and fail 1/3rd of the time. On failure, one of the other actions is executed at random [88].

7.6 Sokoban

Sokoban is a family of difficult procedurally generated grid puzzles that require an agent to push boxes at top of target locations. The agent can push a box as long as the cell behind the box is empty and can move to empty cells that are above it, below it, to the left of it, or to the right of it. While Sokoban can be solved through classic search algorithms, the state space is enormous which makes search algorithms rather problematic. The Sokoban puzzles are MDPs and thus RL agents are good candidates, but, they need to commit to performing an action whereas search algorithms do not. This is rather problematic because it is possible to enter states where the puzzle becomes unsolvable. Such states occur when a box is pushed into a position that can not be moved afterwards. The Sokoban puzzle is a good candidate for planning and model based agents however, recent work has shown that strong inductive biases through architectures

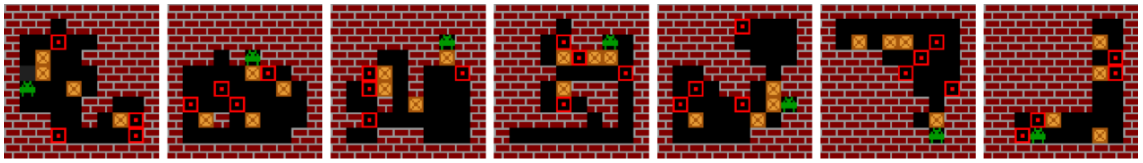

Figure 15: Some procedurally generated instances of the sokoban puzzle. The agent is represented by the green alien, the boxes are represented by the yellow squares and the target locations by the red squares. Source [93]

specifically designed for this task allow even model free algorithms to work [89, 90, 91, 92, 93, 94].

The Sokoban puzzle can be decomposed by separating the puzzle into sub puzzles or sub tasks. From here, we can further decompose the sub tasks into aligning the boxes and then pushing them to the correct spot. It is easy to see that there is some common structure if we consider different sub puzzles in isolation and in relative terms, i.e. two puzzles that need the box to be pushed upwards 5 times are identical even if they are located in different regions of the grid.

7.7 Obstacle Tower

The Obstacle Tower was the main motivation for this thesis, and therefore deserves an extensive analysis. In short the OT is a 3rd person, partially observable, procedurally generated, time attack, 3D platformer. The OT is composed of 100 floors, each floor is filled with puzzles, platforms and enemies. In order to beat a floor, the agent needs to enter the final door of the maze. Upon entering the final door, the agent receives the reward.

The OT is a time attack problem, meaning that the agent has a limited amount of steps available in order to solve it. It is procedurally generated which allows for a vast

number of different configurations that can be created deterministically. The problem is partially observable in the sense that the agent does not have access to the state of the floor. The problem is a platformer which refers to a class of games that include platforms and the player is required to traverse the platforms in order to beat them.

As a vision problem

The OT, unlike the aforementioned problems, is executed in a 3D environment and provides a picture from a third person point of view as the observation. Unlike atari games, the OT is not limited to just one art style. Instead, it includes up to five different visual themes that are applied to the floor layout. This forces the agent to learn how to extract features from all of them. While the different themes use different textures, the environment introduces different lighting conditions that significantly alter the colors of the environment.

As a generalization problem

The OT, unlike Atari games, is also a generalization problem. Atari games are mostly deterministic and while they may include some stochastic elements, it is entirely possible that an agent just learns how to navigate them. Agents that simply learn how to navigate an environment fail to solve slightly modified versions of said environments because they do not generalize, they memorize.

The OT uses procedurally generated mazes (floors) in order to prevent agents from memorizing a correct path and in turn, forces them to identify key components from the observations in order to generalize to the different layouts. Furthermore, the OT includes the ability to tweak the difficulty of the layouts by including cycles and branching.

As a planning problem

The OT incorporates a *continuous* version of the Sokoban puzzle. By continuous, we mean that unlike the original Sokoban, OT does not use cells and instead it uses x and y coordinates. It also extends the Sokoban puzzles by including a z axis, this allows for elevation, and even dropping the box under the platform.

As a control problem

The OT is a control problem because it requires high amount of movement fidelity in order to traverse the platforms, avoid enemies and enemy projectiles. Falling off platforms in the OT, or hitting enemies and their projectiles results in termination of the episode.

As a hard exploration problem

The OT is a hard exploration problem as the agent receives very little information regarding its objective. In fact, the only information it receives is a reward when it manages to beat a specific floor. This is particularly problematic when we take into consideration that the layout of the floors changes and the length of a solution increases with the difficulty of the floors. This makes the task of propagating the reward in the early stages of single step algorithms rather difficult.

Furthermore, higher levels in the OT include a noisy tv. As we mention in 6.5, the noisy TV problem can cause agents to get stuck in a situation where they constantly receive novel observations and therefore do not have an incentive to continue exploring because the intrinsic reward overpowers the extrinsic.

As a constrained information problem

The OT is an imperfect information problem for multiple reasons. First and foremost, the agent does not have access to all information about the current floor, it receives as much information as a camera can provide. The camera is tied to the character model that the agent operates and moves with it. The agent has access to, and can manipulate said camera by turning it around. Thus, the agent is not only in charge of navigating, it is also in charge of the information it receives.

Furthermore a third person point of view also includes a character model. The character model obstructs information about what is behind it and can result in radically different observations when the agent pans the camera to angles it is not used to seeing even though the environment's state is virtually identical. This forces the agent to be careful about how it manipulates the camera.

The inclusion of cycles and branching in the layouts further hinders the performance of the agents. Partially observable problems *require* some form of memory in order to construct an informative state. The inclusion of branches and cycles further increase the requirement for *informative* states as the agent will need to be aware of the cycles otherwise it falls into an infinite loop. However, because the OT is a time attack problem, an agent in an infinite loop will simply finish the episode and fail. This is particularly problematic for Monte Carlo methods since the agent's trajectory will be comprised of effectively the same set of observations.

When attempting to solve the Sokoban puzzles, the agent needs to stand behind a large box. Due to the third person perspective of the camera, the agent does not have access to a lot of information about its surroundings when pushing the box because it obstructs the

view. The situation is further hindered by the fact that when pushing the box, most of the observations are almost identical even if the actual state of the environment is different.

Chapter 8

Design and Implementation

In this chapter, we will outline our algorithm, the implementation and our design choices. As shown in [24], implementation details are important and should not be hand-waved, thus we will go through them. We begin by explaining the optimizations in PPO that allow it to attain good performance, and explain our extensions to make it work in the domain of the Obstacle Tower [21].

8.1 Proximal Policy Optimization

Our algorithm is based on PPO, and the overall algorithm is available in 5.5. While [22] focuses on the clipped objective:

$$L(s, a, \boldsymbol{\theta}_k, \boldsymbol{\theta}) = \min(\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)} \hat{A}_t^{GAE(\gamma,\lambda)}, Clip(\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_k}(a|s)}, 1-\epsilon, 1+\epsilon)\hat{A}_t^{GAE(\gamma,\lambda)}) \tag{89}$$

Further work argues that the objective is only one of the factors that give PPO its performance [95, 24]. More specifically [24] empirically shows that PPO's performance is a consequence of the following additional 'code optimizations' in the standard[1] codebase:

[1] By standard we refer to the code base provided by the authors of PPO: `https://github.com/openai/baselines`

1. Value function clipping. In [22], the suggested objective is

$$L_v = (V_{\boldsymbol{\theta}_k} - V_{target})^2$$

but the standard implementation provided by the authors of [22] instead fits the value network with a PPO-like objective:

$$L_v = \min \left[(V_{\boldsymbol{\theta}_k} - V_{target})^2, (Clip(V_{\boldsymbol{\theta}_k}, V_{\boldsymbol{\theta}_{k-1}} - \epsilon, V_{\boldsymbol{\theta}_{k-1}} + \epsilon) - V_{target})^2) \right]$$

where $V_{\boldsymbol{\theta}}$ is clipped around the previous value estimates and ϵ is the same as the one in [89].

2. Reward Scaling and Clipping. The standard implementation clips the rewards within a certain range $[-10, 10]$ and divides them with the standard deviation of a rolling discounted sum of the rewards.

3. Scaled Orthogonal initialization. Depending on the framework used to implement neural networks, the weights are initialized with different schemes, usually Xavier [96]. The standard implementation uses an Orthogonal initialization that is scaled on a per layer basis.

4. Learning Rate annealing.

5. Observation Normalization and Clipping. The observation is normalized using the running mean and standard deviation of the matrix. The observations are then clipped to the range $[-10, 10]$. Note, in Atari-like problems, the observations are tensors of uint8 values in the range [0,255]. The common approach is to normalize to [0,1] range by dividing by 255, however, this is not necessary here.

6. Usage of hyperbolic tangent activation over ReLu [35].

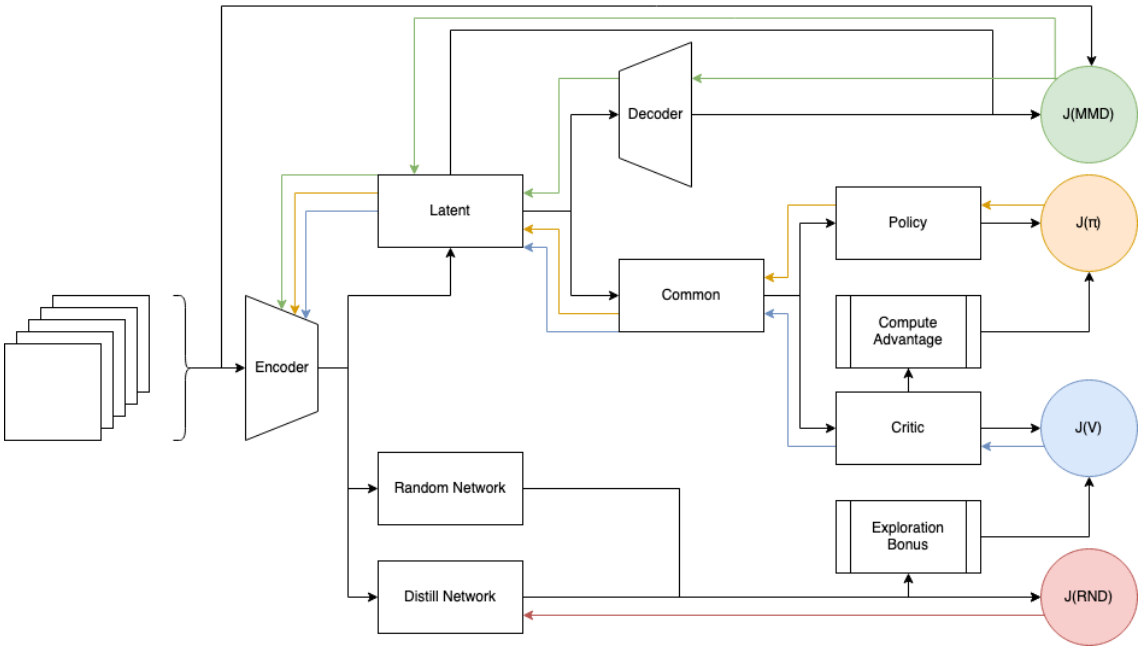

Figure 16: The black lines show the forward propagation of the input. The blue lines show the propagation of the gradients of the critic. The green lines show the propagation of the gradients of the MMD-VAE, the red lines show the propagation of the gradients of the RND, and the orange lines show the propagation of the gradients of the policy.

7. Global gradient clipping (L2 norm) does not exceed 0.5.

8.2 Architecture and Modules

Building up from the work of [97], we include a deterministic auto-encoder using MMD-VAE [98] to help in learning better representations of the observations. As the Obstacle Tower is a partially observable problem [21], we incorporate a GRU cell [35, 34] in order to learn temporarily extended relationships [36]. Due to the inclusion of a GRU cell, we alter the architecture described in [97] by adding the cell after the encoder as evidence suggests it provides better encoding of the state [36].

Agents with *informative* representations can learn faster and more robust policies [99, 97, 100, 28]. In order to achieve better representations, agents use auxiliary tasks

(objectives) in order to get more informative gradients. Some ways this can be achieved is through dynamics predictions, inverse dynamics predictions, immediate reward predictions and observation reconstruction [99, 101]. The usage of AutoEncoder modules has been studied in a number of ways [97, 102, 101, 99, 103, 100] and the conclusion drawn from [97] is that stochastic models are brittle, unstable, and can interfere with the policy. Thus, we employ a MMD-VAE [98] which is deterministic, and exhibits properties similar to $\beta - VAE$ [56], VAE [104], and Regularized AE (RAE) [105], which is low dimensional, stable, and provides disentangled representations.

Further more, due to the nature of the problem, we require a form of structured exploration [21]. For this reason, we include an RND Module [48] in order to facilitate exploration. In addition, we treat the RND as an auxiliary task for the agent. However, since the objective function for the RND is $||\hat{f}(s) - f(s)||_2^2$ and can result in high values, we include a hyper parameter as in [99]. The hyper parameter is used to weigh the effect of the gradients from the RND since it is a secondary goal.

We give illustrate our implementation in 16. One potential modification in the architecture is to stop the propagation of gradients from the actor (orange) from flowing to the shared layers of the critic and actor. This is done in [103] and their experimental findings is that it helps improve the stability of the policy, but the modification was done in SAC and not PPO and the critic and actor networks did not share any layers.

8.3 Obstacle Tower Environment

The Obstacle Tower is implemented with the Unity game engine in C#, therefore it is necessary to have some form of communication interface between the game engine and the agents. This is achieved by having the Obstacle Tower also run a server application.

The server listens to the instructed port for inputs (actions), performs the desired actions and responds back with the observation, the reward, whether the episode has finished and some additional information. In order to communicate with the Obstacle Tower, we implemented an Application Programming Interface (API) based on the one provided in [21] that is compatible with the Gym API[3] and the Stable-Baselines project, the source code is available in C.3.

The recent release of StableBaselines (SB), more specifically version StableBaselines3 (SB3) introduced some breaking changes that changed the available wrappers. Wrappers are objects that alter the behavior of the environment, e.g. they can alter the information presented in the observations and even alter the reward signal.

The StableBaselines3 project makes use of wrappers from [3] that make an environment compliant with suggestions from [106]. In order to make our environment compliant with the wrappers used by SB3, we implemented an object that communicates with Unity environments and emulates an ALE [1] environment. The emulator is not tied to the OT can be used with other environments. The source is available in Appendix C.1.1.

8.3.1 Action Space Modifications

We perform a number of modifications in the action-space to make the problem easier for the agent. In particular, we reduce the action-space from a multi-discrete with $[3, 3, 2, 3]$ actions down to just discrete 12 through the following modifications:

- We remove the option of walking backwards. This change does not reduce the agent's ability to navigate the environment as walking backwards can be achieved by rotating 180 degrees. This modification reduces by $\frac{1}{3}$ the action space to 36 actions.

- We tie the camera movement i.e. camera pan to the agent's direction. This means that the camera pans along with the agent. This change reduces the available actions to just 12.

- We change the Multi-Discrete action-space to plain discrete as there is no significant difference between the two [107] and makes the implementation easier.

We implement these modifications through a family of objects that implement the ActionWrapper interface. Our implementation computes the mapping between the actions before and after the modification, then it intercepts the action provided to the environment and alters it to the correct one. The code for the modifications is available in Appendix C.1.3.

Another potential change in the ActionSpace is to use continuous actions and then discretize them. This approach would be ideal in environments with analog inputs and motion but not as much in OT. Although we have not tested it, we believe that this change removes the semantics behind the actions and introduces unnecessary complexity due to mixed spaces. The other issue is that it, in a sense, removes the notion of entropy for the continuous action. We can have a highly entropic continuous action that when discretized, it becomes deterministic. If we discretize continuous values from [-1, 1] to two discrete actions for [-1, 0] and (0, 1], a policy that creates a Gaussian Distribution may put the mean way too far above 1, and the log standard deviation to be large, but not large enough to get values in [-1, 0] range, thus thee action is deterministic when discretized.

8.4 Current State

Implementing DRL algorithms is both a difficult, and a tedious process. Unlike other types of programs, DRL algorithms fail silently and take an excruciating amount of time

to validate. In particular, DRL algorithms fail silently due to logical errors in the mathematics behind the algorithm, tensor operations that result in incorrect structures and therefore gradients. Furthermore, validation is difficult as we need to perform a large number of tests with different seeds to verify the validity of the algorithm before we are able to look for appropriate hyper-parameters.

8.4.1 Stable Baselines

We began this project using StableBaselines v2 as the project provides a number of high quality implementations with a *stable* API [108]. Our implementations were dependent on the Stable Baselines project, as we reused existing source code. Besides borrowing code, we made a few contributions to the project.

We implemented a minor extension to replay buffers that allowed agents to store a large number of transitions at the same time instead of a single one, this was implemented in commit $109e12a$. Furthermore, we made a few bug and consistency fixes in the commits. More specifically, we corrected a model initialization inconsistency between SAC and TD3/DDPG in commit $ac92d2e$, we fixed an inconsistency between environment wrapper objects in commit $d2364c9$, and last but not least, we corrected a bug that prevented models from resuming the training process in commit $ae4f6c5$. Last but not least, we implemented a module that allows for Noise Processes with different lifespans, this module makes it easier to run multiple environments in parallel without multiprocessing. The stacked noise module was added in commit $78e8d40$ of the SB3 repository and the source is available in Appendix C.4.1

The SB with version 3 has moved on to pytorch, with this change, a large volume of our source code was obsolete. Our implementation was very similar to the SB3 implementation

of PPO, but we chose to migrate to SB3 as it has been extensively tested, validated and used in published work and because it altered a number of tools that we used. Our current implementation builds on-top of SB3 PPO and includes the modules that we mention above, except for the GRU layer, in its place, we use frame stacking[2] which does not introduce additional complexity. Due to the partial observability of the problem, and in particular when cycles exist in the room layout, a large number of frames need to be stacked. We suggest at least 20.

8.4.2 PPO

We provide an implementation of PPO based on the one provided by SB3 in Appendix C.2. The SB3 library separates the algorithm and the underlying network that selects the actions. For example, PPO and A2C use the same underlying networks and action selection methods, the only difference is the optimization processes, which is algorithm specific. We use the same philosophy, we extend the existing 'policy' objects with an RND module. More specifically, we added two additional networks after the feature extraction part of the model, i.e. either after the CNN or, for non image tasks, right in the beginning. We alter the 'policy' object to also provide the RND error along with the action selection, the source code is available in Appendix C.2.1. The RND prediction error is summed onto the current reward, thus, we also altered the algorithm part of PPO, and the source is available in Appendix C.2.2. Our PPO implementation also includes Generalized State Dependent Exploration (gSDE) as that is a core feature of the SB3 [109].

We provide three different 'policy' classes, 'RndPolicy','CnnRndPolicy' and 'AERndPolicy'. RndPolicy implements the RND modules and action selection logic. The action

[2]Frame stacking provides a history of the last n frames provided by the environment.

selection logic is derived from SB3's 'PPOPolicy' class. The CnnRndPolicy class extends the 'RndPolicy' class and adds a CNN feature extractor. The AERndPolicy class extends the CnnRndPolicy and adds a few modifications in order to use an AutoEncoder module. The source code for all policy classes is available in Appendix C.2.1

In SB3, the feature extraction is performed through a family of objects that extend 'BaseFeaturesExtractor'. In order to add support for the MMD-Vae, we create the 'NatureAE'. NatureAE is a feature-extractor class that also implements an AutoEncoder. More specifically, it uses an encoder with the CNN provided in [2], which is referred to as 'NatureCnn', and a decoder that is symmetric to the encoder. We then derive the NatureAE to create NatureMMD which implements an MMD-Vae. The code for NatureAE and NatureMMD is available in Appendix C.3.1.

8.4.3 Current Limitations

Due to lack of computational resources, we are unable to test and fine tune our model in Atari Games or the OT. One of the main bottlenecks is the inclusion of MMD-VAE. While MMD-VAE allows us to significantly reduce the size of the latent dimension, it also introduces a large bottleneck in the form of Gaussian Kernels. In particular, we need to compute a non trivial number of kernels in order to get a good loss for the latent variable, however, this can be alleviated using a dedicated processor for tensor products [3] .

Our implementation also lacks a memory module, eg. GRU which severely hinders performance in POMDPs. While this can be partially alleviated using a frame buffer/stacked frames, certain papers have shown that Recurrent Networks can learn significantly

[3] A Graphics Processing Unit (GPU) or a Tensor Processing Unit (TPU).

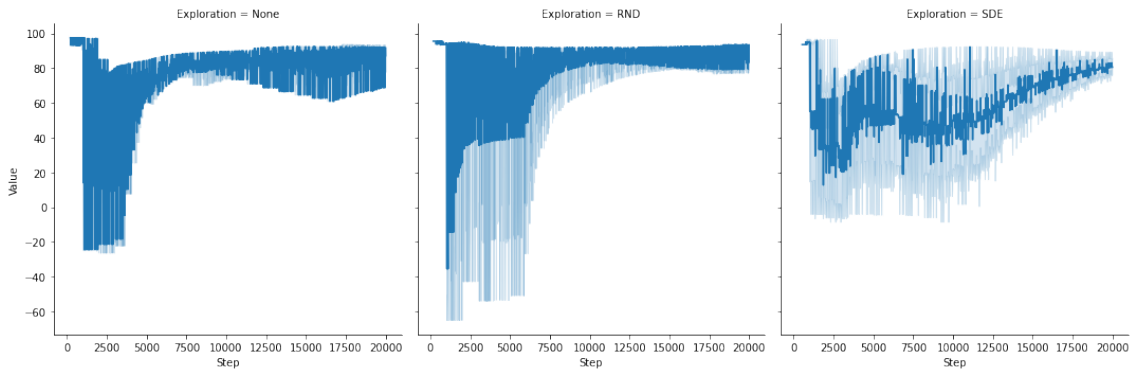

Figure 17: Performance of our implementation with different kinds of Exploration.

better representations over non recurrent ones even in MDPs [36] and that RNNs may be crucial to certain problems [7, 8].

8.4.4 Preliminary Results

We made some preliminary experiments with our implementation on the MountainCar (Continuous Version) [3]. The results are shown in figure 17. Unfortunately MountainCar does not allow RND and gSDE to show their full strength as it is a relatively easy problem to solve. The nature of PPO, i.e. take a series of steps before optimizing along with the sampling nature allows PPO to reach the terminal state faster than other algorithms, but RND appears to help. We performed 9 experiments with 9 fixed seeds and display the performance of the 5 runs with the best final performance.

8.4.5 Computational Resources

Atari-like environments are usually run for 100-200 million timesteps with multiple seeds and domain specific hyper parameter tuning. On-policy policy gradients such as PPO are run for 200 million timesteps. Fortunately, the OT and PPO can run multiple environments in parallel. We tried to run our implementation on an 8 core workstation

with a gtx 1080ti and the StableBaselines 2's implementation of PPO without any modifications. The workstation needed approximately 1 hour to compute 200 thousand steps, thus it will take roughly 42 days to complete 200M steps. If we are to assume perfect strong scaling, then with 32 cores, the process will take approximately 10.5 days. Thus, in order to have a *general* algorithm without any form of domain knowledge learn how to behave in a reasonable amount of time, a cluster of 128 cores and 16 GPUs of 1080ti class or greater are the bare minimum. The GPUs are necessary in order to run the environment and we believe that 128 cores running 4 instances of the environment each will saturate a single GPU, thus we suggest running 16 of them, so each GPU can run up to 35 instances of the environment and leave 1 GPU dedicated to computing the forward and backward passes.

Chapter 9

Ghosts in the Tensors

Adaptive programs are guilty of hiding nuances and are ridden with pitfalls, both large and small; thus, they often exhibit chaotic and undesired behavior. This chapter is an attempt to enumerate some of these nuances and provide a mental toolkit to help understand and diagnose unexpected behavior.

9.1 Fantastic deltas and how to fight them

Let us assume that we have stumbled on some optimal policy π in some fully observable and deterministic environment ε, with state-space $\mathcal{S}_\varepsilon$ and action-space $\mathcal{A}_\varepsilon$. A logical first assumption about π is that it is also deterministic, otherwise it would not have been optimal because the probability of making a suboptimal decision would be non zero.

While in the common case a stochastic policy is suboptimal, it is not always true. Suppose that some state $s \in \mathcal{S}_\varepsilon$ exists which acts as a sink, and thus, for every action $a \in \mathcal{A}$, $p(s|s,a) = 1$. In this case, every action is optimal so a stochastic policy can be optimal iff it is also optimal in every other state. The question now becomes, is it possible to have a stochastic function that is almost always optimal even in non sink states?

The answer is a solid maybe. It depends on the function approximator we use. Because neural networks are universal approximators, they can also approximate Dirac δ functions or, with some help, Kronecker δ_{ij} functions. In the continuous action-space, an optimal and deterministic policy can be thought of as an estimator of a Dirac δ function centered at the optimal value and is used as a probability density function. In a discrete action-space, an optimal policy would behave like a Kronecker δ_{ij} function, where the optimal action has probability 1, and every other action has probability 0. In consequence, a function approximator that can behave like a delta function can result in an almost deterministic policy.

Although deterministic and optimal functions are desirable and often the end goal, suboptimal deterministic functions on their own are not, as they can fall into local optima. Although local optima can be stable, and the algorithms can behave in a predictable manner, deterministic policies do not allow us to perform sufficient exploration which prohibits us from learning an optimal policy. In addition, stochastic policies can also degenerate to deterministic behavior and fall into local optima as well.

The degeneration of a NN into a delta function in the discrete domain in policy gradient methods can be solved through the addition of an entropy term in the loss function.

$$\nabla_{\boldsymbol{\theta}} J = \mathbb{E}_{\mathcal{T}\sim\pi_{\boldsymbol{\theta}}}\left[\sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s_t) A^{\pi_{\boldsymbol{\theta}}}(s_t, a_t) + \beta H(A_t|\pi_{\boldsymbol{\theta}}, s_t)\right] \tag{90}$$

$$= \mathbb{E}_{\mathcal{T}\sim\pi_{\boldsymbol{\theta}}}\left[\sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s_t) A^{\pi_{\boldsymbol{\theta}}}(s_t, a_t) + \beta \sum_{a'\in\mathcal{A}} \pi_{\boldsymbol{\theta}}(a'|s_t) \log \pi_{\boldsymbol{\theta}}(a'|s_t)\right] \tag{91}$$

We added two terms, the Shannon entropy on the selected action A_t conditioned on the current policy $\pi_{\boldsymbol{\theta}}$ and the current state s_t, and a hyper parameter, β, which is often referred to as the entropy loss factor, together, they are referred to as entropy loss. The entropy loss factor is used to *regularize* the effect of entropy and is usually a small number

close to zero, eg. $1e-4$. It is paramount that we understand the effect of entropy loss on our objective function. Since we are performing *gradient ascent* on the whole objective function, the optimitzation algorithms will attempt to increase the final objective. Thus, a very high entropy loss factor can end up dominating the advantage loss and the agent will fall into a local minima.

9.2 Actor Collapse

A common practice in Actor-Critic agents that use *first order* optimization methods such as Adam, SGD, RMSProp and so on, is to share the initial input weights between the critic and the actor, and then separate them. The idea is that the model learns to extract useful features from the state that are useful to both the actor and the critic. In addition, weight sharing greatly reduces the number of parameters in a model which results in less computation and greater memory efficiency.

If we compute the backward pass for either the critic, or the actor, and apply the gradients, the optimizer will release the gradients. In consequence, when we attempt to do a backward pass on the other model, we will have to recompute both the forward and backward pass again, as the shared gradients were released already.

The common practice is to sum the objectives of the two models and perform gradient ascent or descent depending on the case. Thus, the objective function becomes:

$$J(\pi_{\boldsymbol{\theta}}, Q_\phi) = J_\pi(\pi_{\boldsymbol{\theta}}) + J_Q(Q_\phi|\pi_{\boldsymbol{\theta}}) \tag{92}$$

Where $J_\pi(\pi_{\boldsymbol{\theta}})$ is the objective function of the policy network, and $J_Q(Q_\phi)$ is the objective function of the critic network. Note that the objective function of the policy network is usually eq. 91, which includes the entropy loss.

Although it is not very obvious at first, summing the two objective functions has catastrophic consequences as the two objective functions may end up having losses in different orders of magnitude. Recall that the objective function for a critic is usually either of:

$$J_Q(Q_\phi|\pi_{\boldsymbol{\theta}}) = \mathbb{E}_{\mathcal{T}\sim\pi_{\boldsymbol{\theta}}}\left[T^{-1}\sum_{t=0}^{T}(r+\gamma(1-d)Q(s_{t+1},\pi_{\boldsymbol{\theta}}(s_{t+1}))-Q(s_t,a_t))^2\right] \tag{93}$$

$$J_V(V_\phi|\pi_{\boldsymbol{\theta}}) = \mathbb{E}_{\mathcal{T}\sim\pi_{\boldsymbol{\theta}}}\left[T^{-1}\sum_{t=0}^{T}(r+\gamma(1-d)V^{\pi_{\boldsymbol{\theta}}}(s_{t+1})-V^{\pi_{\boldsymbol{\theta}}}(s_t))^2\right] \tag{94}$$

Where the former is the objective function used by the critic in deterministic policy gradients, and the latter is used by stochastic policy gradients, both are, in essence the Temporal Difference Residual Error squared. The term d is a convention, represents whether s_{t+1} is a terminal state and is a boolean, by convention $1-d=0$ if d is true, and $1-d=1$ if d is false.

The problem with summing the critic and actor objectives is that both of the critic objectives shown above may be in different order of magnitude than objective function of the policy $J_\pi(\pi_{\boldsymbol{\theta}})$. Thus, the weight of the critic's loss collapses the policy by changing the weights of the common layers in the direction that causes the critic's loss to be less, not in a way that increases the policy's performance. In consequence, the objective is minimized without actually solving the problem. To deal with critic collapse, we introduce another term, α in 92, that acts as a regularization factor.

$$J(\pi_{\boldsymbol{\theta}}, Q_\phi) = J_\pi(\pi_{\boldsymbol{\theta}}) + \alpha J_Q(Q_\phi|\pi_{\boldsymbol{\theta}}) \tag{95}$$

Alternatively, instead of reducing the mean squared error, we can reduce the mean smooth L1 error, or Huber Loss:

$$L_{1,\text{smooth}}(Y, \hat{Y}) \doteq \begin{cases} |Y - \hat{Y}| & \text{if } |Y - \hat{Y}| > \alpha \\ \frac{(Y-\hat{Y})^2}{|\alpha|} & \text{if } |Y - \hat{Y}| <= \alpha \end{cases} \tag{96}$$

This kind of collapse is only possible in A2C and derivatives, e.g. PPO, and the reason is that they share a common core of layers. We can completely resolve this issue by simply having the two networks be independent. The reason being that the computation graph of the two losses is independent besides the addition, and the gradient of the addition of a loss function with respect to the other loss function is 0 (iff the graphs are independent).

9.3 Off Policy Over-Optimization

On policy algorithms like A2C, with the exception of PPO [22], perform a single optimization step on a collection of trajectories whereas off policy algorithms like DQN, DDPG and TD3 can perform multiple optimization steps per step in the environment (note that PPO also performs multiple optimization steps on a collection of trajectories).

Because off-policy algorithms (and PPO) can perform multiple optimization steps, they tend to have better sample efficiency over on-policy algorithms at the cost of additional hyper parameter tuning. A direct consequence of performing multiple optimization steps is that a model that learns *quickly*, will fall into local optima due to insufficient exploration [19].

To combat the issue of over-fitting on few samples or early converging is through a set of orthogonal methods, a) the addition of noise in parameters, b) addition of noise in actions, c) delaying optimization until a large number of experiences has been collected, and d) uniform sampling from the action-space in early stages in order to diversify the

observations [46]. Note that not all methods are applicable in all problems. For discrete problems, we can use d, but not b, for continuous problems, we can use both b and d, but d is uncommon.

Noise addition in the network parameters is done by sampling noise from a Gaussian distribution and perturbing the parameters. This technique allows the model to perform better exploration by disallowing the convergence of the policy. Adding noise to actions can be done by sampling from a Gaussian distribution, or by simulating an Ornstein-Uhlenbeck process, in practice sampling from a Gaussian performs better. Delaying optimization is done by ensuring that the replay buffer used by the algorithm is sufficiently large and uniform sampling on the action-space is done until a number of steps have been performed [49, 51, 50].

9.4 Proximal Policy Over-Optimization

PPO has a number of sources of bias that are rooted in PPO 's nature of multiple optimization steps [22]. As stated in 9.3, off-policy algorithms suffer by over-fitting on old experiences, in contrast, PPO can suffer from over-fitting on current experiences.

Consider the following two trajectories: a) The agent receives -0.1 reward on every step for N steps, b) The agent receives -10 reward on every step for N steps as well. Assume that we are using GAE [51] to compute the return. Because we are using normalization to scale the return of each trajectory to a Gaussian distribution centered at 0 in order to avoid blowing up the gradients, both of the trajectories will result in the same values despite the latter being objectively worse. The result is that the two trajectories will be considered equivalent and thus, the weight update can be catastrophic, this is further exacerbated when we consider that the optimization steps are more than one, thus it is

important that we are conservative with the amount of optimization steps when using PPO.

The issue of adjusted rewards can be mitigated by keeping an average over the rewards collected across concurrent trajectories. We can keep the mean across the trajectories the same, but reduce the variance. Altering the mean of the rewards alters an agent's will to live [22]. However, removing some variance is beneficial.

9.5 Diluted Experience

A common approach to gathering experience, or, rollouts is to have a number of worker threads, that simply run the policy and collect instances of $\langle s, a, r, s', d \rangle$ where s, a, r, s', d are state, action, reward, new state, and terminal respectively. Parallel workers allow us to better utilize the available machines and hardware.

The general procedure of collecting rollouts is to simply run the policy for N steps, then update the policy or store the rollout in an Experience Replay Buffer. Although collecting N steps works just fine in environments with dense rewards, they can back-fine in sparse rewards.

To illustrate the issue, let's assume that we have an environment that gives 100 upon reaching the goal and in the general case $[-0.03, 0]$ depending on the action. Let's also assume that reaching the goal is also rare, but it is terminal. The environment terminates on either 100 steps or, upon reaching the goal. If the environment terminates, we will reset and continue collecting experience until we reach the desired rollout steps.

Since the goal is rare, if we were to always collect, say 100 steps, even if we end up reaching the goal, we will continue collecting experience, thus, the probability of choosing the goal experience during the sampling process is lower than if we just stopped on observing the goal and collected a shorter rollout.

9.6 Minding your business

DDPG and derivatives are based on the idea that we can use a Q function as any other generic function that can be optimized, i.e. minimized or maximized [51, 50]. Thus, we use $Q : \mathcal{S} \times \mathcal{R} \rightarrow R$ to teach the policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ how to maximize Q on a given state S.

When we define the objective function $J_{\boldsymbol{\theta},\phi,\ldots}$ for DDPG and derivatives, we define an objective function that returns a *tuple* of losses. If we were to join the tuples through summation or multiplication, we end up with the issues discussed in 9.2, where one of the network losses dominates the loss function, in addition, it is not necessary to join them as the networks that need to be optimized are not involved with one another.

In DDPG and derivatives, the optimization of the policy network is *delayed* until we have performed at least 1 optimization step on the Q network [51, 50]. in the case of TD3 [50], we are not only optimizing two Q networks to stabilize them, we are also updating the policy after a number of optimization steps on the Q functions. A naive approach to optimizing multiple networks is to create a single optimizer i.e. an object that performs a form of optimization on the networks for every network. The correctness of this approach is dependent on the underlying algorithm and its implementation.

An optimization algorithm like Adam[52], keeps track of first and second order moment estimations to accelerate learning, this approach though comes with some caveats. Adam can not use the moment estimation from early steps due to the high amount of *bias* it

introduces, therefore, it employs a *de-biasing* mechanism based on the number of steps the algorithm has performed. In consequence, using a single optimizer object can cause stability issues if the algorithm's number of steps is inconsistent with the actual number of optimization steps performed on the parameters.

Another issue that arises when using a single optimizer object is that because the optimization objective uses two networks, makes it possible to accumulate gradients on the critic networks while training the actor network, thus performing an update step on the actor also changes the critic's weights causing catastrophic interference. In fact, performing gradient ascent on the result of the critic introduces over estimation which is one of the problems that plague DDPG. Over estimation occurs because we take the negative of the estimation and perform gradient descent.

9.7 Handling Termination

An important workflow change introduced in A2C [82] was using a single agent to run multiple instances of the environment instead of each environment to run on their own. This process is referred to as running a *Batched Environment* (or *Vectorized Environment*). This change allowed for much better resource utilization of the underlying hardware as more trajectories could be collected in parallel which results in greater *diversity* among the training samples, all while remaining an *on-policy* algorithm.

Although there are many ways to implement Batched Environments, the general idea is that a shell environment receives a list of actions which are then propagated accordingly, the results are collected into lists or matrices and are propagated back to the agent. An issue that arises in this scenario is how to handle the terminal cases. In RL theory, we treat episodic environments as infinite horizon problems where every terminal state always

transitions to itself with reward 0. In the case of a single, non vectorized environment, the process is simple, when the state is terminal, return it, and reset the environment to start over. The trajectory has a clear boundary.

When implementing batched environments where the agent may perform more steps than a trajectory, the question that arises is, how should we deal with termination? The agent needs to be informed of both the terminal state, and the new state. The solution here is that we replace the terminal state with the new state, but we signal to the agent that the new state is terminal. Although this approach does not appear very sound, it is based on the idea that if the state is terminal, we do not *actually* need to use it, thus, even if it is filled with garbage, it does not bother the algorithm. This is obvious when we consider the δ_t error:

$$\delta_t = r + \gamma(1 - d)Q(s', a') - Q(s)$$

Because of the $(1 - d)$ term, the state s', if terminal, becomes irrelevant. This occurs by the convention that if $d = True$, then it evaluates to 1 and 0 otherwise. This allows us to replace the terminal state with the new state without introducing issues to the agent when computing the return.

9.8 Dead neurons and misleading gradients

As mentioned in 2.4, dropout is used as a regularization technique and to facilitate generalization [19, 45]. The use of dropout, while useful, is particularly problematic when we consider on-policy algorithms and the behavior is entirely dependent on the workflow of the training process. A common approach is to have a number of worker processes that run inference on the model to collect experience. When we run inference on a model, we

avoid computing gradients and instead send the collected experience to a procedure that performs the optimization step.

In the scenario above, the optimization procedure goes through the collected experience and computes the gradients. The issue here is that because drop out is stochastic, the optimization procedure will compute different results to those that resulted in the collected trajectory. In consequence, we are training an on-policy algorithm with experience gathered from a different policy. It is evident that in this scenario the optimization procedure will fail to properly fit the model [17].

9.9 Sawtooth

Off-policy agents can perform a large number of optimization steps with experience collected from other policies. In consequence, they are usually more sample efficient than on-policy algorithms at the cost of stability [17]. In DRL applications, the instability is further exacerbated by the fact that we are using function approximators and often bootstrap from their results.

Off Policy algorithms can exhibit performance that resembles a *sawtooth*, where the algorithm improves in performance until it reaches a peak, and then drops sharply to random or worse. During the improvement the agent may exhibit a decrease in the rate of improvement. This behavior is referred to as *Catastrophic Forgetting* [44]. Catastrophic Forgetting in DRL does not have a ubiquitous solution, however, a number of useful tools exist to handle it [110].

Keeping a logbook

The first, and most obvious solution is to keep a history of backups of the models and when the optimization process shows signs of catastrophic forgetting to explicitly drop the current model and use an older one. This comes with some caveats.

First and foremost, this approach increases the computational cost in order to deduce that catastrophic forgetting has indeed occurred, and that the model was not *unlucky* due to the seed, thus, more evaluations need to be done.

Second, drops in performance, are not always the result of catastrophic forgetting. To showcase this, consider the *Mountain Car* problem. In this environment, the agent controls a car at the bottom of a valley and needs to reach the top. In order to do that however, it needs to build up momentum by going towards the direction opposite of the goal. The agent receives a penalty based on the effort the car exerted, the more the effort, the bigger the penalty. However, if it reaches the top, it receives a very large reward.

Without good exploration, the agent will learn that the optimal behavior is to do nothing, and thus exert no effort and get a reward very close to 0. With good exploration, the agent learns that it needs to get to the top by driving towards the opposite direction, incurring a short term penalty and then going towards the goal as using its momentum.

In consequence, it is possible to mistake learning how to solve a task with catastrophic forgetting because the solution is requires an upfront penalty. Thus, identifying catastrophic forgetting solely through the changes in the returns of an agent may be a mistake and depends on the environment.

Learning Rate Annealing, Gradient Clipping, Gradient Norm Scaling

Learning rate annealing is a common approach used in all forms of DL. In the DRL case, we often have access to a reward threshold, i.e. the reward the agent needs to achieve. Using the reward threshold we can construct learning rate scheduling mechanisms that get diminished as the agent's performance increases, thus reducing the odds of catastrophically forgetting.

Gradient clipping is another way of minimizing the effect of large prediction errors. More specifically, after we computing the gradients using backprop, we clamp them into a specified range. The most common starting point is $[-\sqrt{5}, \sqrt{5}]$. Gradient clipping is particularly useful in RMSProp and Adam. The two methods keep a running average of the magnitude of the gradient for particular weights which, allows them to accelerate learning. However, exactly because they keep track of the magnitude, they can cause very large shifts when the estimated gradient is biased, thus clipping can help reduce the size of the change. Alternatively, we can reduce limit the global norm of the gradient to 0.5 as done in [22, 24, 95].

Identifying Catastrophic Forgetting

As we mentioned above, it is not always possible to identify catastrophic forgetting through the returns alone, however, depending on the algorithm, different tools can be used to diagnose this.

In the case of *deterministic* actor critic methods, such as DDPG [51], TD3 [50], and derivatives, we can use the critic network(s) to estimate the performance of the current policy. More specifically, we can use the critic to estimate the values of the policy after

optimizing and compare with the previous ones. The caveat here is that the critic may be wrong in its estimations.

In the case of *stochastic* policy gradient algorithms like A2C and A3C [82], PPO[22], we can use the fact that the policy creates a distribution of the actions and measure the KL Divergence between the actions before, and after the changes in the policy. In particular, PPO often exhibits a peak in KL Divergence after the updates and then catastrophically forgets [95, 24].

9.10 Time is but a stubborn illusion

In DRL implementations, the agent-environment interaction is done in two steps, the agent performs an action, and the environment responds with a new observation, a reward, a termination signal, and diagnostic information. We could like to draw attention to the termination signal. A common practice is to include a time-limit to prevent the agent from getting stuck in uninformative areas of the environment. When the time-limit is reached, the environment communicates through the done signal that the environment has terminated. This is problematic if the time-limit is not communicated back to the agent.

If the time-limit is *not* given to the agent, but the environment terminates, then the agent learns a different MDP than the true because it believes that a non terminal state is terminal, simply because the time was not provided. We can correct the behavior through by, a) including the time in the observation space, or b) by altering the done signal and accounting for the end of the trajectory by resetting the environment.

Chapter 10

Conclusion

Traditionally, a conclusion chapter is meant to provide answers for the questions posed in the introduction through support from experiments, sum-up insights and provide the grounds for future work. Due to the trajectory of the thesis, we will focus on our findings in our attempts at implementing a *general* solution, and outline our thoughts on potential ways for future work.

This thesis began as an attempt at finding a general, hierarchical approach that could learn and solve the Obstacle Tower. Throughout the process we studied a number of algorithms, flat and hierarchical in an attempt to identify their shortcomings and potential extensions or improvements. However, after attempting to implement extensions and verify the validity of our models, we concluded that the task was a Herculean one.

Reinforcement Learning is a unique field in many regards. In its core, Reinforcement Learning is learning how to approximately solve a shortest path problem with extra steps and, in a sense, it is the processes of learning a guess from a guess. Unlike other '* Learning' fields, RL does not have the I.I.D property, in-fact, RL lacks it by its very definition, and thus, learning is difficult.

In this thesis we put emphasis on hard exploration problems, and decomposable problems to provide a motivation for hierarchical algorithms. We provide a summary of common model-free algorithms, tools and extensions that, in our opinion, provide the necessary stepping stone towards hierarchical algorithms. In addition, we provide the structure, a partial implementation, and the reasoning behind our solution for the Obstacle Tower. Last but not least, we provide a mental toolbox to help other researchers diagnose problems with their implementations.

10.1 Future Work

While this thesis was concerned with partially observable, hard exploration problems, there are many other open topics in the field that should be first solved before we venture into more complicated solutions such as hierarchical algorithms, after all, Occam's razor seems to repeatedly show up and we are guilty of ignoring it.

We believe that the first order of business is to understand the relationship between different initialization procedures. Starting from randomly initialized Neural Networks results in low inductive bias that hinders the network's ability to learn [2, 97]. At the same time, recent work has shown that in a randomly initialized Neural Network, multiple subnetworks exist with very strong inductive bias and can achieve equivalent, or even better performance than the whole network [111, 112].

The second area of interest is deciding and managing changes in the weights beyond a simple gradient descent mechanism. PPO introduced a mechanism that restricted the flow of gradients [22], while TRPO used a natural gradient optimization algorithm to achieve monotonic improvement [23]. Further work has shown that in large and wide Neural Networks, even tiny changes in weights can have tremendous effects in the policy [95].

The third area of interest is related to distributional agents. Agents that estimate distributions, such as C51 [61, 63], Quantile Regression DQN [113], Implicit Quantile Regression (IQR) [114], Distributed Distributional DDPG (D4PG) [115] exhibit enormous improvements over their non distributional counterparts. Our hypothesis is that estimating distributions and in particular quantiles is significantly easier than estimating true values as true values are usually very large. This belief stems from the fact that normalized actions e.g. squashed in [-1,1], and normalized observations seem to greatly assist in learning [24, 47].

10.1.1 The future for the Obstacle Tower

The Obstacle Tower proved to be a formidable opponent, one that has not seen a conqueror yet, let alone one with a general solution that did not include domain knowledge. We believe there is potential in this domain, especially when we consider the overall poor sample efficiency of current methods.

First, we observe that the OT allows an agent to observe the world through different themes. We believe that this can be exploited by an agent through Contrastive Learning [103], or by forcing similar representations from a Vision module, in order to achieve better state representations. This can be easily achieved by running the environment multiple times with the same seed and just the different theme.

Our current implementation can be easily extended to include a form of Contrastive Learning as done in [103]. In particular, the usage of MMD-VAE was deliberate because it makes it fairly trivial to enforce a constraint on the representation. MMD-VAE works by enforcing a constraint on the latent variable. The constraint forces the latent variable to follow a Gaussian Distribution and is computed through Gaussian Kernels, the target

kernels. We can treat the representations created by the AutoEncoder with one theme as the target kernel, and the other as the traditional latent variable. This modification is similar to Siamese Networks [35].

Second, we believe that Imitation Learning is paramount to learning how to solve the OT. Imitation Learning is a class of algorithms that use experience derived from an expert in order to learn how to solve a particular problem [116, 117, 118]. Algorithms such as Generative Adversarial Imitation Learning (GAIL) [117] and Behaviour Cloning (BC) [116] are popular IL methods that are often used as a form of Pre-training for an agent [29]. In essence, they introduce some form of inductive bias into the model.

Last, but not least, believe that there is potential in constructing sophisticated meta-controllers to control an agent's curriculum such as those used in [11]. Because the OT exhibits natural difficulty progression scheme, we believe that it can be exploited to learn faster.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Computing Research Repository CoRR*, vol. abs/1207.4708, 2012. [Preprint] arxiv:1207.4708.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *Computing Research Repository CoRR*, vol. abs/gym, 2016. [Preprint] arxiv:gym.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017. [Preprint] arxiv:1712.01815.

[6] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, "Starcraft ii: A new challenge for reinforcement learning," 2017. [Preprint] arxiv:1708.04782.

[7] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, pp. 350–354, Nov. 2019.

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

[9] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, "Go-explore: a new approach for hard-exploration problems," *Computing Research Repository, CoRR*, 2019. [Preprint] arxiv:1901.10995.

[10] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, "First return then explore," 2020. [Preprint] arxiv:2004.12919.

[11] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, D. Guo, and C. Blundell, "Agent57: Outperforming the atari human benchmark," 2020. [Preprint] arxiv:2003.13350.

[12] S. Kapturowski, G. Ostrovski, W. Dabney, J. Quan, and R. Munos, "Recurrent experience replay in distributed reinforcement learning," in *International Conference on Learning Representations*, 2019.

[13] T. L. Paine, Ç. Gülçehre, B. Shahriari, M. Denil, M. D. Hoffman, H. Soyer, R. Tanburn, S. Kapturowski, N. C. Rabinowitz, D. Williams, G. Barth-Maron, Z. Wang, N. de Freitas, and W. Team, "Making efficient use of demonstrations to solve hard exploration problems," *Computing Research Repository, CoRR*, 2019. [Preprint] arxiv:1909.01387.

[14] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, "Mastering atari, go, chess and shogi by planning with a learned model," 2019. [Preprint] arxiv:1911.08265.

[15] A. P. Badia, P. Sprechmann, A. Vitvitskyi, D. Guo, B. Piot, S. Kapturowski, O. Tieleman, M. Arjovsky, A. Pritzel, A. Bolt, and C. Blundell, "Never give up:

Learning directed exploration strategies," 2020. [Preprint] arxiv:2020.06038.

[16] R. Raileanu and T. Rocktäschel, "Ride: Rewarding impact-driven exploration for procedurally-generated environments," in *International Conference on Learning Representations*, 2020.

[17] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. London, England: The MIT Press, Cambridge Massachusetts, second ed., 2018.

[18] D. Sproatt and A. Navab, "Operant Conditioning," in *Encyclopedia of Autism Spectrum Disorders* (F. R. Volkmar, ed.), pp. 2087–2088, New York, NY: Springer New York, 2013.

[19] I. Osband, Y. Doron, M. Hessel, J. Aslanides, E. Sezener, A. Saraiva, K. McKinney, T. Lattimore, C. Szepesvari, S. Singh, B. V. Roy, R. Sutton, D. Silver, and H. V. Hasselt, "Behaviour suite for reinforcement learning," *Computing Research Repository, CoRR*, 2019. [Preprint] arxiv:1908.03568.

[20] I. Osband, C. Blundell, A. Pritzel, and B. V. Roy, "Deep exploration via bootstrapped dqn," 2016. [Preprint] arxiv:1602.04621.

[21] A. Juliani, A. Khalifa, V.-P. Berges, J. Harper, E. Teng, H. Henry, A. Crespi, J. Togelius, and D. Lange, "Obstacle tower: A generalization challenge in vision, control, and planning," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI-19*, pp. 2684–2691, International Joint Conferences on Artificial Intelligence Organization, 2019.

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *Computing Research Repository, CoRR*, vol. abs/1707.06347, 2017. [Preprint] arxiv:1707.06347.

[23] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015* (F. R. Bach and D. M. Blei, eds.), pp. 1889–1897, PMLR, 2015.

[24] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "Implementation matters in deep rl: A case study on ppo and trpo," in *International Conference on Learning Representations*, 2020.

[25] R. S. Sutton, D. Precup, and S. P. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artif. Intell.*, vol. 112, no. 1-2, pp. 181–211, 1999.

[26] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," 2016. [Preprint] arxiv:1609.05140.

[27] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," in *Advances in Neural Information Processing Systems 29* (D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, eds.), pp. 3675–3683, Curran Associates, Inc., 2016.

[28] O. Nachum, S. Gu, H. Lee, and S. Levine, "Data-efficient hierarchical reinforcement learning," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018* (S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), pp. 3307–3317, Curran Associates, Inc., 2018.

[29] A. Gupta, V. Kumar, C. Lynch, S. Levine, and K. Hausman, "Relay policy learning: Solving long-horizon tasks via imitation and reinforcement learning," *Computing Research Repository, CoRR*, 2019. [Preprint] arxiv:1910.11956.

[30] A. Levy, R. P. Jr., and K. Saenko, "Learning multi-level hierarchies with hindsight," *Computing Research Repository, CoRR*, 2017. [Preprint] arxiv:1712.00948.

[31] N. Bostrom, "Ethical issues in advanced artificial intelligence," in *Science fiction and philosophy: from time travel to superintelligence* (S. Schneider, ed.), pp. 277–284, Whiley-Blackwell, 2009.

[32] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu, "Feudal networks for hierarchical reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 3540–3549, PMLR, 2017.

[33] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[34] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for

statistical machine translation," *Computing Research Repository CoRR*, 2014. [Preprint] arxiv:1406.1078.

[35] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. London, England: The MIT Press, Cambridge Massachusetts, 2016. `http://www.deeplearningbook.org`.

[36] M. J. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," in *2015 AAAI Fall Symposia, Arlington, Virginia, USA, November 12-14, 2015*, pp. 29–37, AAAI Press, 2015.

[37] L. P. Kaelbling, "Learning to achieve goals," in *Proceedings of IJCAI 1993*, pp. 1094–1098, Morgan Kaufmann, 1993.

[38] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," *Computing Research Repository, CoRR*, vol. abs/1707.01495, 2017. [Preprint] arxiv:1707.01495.

[39] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal value function approximators," in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 1312–1320, PMLR, 07–09 Jul 2015.

[40] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2009.

[41] J. Z. Leibo, J. Pérolat, E. Hughes, S. Wheelwright, A. H. Marblestone, E. A. Duéñez-Guzmán, P. Sunehag, I. Dunning, and T. Graepel, "Malthusian reinforcement learning," *Computing Research Repository CoRR*, vol. abs/1812.07019, 2018. [Preprint]

arxiv:1812.07019.

[42] P. Chrabaszcz, I. Loshchilov, and F. Hutter, "Back to basics: Benchmarking canonical evolution strategies for playing atari," 2018. [Preprint] arxiv:1802.08842.

[43] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman, "Gotta learn fast: A new benchmark for generalization in rl," 2018. [Preprint] arxiv:1804.03720.

[44] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," vol. 24 of *Psychology of Learning and Motivation*, pp. 109 – 165, Academic Press, 1989.

[45] K. Lee, K. Lee, J. Shin, and H. Lee, "Network randomization: A simple technique for generalization in deep reinforcement learning," in *International Conference on Learning Representations*, 2020.

[46] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying generalization in reinforcement learning," *Computing Research Repository, CoRR*, vol. abs/1812.02341, 2018. [Preprint] arxiv:1812.02341.

[47] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018* (J. G. Dy and A. Krause, eds.), pp. 1856–1865, PMLR, 2018.

[48] Y. Burda, H. Edwards, A. J. Storkey, and O. Klimov, "Exploration by random network distillation," *Computing Research Repository, CoRR*, 2018. [Preprint] arxiv:1810.12894.

[49] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, "Noisy networks for exploration," *Computing Research Repository, CoRR*, vol. abs/1706.10295, 2019. [Preprint] arxiv:1706.10295.

[50] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *Computing Research Repository, CoRR*, vol. abs/1802.09477, 2018. [Preprint] arxiv:1802.09477.

[51] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *Computing Research Repository, CoRR*, 2015. [Preprint] arxiv:1509.02971.

[52] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *Computing Research Repository, CoRR*, vol. abs/1412.6980, 2014. [Preprint] arxiv:1412.6980.

[53] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *Computing Research Repository, CoRR*, 2014. [Preprint] arxiv:1312.6114.

[54] A. A. Alemi, B. Poole, I. Fischer, J. V. Dillon, R. A. Saurous, and K. Murphy, "Fixing a broken elbo," 2017. [Preprint] arxiv:1711.00464.

[55] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio, "Generating sentences from a continuous space," 2015. [Preprint] arxiv:1511.06349.

[56] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner, "beta-vae: Learning basic visual concepts with a constrained variational framework," in *International Conference on Learning Representations*, 2017.

[57] A. Gretton, K. Borgwardt, M. Rasch, B. Schölkopf, and A. J. Smola, "A kernel method for the two-sample-problem," in *Advances in Neural Information Processing Systems 19* (B. Schölkopf, J. C. Platt, and T. Hoffman, eds.), pp. 513–520, MIT Press, 2007.

[58] S. Zhao, J. Song, and S. Ermon, "Infovae: Information maximizing variational autoencoders," *Computing Research Repository, CoRR*, vol. abs/1706.02262, 2017. [Preprint] arxiv:1706.02262.

[59] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *Computing Research Repository CoRR*, 2015. [Preprint] arxiv:1509.06461.

[60] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," *Computing Research Repository CoRR*, 2015. [Preprint] arxiv:1511.06581.

[61] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," 2017. [Preprint] arxiv:1707.06887.

[62] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver, "Distributed prioritized experience replay," *Computing Research Repository, CoRR*, 2018. [Preprint] arxiv:1803.00933.

[63] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI-18)* (S. A. McIlraith and K. Q. Weinberger, eds.), pp. 3215–3222, AAAI Press, 2018.

[64] R. Munos, T. Stepleton, A. Harutyunyan, and M. G. Bellemare, "Safe and efficient off-policy reinforcement learning," 2016. [Preprint] arxiv:1606.02647.

[65] A. Harutyunyan, M. G. Bellemare, T. Stepleton, and R. Munos, "Q(λ) with off-policy corrections," 2016. [Preprint] arxiv:1602.04951.

[66] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2015. [Preprint] arxiv:1602.04951.

[67] D. Precup, R. S. Sutton, and S. P. Singh, "Eligibility traces for off-policy policy evaluation," in *ICML*, 2000.

[68] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," 2015. [Preprint] arxiv:1502.05477.

[69] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, May 1992.

[70] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2015. [Preprint] arxiv:1506.02438.

[71] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, p. I–387–I–395, JMLR.org, 2014.

[72] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," 2016. [Preprint] arxiv:1606.01868.

[73] G. Ostrovski, M. G. Bellemare, A. van den Oord, and R. Munos, "Count-based exploration with neural density models," 2017. [Preprint] arxiv:1703.01310.

[74] A. van den Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel recurrent neural networks," *Computing Research Repository CoRR*, vol. abs/1601.06759, 2016. [Preprint] arxiv:1601.06759.

[75] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. D. Turck, and P. Abbeel, "Vime: Variational information maximizing exploration," 2016. [Preprint] arxiv:1605.09674.

[76] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," 2017. [Preprint] arxiv:1705.05363.

[77] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu, "Feudal networks for hierarchical reinforcement learning," 2017. [Preprint] arxiv:1703.01161.

[78] P. Dayan and G. E. Hinton, "Feudal reinforcement learning," in *Advances in Neural Information Processing Systems 5* (S. J. Hanson, J. D. Cowan, and C. L. Giles, eds.), pp. 271–278, Morgan-Kaufmann, 1993.

[79] J. Koutník, K. Greff, F. Gomez, and J. Schmidhuber, "A clockwork rnn," 2014. [Preprint] arxiv:1402.3511.

[80] A. Goyal, S. Sodhani, J. Binas, X. B. Peng, S. Levine, and Y. Bengio, "Reinforcement learning with competitive ensembles of information-constrained primitives," in *International Conference on Learning Representations*, 2020.

[81] A. A. Alemi, I. Fischer, J. V. Dillon, and K. Murphy, "Deep variational information bottleneck," 2016. [Preprint] arxiv:1612.00410.

[82] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016* (M. Balcan and K. Q. Weinberger, eds.), vol. 48 of *JMLR Workshop and Conference Proceedings*, pp. 1928–1937, PMLR, 2016.

[83] P. Christodoulou, "Soft actor-critic for discrete action settings," *Computing Research Repository CoRR*, 2019. [Preprint] arxiv:1910.07207.

[84] A. A. Taiga, W. Fedus, M. C. Machado, A. Courville, and M. G. Bellemare, "On bonus based exploration methods in the arcade learning environment," in *International Conference on Learning Representations*, Paper 1510, 2020.

[85] A. W. Moore, "Efficient memory-based learning for robot control," tech. rep., 209 (UCAM-CL-TR-209 ISSN 1476-2986), Computer Laboratory, Univ of Cambridge, 1990.1990.

[86] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Phys. Rev.*, vol. 36, pp. 823–841, Sep 1930.

[87] T. G. Dietterich, "Hierarchical reinforcement learning with the MAXQ value function decomposition," *J. Artif. Intell. Res.*, vol. 13, pp. 227–303, 2000.

[88] R. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, pp. 181–211, 1999.

[89] M.-P. B. Schrader, "gym-sokoban." `https://github.com/mpSchrader/gym-sokoban`, 2018.

[90] A. Guez, M. Mirza, K. Gregor, R. Kabra, S. Racaniere, T. Weber, A. S. David Raposo, L. Orseau, T. Eccles, G. Wayne, D. Silver, T. Lillicrap, and V. Valdes, "An investigation of model-free planning: boxoban levels." `https://github.com/deepmind/boxoban-levels/`, 2018.

[91] A. Guez, M. Mirza, K. Gregor, R. Kabra, S. Racanière, T. Weber, D. Raposo, A. Santoro, L. Orseau, T. Eccles, G. Wayne, D. Silver, and T. Lillicrap, "An investigation of model-free planning," 2019. [Preprint] arxiv:1901.03559.

[92] L. Orseau, L. H. S. Lelis, T. Lattimore, and T. Weber, "Single-agent policy tree search with guarantees," 2018. [Preprint] arxiv:1811.10928.

[93] S. Racanière, T. Weber, D. Reichert, L. Buesing, A. Guez, D. Jimenez Rezende, A. Puigdomènech Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Hassabis, D. Silver, and D. Wierstra, "Imagination-augmented agents for deep reinforcement learning," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 5690–5701, Curran Associates, Inc., 2017.

[94] S. I. Pspace-complete, J. C. Culberson, and J. C. Culberson, "Sokoban is pspace-complete," 1997.

[95] A. Ilyas, L. Engstrom, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "Are deep policy gradient algorithms truly policy gradient algorithms?," *Computing Research Repository CoRR*, 2018. [Preprint] arxiv:1811.02553.

[96] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Y. W. Teh and M. Titterington, eds.), vol. 9 of *Proceedings of Machine Learning Research*, (Chia Laguna Resort, Sardinia, Italy), pp. 249–256, PMLR, 13–15 May 2010.

[97] D. Yarats, A. Zhang, I. Kostrikov, B. Amos, J. Pineau, and R. Fergus, "Improving sample efficiency in model-free reinforcement learning from images," 2019. [Preprint] arxiv:1910.01741.

[98] S. Zhao, J. Song, and S. Ermon, "Infovae: Information maximizing variational autoencoders," *Computing Research Repository, CoRR*, vol. abs/1706.02262, 2017. [Preprint] arxiv:1706.02262.

[99] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu, "Reinforcement learning with unsupervised auxiliary tasks," *Computing Research Repository CoRR*, 2016. [Preprint] arxiv:1611.05397.

[100] A. Raffin, A. Hill, K. R. Traoré, T. Lesort, N. D. Rodríguez, and D. Filliat, "Decoupling feature extraction from policy learning: assessing benefits of state representation learning in goal based robotics," *Computing Research Repository, CoRR*, 2019. [Preprint] arxiv:1901.08651.

[101] E. Shelhamer, P. Mahmoudieh, M. Argus, and T. Darrell, "Loss is its own reward: Self-supervision for reinforcement learning," 2016. [Preprint] arxiv:1612.07307.

[102] I. Higgins, A. Pal, A. A. Rusu, L. Matthey, C. P. Burgess, A. Pritzel, M. Botvinick, C. Blundell, and A. Lerchner, "Darla: Improving zero-shot transfer in reinforcement

learning," 2017. [Preprint] arxiv:1707.08475.

[103] A. Srinivas, M. Laskin, and P. Abbeel, "Curl: Contrastive unsupervised representations for reinforcement learning," 2020. [Preprint] arxiv:2004.04136.

[104] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2013. [Preprint] arxiv:1312.6114.

[105] P. Ghosh, M. S. M. Sajjadi, A. Vergari, M. Black, and B. Schölkopf, "From variational to deterministic autoencoders," 2019. [Preprint] arxiv:1903.12436.

[106] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling, "Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents," 2017. [Preprint] arxiv:1709.06009.

[107] A. Kanervisto, C. Scheller, and V. Hautamäki, "Action space shaping in deep reinforcement learning," 2020. [Preprint] arxiv:2004.00980.

[108] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines." `https://github.com/hill-a/stable-baselines`, 2018.

[109] A. Raffin and F. Stulp, "Generalized state-dependent exploration for deep reinforcement learning in robotics," 2020. [Preprint] arxiv:2005.05719.

[110] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, vol. 113, pp. 54 – 71, 2019.

[111] V. Ramanujan, M. Wortsman, A. Kembhavi, A. Farhadi, and M. Rastegari, "What's hidden in a randomly weighted neural network?," 2019. [Preprint] arxiv:1911.13299.

[112] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," *Computing Research Repository CoRR*, 2018. [Preprint] arxiv:1803.03635.

[113] W. Dabney, M. Rowland, M. G. Bellemare, and R. Munos, "Distributional reinforcement learning with quantile regression," 2017. [Preprint] arxiv:1710.10044.

[114] W. Dabney, G. Ostrovski, D. Silver, and R. Munos, "Implicit quantile networks for distributional reinforcement learning," 2018. [Preprint] arxiv:1806.06923.

[115] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. Lillicrap, "Distributional policy gradients," in *International Conference on Learning Representations*, 2018.

[116] D. A. Pomerleau, "Efficient training of artificial neural networks for autonomous navigation," *Neural Comput.*, vol. 3, p. 88–97, Mar. 1991.

[117] J. Ho and S. Ermon, "Generative adversarial imitation learning," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016* (D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, eds.), pp. 4565–4573, Curran Associates, Inc., 2016.

[118] T. Salimans and R. Chen, "Learning montezuma's revenge from a single demonstration," 2018. [Preprint] arxiv:1812.03381.

Appendix A

DQN and Friends

A.1 Dueling Architectures

The common approach to thinking about Q-functions is as estimators of state-action pairs. An important alternative view is that the Q-function $Q(s, a)$ learns the value function $V^\pi(s)$ for state s, and the advantage function $A^\pi(s, a)$. This simple decomposition allows us to create a neural network that learns both $A^\pi(s, a)$ and $V^\pi(s)$ by providing two output streams or 'dueling networks'[60]. From there, we can retrieve $Q^\pi(s, a)$ using:

$$Q^\pi(s, a) = A^\pi(s, a) + V^\pi(s) \tag{97}$$

This change, in theory, allows the network to generalize the state value across all actions, thus, it does not need to be learnt for every action. However, the semantics are not preserved, in-fact, this change prevents us from retrieving either $A^\pi(s, a)$ or $V^\pi(s)$:

$$\hat{A}^\pi(s, a) = A^\pi(s, a) + Z$$

$$\hat{V}^\pi(s) = V^\pi(s) - Z$$

$$\implies Q^\pi(s, a) = A^\pi(s, a) + V^\pi(s) = \hat{A}^\pi(s, a) + \hat{V}^\pi(s)$$

Thus, the network can not estimate $V^{\pi}(s)$ and $A^{\pi}(s,a)$ directly.

Let us recall that $V^{*}(s) = \max_a Q^{*}(s,a)$, thus, $\max_a A^{*}(s,a) = 0$. For a greedy policy, $V^{\pi}(s) = \max_a Q^{\pi}(s,a)$, and $\max_a A^{\pi}(s,a) = 0$ thus, we can estimate both $V^{\pi}(s)$ and $A^{\pi}(s,a)$ by learning the mapping:

$$Q^{\pi}(s,a) = V^{\pi}(s) + (A^{\pi}(s,a) - \max_{a'} A^{\pi}(s,a')) \tag{98}$$

Another approach that, while does not preserve the original semantics of $V^{\pi}(s)$ and $A^{\pi}(s,a)$, is more stable[60] and learns the mapping:

$$Q^{\pi}(s,a) = V^{\pi}(s) + (A(s,a) - |\mathcal{A}|^{-1} \sum_{a'} A^{\pi}(s,a')) \tag{99}$$

A.2 C51

Let us consider the Q-function:

$$Q_{\pi}(s,a) = \mathbb{E}_{\pi}\left[R|S = s, A = a\right] \tag{100}$$

The Q function learns an expectation over the immediate reward, plus the discounted returns after following policy π. Since we are learning an expectation, instead of learning the mean, we can learn a distribution and retrieve back the mean. Indeed this is the idea behind C51 [61].

Modelling distributions has the additional benefit of allowing us to estimate the variance between the estimated rewards of the actions. Thus, given two actions that have the same mean, a risk averse agent would prefer the one with the lowest variance. In addition, variance *could* be a proxy for the amount of exploration performed with particular state-action pairs, this however is not a guaranteed. Further on, estimating distributions makes the process of optimization easier since the targets are bound within some range, and thus, do not need to estimate values of different, or very large magnitudes [61].

Learning to learn a Q-function, we recursively apply it to gain the estimate for the next step. Estimating distributions over scalar values raises the following issues:

- how do we represent the estimations?
- how do we adjust the estimations between two states?

C51 solves the first issue by estimating a Categorical distribution of 51 discrete values, and solves the second issue by minimizing the cross entropy loss between the current estimate and the future estimate [61].

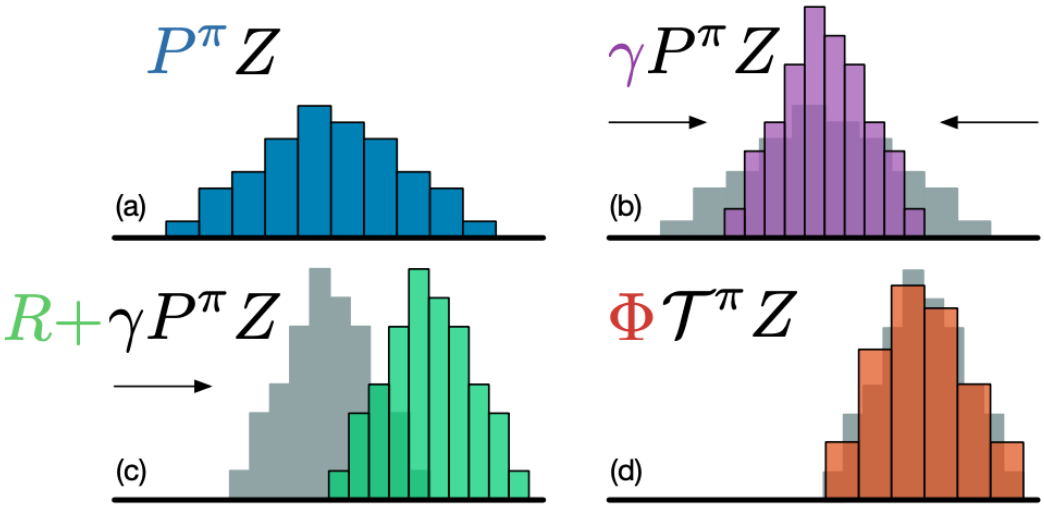

Figure 18: a) The target distribution as estimated by the Q-Network with emerging policy π. b) applying the discount factor results in shrinking the distribution towards 0. c) Adding the the reward shifts the distribution. d) The distribution is projected onto the target distribution. The target distribution is readjusted to fill in the correct number of atoms [61]

To perform control, the Q-Network in C51 outputs $|\mathcal{A}|$ vectors of 51 values. We perform Softmax onto each vector to compute the support p_i the atom i and then estimate the mean. In addition, each bin is given a value z_i annealed from V_{min} up to V_{max}, where V_{min}, V_{max} are the bounds of the distribution. In C51 $V_{max} = -V_{min} = 10$.

$$\Delta z = (V_{max} - V_{min})/N \tag{101}$$

$$z_i = V_{min} + i\Delta z \tag{102}$$

$$p_i(s,a) = \frac{e^{\theta_i(s,a)}}{\sum_j^N e^{\theta_j(s,a))}} \tag{103}$$

Where N is the number of atoms, and $\theta_i(s,a)$ is the output of the Q-Network for state and action s, a and atom i. Therefore, to select the best action, we use p and z to find the mean of the distribution as in the Q-value Estimation algorithm.

Algorithm 16 Q-value Estimation

Require: $s \in \mathcal{S}$, $a \in \mathcal{A}$
  $q = \sum_i^N z_i p_i(s,a)$
  return q

Recall that to train a Q-Network with DQN and friends, we treat it as a regression problem. In C51, we treat it as a classification problem, where the result of the network for state-action distribution can be thought of as the labels and the target labels are derived from the target network.

In C51, we estimate the distribution of the next state, and discount the values by γ and add the reward r. Then we readjust the target distribution in 51 bins. The step of readjusting is necessary because we shift and squeeze the distribution, thus it is possible that it ends up out of bounds. The readjustment process ensures that estimations over the bounds are put into the final or first bin if the estimation is over or under the bin. The error is reduced through CrossEntropyLoss [61].

Algorithm 17 C51 Optimization Step

Require: $s \in \mathcal{S}, a \in \mathcal{A}, \gamma, s' \in \mathcal{S}$

$a' \leftarrow \arg\max_{a'}$ Q-Value-Estimation(s', a') ▷ Compute the best action

$b_i = (\gamma z_i) p_i(s', a') + r$ ▷ Create Projection

$m \leftarrow adjust\ bins(b, 51)$

return $-\sum_i^N m_i \log p_i(s, a)$

Appendix B

Policy Gradient methods

B.1 Variance of Advantage function

Consider the generalized policy gradient gradient estimate:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{T} \in \mathcal{D}} \sum_{t=0}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t|S_t) \Psi_t \tag{104}$$

Ψ_t can be any of

- $G_{t:T}$
- $\sum_{t'=t}^{T} \gamma^{t'-t} R_{t'} - b(s_{t'})$[1]
- $V^{\pi}(s)$
- $Q^{\pi}(s, a)$
- $A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s')$
- $r_t + \gamma V^{\pi}(s') - V(s)$

[1] Any substitute of $b(s_t)$ is a valid baseline function

However, the common approach is to use the advantage function because it has the lowest variance. This is very simple to evaluate:

$$Q^{\pi}(s,a) = r(s,a) + \mathbb{E}_{s'}\left[\gamma V^{\pi}(s') \middle| S = s, A = a\right]$$

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}\left[r(s,a) + \mathbb{E}_{s'}\left[\gamma V^{\pi}(s') \middle| S = s, A = a\right]\right]$$

$$A^{\pi}(s,a) = r(s,a) - \mathbb{E}_{a\sim\pi}\left[r(s,a)\right] = r(s,a) - \sum_{a'\in\mathcal{A}} r(s,a')\pi(a'|s) \tag{105}$$

$$\hat{A}^{\pi}(s,a) = -V(s) + r_t + \gamma\mathbb{E}_{s'}\left[V(s')\right] \tag{106}$$

Whereas the rest of the functions are either the returns, or estimates of the returns, thus, the Advantage function and its estimate provide the lowest variance.

Appendix C

Source

C.1 Environment

C.1.1 ALE Emulator

```
from pathlib import Path
import contextlib
import gym
import cv2
import numpy as np
from mlagents.envs.environment import UnityEnvironment
from stable_baselines3.common.type_aliases import (
    GymStepReturn, GymObs
)
from typing import Dict


class AleEmulator:
    """
    Implements the necessary functions for the OT
    to be used with gym.wrappers.atari_preprocessing
    """
```

```
def __init__(self, worker_id, path,
             realtime, timeout_wait):
    self.realtime = realtime
    self.worker_id = worker_id - 1
    while getattr(self, "unity_env", None) is None:
        self.worker_id += 1
        with contextlib.suppress(Exception):
            self.unity_env = UnityEnvironment(
                path,
                self.worker_id,
                docker_training=False,
                timeout_wait=timeout_wait,
            )
    self.brain_name = self.unity_env.external_brain_names[0]
    self.brain = self.unity_env.external_brains[
        self.brain_name
    ]
    self.info = self.unity_env.reset(
        train_mode=self.training
    )[self.brain_name]

def lives(self):
    return 1

def getScreenGrayscale(self, obs: np.ndarray):
    obs[:] = cv2.cvtColor(
        self.visual_obs, cv2.COLOR_BGR2GRAY
    )

def getScreenRGB2(self, obs: np.ndarray):
    obs[:] = self.visual_obs

def act(self, action):
    self.info = self.unity_env.step(
        action)[self.brain_name]
```

```
def reset(self, params):
    self.info = self.unity_env.reset(
        config=params, train_mode=self.training
    )[self.brain_name]

@property
def realtime(self) -> bool:
    return self._realtime

@realtime.setter
def realtime(self, rt) -> None:
    self._realtime = bool(rt)

@property
def training(self) -> bool:
    return not self.realtime

@training.setter
def training(self, tr) -> None:
    self.realtime = not tr

@property
def visual_obs(self) -> GymObs:
    return (
        self.info.visual_observations[0][0][:,
                                            :, :] * 255.0
    ).astype(np.uint8)

@property
def vector_obs(self) -> np.ndarray:
    return self.info.vector_observations[0]

@property
def reward(self) -> float:
    return self.info.rewards[0]
```

```
    @property
    def done(self) -> bool:
        return self.info.local_done[0]

    @property
    def reset_parameters(self) -> Dict[str, float]:
        return self.unity_env.reset_parameters

    def close(self):
        self.unity_env.close()
```

C.1.2 Communication

```
from pathlib import Path
import contextlib
import gym
import cv2
import numpy as np
from mlagents.envs.environment import UnityEnvironment
from stable_baselines3.common.type_aliases import (
    GymStepReturn, GymObs
)
from typing import Dict


class ObstacleTowerEnv(gym.Env):
    ALLOWED_VERSIONS = ["3.1"]

    def __init__(
        self, env_path=None,
        worker_id=0, timeout_wait=30,
        realtime=False, config=None
    ):
        """
        Arguments:
```

```
        env_path: The file path to the Unity executable.
        Does not require the extension.
        docker_training: Whether this is running within
        a docker environment and should use a virtual
          frame buffer (xvfb).
        worker_id: The index of the worker in the case where
          multiple environments are running.  Each
          environment reserves port (5005 + worker_id) for
          communication with the Unity executable.
        realtime_mode: Whether to render the
          environment window image and run environment
          in realtime.
      """
      env_path = env_path or str(
          Path(
              "~/.obstacle/obstacletower"
          ).expanduser().resolve()
      )
      self.ale = AleEmulator(
          worker_id, env_path, realtime, timeout_wait)
      self.action_space = gym.spaces.MultiDiscrete(
          [3, 3, 2, 3])
      self.observation_space = gym.spaces.Box(
          0, 255, self.ale.visual_obs.shape, dtype=np.uint8
      )

      # Environment reset parameters
      self._seed = None
      self._floor = None
      self.current_floor = self.ale.vector_obs[7]

      self.config = config or {}
      self.action_meanings = {0: "NOOP"}

  def reset(self, config=None):
      """
```

```
        Resets the state of the environment and returns an
        initial observation.
        In the case of multi-agent environments, this is a list.
        Returns: observation (object/list): the initial
            observation of the space.
        """
        reset_params = {}
        reset_params.update(getattr(self, "config", {}))
        reset_params.update(config or {})

        if self.floor is not None:
            reset_params["starting-floor"] = self.floor
        if self.seed is not None:
            reset_params["tower-seed"] = self.seed

        self.reset_params = self._env.reset_parameters
        self.ale.reset(reset_params)

        return self.ale.visual_obs

    def step(self, action) -> GymStepReturn:
        """
        Run one timestep of the environment's dynamics.
        When end of episode is reached, you are responsible
        for calling `reset()` to reset this environment's state.
        Accepts an action and returns a tuple
        (observation, reward, done, info). In the case of multi-agent
        environments, these are lists.
        Args:
            action (object/list): an action provided by the environment
        Returns:
            observation (object/list): agent's observation of the current
                environment
            reward (float/list) : amount of reward returned after previous action
            done (boolean/list): whether the episode has ended.
            info (dict): contains auxiliary diagnostic information,
```

```
        """
        action = list(action)
        self.ale.act(action)
        obs = self.ale.visual_obs
        reward = self.ale.reward
        done = self.ale.done
        key = np.argmax(self.ale.vector_obs[0:6], axis=0)
        time = self.ale.vector_obs[6]
        floor = int(self.ale.vector_obs[7])
        info = dict(time=time, floor=floor, key=key)
        done = done or self.current_floor < floor
        self.current_floor = floor
        return self.ale.visual_obs, self.ale.reward, self.ale.done, info

    def close(self):
        """Override _close in your subclass to perform any necessary cleanup.
        Environments will automatically close() themselves when
        garbage collected or when the program exits.
        """
        self.ale.close()

    def get_action_meanings(self):
        return self.action_meanings

    def seed(self, seed=None):
        """Sets a fixed seed for this env's random number generator(s).
        The valid range for seeds is [0, 99999). By default a random seed
        will be chosen.
        """
        assert 0 <= int(seed) < 99999
        self._seed = int(seed)

    @property
    def floor(self):
        return self._floor
```

```
    @floor.setter
    def floor(self, floor):
        assert 0 < floor < 100
        self._floor = floor
s
    @property
    def reward_range(self):
        return -float("inf"), float("inf")


gym.register(
    "OT-v0", entry_point="obstacle.tower.env:ObstacleTowerEnv")

gym.register(
    "OTRT-v0",
    entry_point="obstacle.tower.env:ObstacleTowerEnv",
    kwargs=dict(realtime=True),
)
```

C.1.3 Action Wrappers

```
import itertools
import functools
import operator
from typing import Union, Dict, Tuple, List, Iterable

import gym
from gym.spaces import Discrete, MultiDiscrete


class ActionMapping(gym.ActionWrapper):
    def __init__(self, env, action_space, mapping):
        super().__init__(env)
        self.__action_mapping = mapping
        self.action_space = action_space
        assert mapping is not None
```

```
def action(self, act):
    return self.__action_mapping[act]


@classmethod
def Flatten(cls, env: gym.Env) -> "ActionMapping":
    action_space: Union[Discrete,
                        MultiDiscrete] = env.action_space
    mapping: Dict[int, Union[int, Tuple[int, ...]]] = {}
    if isinstance(action_space, Discrete):
        mapping.update(
            {i: i for i in range(action_space.n)})
    else:
        action_ranges = itertools.product(
            *map(range, action_space.nvec))
        mapping.update(
            {i: tuple(action)
             for i, action in enumerate(action_ranges)}
        )
        action_space = Discrete(
            functools.reduce(
                operator.mul, action_space.nvec)
        )
    return cls(env, action_space, mapping)


@classmethod
def Combiner(
    cls, env: gym.Env, target_dim: int, other: int, *others: int
) -> "ActionMapping":
    dims = env.action_space.nvec
    others = (other, *others)
    assert target_dim not in others
    assert all(dims[target_dim] == dims[other]
               for other in others)
    mapping: Dict[Tuple[int, ...], Tuple[int, ...]] = {}
    action_ranges = itertools.product(
```

```
            *map(range, env.action_space.nvec))
        for action in action_ranges:
            newaction: List[int] = []
            for index, val in enumerate(action):
                if index in others:
                    newaction.append(action[target_dim])
                else:
                    newaction.append(val)
            action = tuple(
                [action[i]
                    for i in range(len(action)) if i not in others]
            )
            mapping[action] = tuple(newaction)
        action_space = gym.spaces.MultiDiscrete(
            [dim for i, dim in enumerate(
                dims) if i not in others]
        )
        return cls(env, action_space, mapping)

    @classmethod
    def Remover(cls, env: gym.Env, index: int,
                value: int) -> "ActionMapping":
        mapping: Dict[Tuple[int, ...], Tuple[int, ...]] = {}
        action_space = env.action_space
        assert action_space.nvec[index] >= 2
        assert action_space.nvec[index] > value

        action_ranges: Iterable[Tuple[int, ...]] = itertools.product(
            *map(range, action_space.nvec)
        )
        els = range(action_space.nvec[index])
        actions: List[int] = [
            i for i in range(action_space.nvec[index]) if i != value
        ]
        action_mapping = {
            oldvalue: newvalue for newvalue, oldvalue in enumerate(actions)
```

```
        }

        for action in action_ranges:
            if action[index] == value:
                continue
            newaction = (
                *action[:index],
                action_mapping[action[index]],
                *action[index + 1:],
            )
            mapping[newaction] = action

        nvec = list(action_space.nvec)
        nvec[index] -= 1
        action_space = MultiDiscrete(nvec)
        return cls(env, action_space, mapping)

    @classmethod
    def Compose(cls, env, *wrappers):
        mapping: Dict[Union[Tuple[int, ...], int],
                      Tuple[int, ...]] = {}
        action_space = wrappers[0].action_space

        def go_through(action):
            for f in wrappers:
                action = f.action(action)
            return action

        actions: Iterable[Union[Tuple[int, ...], int]]
        if isinstance(action_space, Discrete):
            actions = range(action_space.n)
        elif isinstance(action_space, MultiDiscrete):
            action_ranges: Iterable[Tuple[int, ...]] = itertools.product(
                *map(range, action_space.nvec)
            )
        else:
```

```
            raise TypeError(action_space)
        for action in actions:
            mapping[action] = go_through(action)

        return cls(env, action_space, mapping)
```

C.2 PPO

C.2.1 Policies

```
from typing import Optional, List, Tuple, Callable, Union, Dict, Type, Any
from functools import partial

import gym
import torch as th
import torch.nn as nn
import numpy as np

from hierarchical_methods.modules import NatureAE, NatureMMD
from stable_baselines3.common.policies import (
    BasePolicy,
    register_policy,
    MlpExtractor,
    create_sde_features_extractor,
    NatureCNN,
    BaseFeaturesExtractor,
    FlattenExtractor,
    create_mlp,
)
from stable_baselines3.common.distributions import (
    make_proba_distribution,
    Distribution,
    DiagGaussianDistribution,
    CategoricalDistribution,
    MultiCategoricalDistribution,
    BernoulliDistribution,
```

```
    StateDependentNoiseDistribution,
)
from stable_baselines3.common.preprocessing import preprocess_obs
from stable_baselines3.ppo.policies import PPOPolicy


class RndPolicy(PPOPolicy):
    """
    RndPolicy class (with both actor and critic) for A2C and derivates (PPO).

    :param observation_space: (gym.spaces.Space) Observation space
    :param action_space: (gym.spaces.Space) Action space
    :param lr_schedule: (Callable) Learning rate schedule (could be constant)
    :param net_arch: ([int or dict]) The specification of the policy and value networks.
    :param device: (str or th.device) Device on which the code should run.
    :param activation_fn: (Type[nn.Module]) Activation function
    :param ortho_init: (bool) Whether to use or not orthogonal initialization
    :param use_sde: (bool) Whether to use State Dependent Exploration or not
    :param log_std_init: (float) Initial value for the log standard deviation
    :param full_std: (bool) Whether to use (n_features x n_actions) parameters
        for the std instead of only (n_features,) when using gSDE
    :param sde_net_arch: ([int]) Network architecture for extracting features
        when using gSDE. If None, the latent features from the policy will be used.
        Pass an empty list to use the states as features.
    :param use_expln: (bool) Use ``expln()`` function instead of ``exp()`` to ensure
        a positive standard deviation (cf paper). It allows to keep variance
        above zero and prevent it from growing too fast. In practice, ``exp()`` is usually enou
    :param squash_output: (bool) Whether to squash the output using a tanh function,
        this allows to ensure boundaries when using gSDE.
    :param features_extractor_class: (Type[BaseFeaturesExtractor]) Features extractor to use.
    :param features_extractor_kwargs: (Optional[Dict[str, Any]]) Keyword arguments
        to pass to the feature extractor.
    :param normalize_images: (bool) Whether to normalize images or not,
         dividing by 255.0 (True by default)
    :param optimizer_class: (Type[th.optim.Optimizer]) The optimizer to use,
        ``th.optim.Adam`` by default
```

```
    :param optimizer_kwargs: (Optional[Dict[str, Any]]) Additional keyword arguments,
        excluding the learning rate, to pass to the optimizer
    """

    def __init__(
        self,
        observation_space: gym.spaces.Space,
        action_space: gym.spaces.Space,
        lr_schedule: Callable,
        net_arch: Optional[List[Union[int, Dict[str, List[int]]]]] = None,
        device: Union[th.device, str] = "auto",
        activation_fn: Type[nn.Module] = nn.Tanh,
        ortho_init: bool = True,
        use_sde: bool = False,
        log_std_init: float = 0.0,
        full_std: bool = True,
        sde_net_arch: Optional[List[int]] = None,
        use_expln: bool = False,
        squash_output: bool = False,
        features_extractor_class: Type[BaseFeaturesExtractor] = FlattenExtractor,
        features_extractor_kwargs: Optional[Dict[str, Any]] = None,
        rnd_rand_arch: Optional[List[int]] = None,
        rnd_distill_arch: Optional[List[int]] = None,
        rnd_latent_dim: Optional[int] = None,
        normalize_images: bool = True,
        optimizer_class: Type[th.optim.Optimizer] = th.optim.Adam,
        optimizer_kwargs: Optional[Dict[str, Any]] = None,
    ):
        self.rnd_latent_dim = rnd_latent_dim or 64
        self.rnd_rand_arch = rnd_rand_arch or [64, 64]
        self.rnd_distill_arch = rnd_distill_arch or [64, 64]

        super(RndPolicy, self).__init__(
            observation_space=observation_space,
            action_space=action_space,
            lr_schedule=lr_schedule,
```

```
            net_arch=net_arch,
            device=device,
            activation_fn=activation_fn,
            ortho_init=ortho_init,
            use_sde=use_sde,
            log_std_init=log_std_init,
            full_std=full_std,
            sde_net_arch=sde_net_arch,
            use_expln=use_expln,
            squash_output=squash_output,
            features_extractor_class=features_extractor_class,
            features_extractor_kwargs=features_extractor_kwargs,
            normalize_images=normalize_images,
            optimizer_class=optimizer_class,
            optimizer_kwargs=optimizer_kwargs,
        )

    def _get_data(self) -> Dict[str, Any]:
        data = super()._get_data()

        data.update(
            dict(
                net_arch=self.net_arch,
                activation_fn=self.activation_fn,
                use_sde=self.use_sde,
                log_std_init=self.log_std_init,
                squash_output=(
                    self.dist_kwargs["squash_output"] if self.dist_kwargs else None
                ),
                full_std=(
                    self.dist_kwargs["full_std"] if self.dist_kwargs else None),
                sde_net_arch=(
                    self.dist_kwargs["sde_net_arch"] if self.dist_kwargs else None
                ),
                use_expln=self.dist_kwargs["use_expln"] if self.dist_kwargs else None,
                # dummy lr schedule, not needed for loading
```

```
                # policy alone
                lr_schedule=self._dummy_schedule,
                ortho_init=self.ortho_init,
                optimizer_class=self.optimizer_class,
                optimizer_kwargs=self.optimizer_kwargs,
                features_extractor_class=self.features_extractor_class,
                features_extractor_kwargs=self.features_extractor_kwargs,
                rnd_features_extractor_class=self.rnd_features_extractor_class,
                rnd_features_extractor_kwargs=self.rnd_features_extractor_kwargs,
            )
        )
        return data

    def _build(self, lr_schedule: Callable) -> None:
        """
        Create the networks and the optimizer.

        :param lr_schedule: (Callable) Learning rate schedule
            lr_schedule(1) is the initial learning rate
        """
        self.mlp_extractor = MlpExtractor(
            self.features_dim,
            net_arch=self.net_arch,
            activation_fn=self.activation_fn,
            device=self.device,
        )

        self.rnd_net = nn.Sequential(
            *create_mlp(
                self.features_dim,
                self.rnd_latent_dim,
                net_arch=self.rnd_rand_arch,
                activation_fn=self.activation_fn,
            )
        )
        self.rnd_net.requires_grad_(False)
```

```
self.rnd_distill_net = nn.Sequential(
    *create_mlp(
        self.features_dim,
        self.rnd_latent_dim,
        net_arch=self.rnd_distill_arch,
        activation_fn=self.activation_fn,
    )
)

latent_dim_pi = self.mlp_extractor.latent_dim_pi

# Separate feature extractor for gSDE
if self.sde_net_arch is not None:
    self.sde_features_extractor, latent_sde_dim = create_sde_features_extractor(
        self.features_dim, self.sde_net_arch, self.activation_fn
    )

if isinstance(self.action_dist,
              DiagGaussianDistribution):
    self.action_net, self.log_std = self.action_dist.proba_distribution_net(
        latent_dim=latent_dim_pi, log_std_init=self.log_std_init
    )
elif isinstance(self.action_dist, StateDependentNoiseDistribution):
    latent_sde_dim = (
        latent_dim_pi if self.sde_net_arch is None else latent_sde_dim
    )
    self.action_net, self.log_std = self.action_dist.proba_distribution_net(
        latent_dim=latent_dim_pi,
        latent_sde_dim=latent_sde_dim,
        log_std_init=self.log_std_init,
    )
elif isinstance(self.action_dist, CategoricalDistribution):
    self.action_net = self.action_dist.proba_distribution_net(
        latent_dim=latent_dim_pi
    )
```

```
        elif isinstance(self.action_dist, MultiCategoricalDistribution):
            self.action_net = self.action_dist.proba_distribution_net(
                latent_dim=latent_dim_pi
            )
        elif isinstance(self.action_dist, BernoulliDistribution):
            self.action_net = self.action_dist.proba_distribution_net(
                latent_dim=latent_dim_pi
            )

        self.value_net = nn.Linear(
            self.mlp_extractor.latent_dim_vf, 1)

        # Init weights: use orthogonal initialization
        # with small initial weight for the output
        if self.ortho_init:
            # TODO: check for features_extractor
            inits = {
                self.features_extractor: np.sqrt(2),
                self.mlp_extractor: np.sqrt(2),
                self.action_net: 1,
                self.value_net: 0.01,
                self.rnd_net: np.sqrt(2),
                self.rnd_distill_net: np.sqrt(2),
            }
            for module, gain in inits.items():
                module.apply(
                    lambda module: self.init_weights(module, gain))
        # Setup optimizer with initial learning rate
        self.optimizer = self.optimizer_class(
            self.parameters(), lr=lr_schedule(1), **self.optimizer_kwargs
        )

    def forward(
        self, obs: th.Tensor, deterministic: bool = False
    ) -> Tuple[th.Tensor, th.Tensor, th.Tensor, th.Tensor]:
        """
```

```
        Forward pass in all the networks (actor and critic)

        :param obs: (th.Tensor) Observation
        :param deterministic: (bool) Whether to sample or use deterministic actions
        :return: (Tuple[th.Tensor, th.Tensor, th.Tensor, th.Tensor])
            action, value, log probability of the action and distillation error
        """
        latent_pi, latent_vf, latent_sde, latent_rnd, latent_distill = self._get_latent(
            obs
        )
        # Evaluate the values for the given observations
        values = self.value_net(latent_vf)
        distribution = self._get_action_dist_from_latent(
            latent_pi, latent_sde=latent_sde
        )
        actions = distribution.get_actions(
            deterministic=deterministic)
        log_prob = distribution.log_prob(actions)
        rnd_diff = ((latent_rnd - latent_distill)
                    ** 2).mean(dim=1)
        return actions, values, log_prob, rnd_diff

    def _get_latent(
        self, obs: th.Tensor
    ) -> Tuple[th.Tensor, th.Tensor, th.Tensor, th.Tensor, th.Tensor]:
        """
        Get the latent code (i.e., activations of the last layer of each network)
        for the different networks.

        :param obs: (th.Tensor) Observation
        :return: (Tuple[th.Tensor, th.Tensor, th.Tensor]) Latent codes
            for the actor, the value function and for gSDE function
        """
        # Preprocess the observation if needed
        features = self.extract_features(obs)
        latent_pi, latent_vf = self.mlp_extractor(features)
```

```
        # Features for sde
        latent_sde = latent_pi
        if self.sde_features_extractor is not None:
            latent_sde = self.sde_features_extractor(
                features)

        latent_rnd = self.rnd_net.forward(features)
        latent_distill = self.rnd_distill_net(features)
        return latent_pi, latent_vf, latent_sde, latent_rnd, latent_distill

    def _predict(
        self, observation: th.Tensor, deterministic: bool = False
    ) -> th.Tensor:
        """
        Get the action according to the policy for a given observation.

        :param observation: (th.Tensor)
        :param deterministic: (bool) Whether to use stochastic or deterministic actions
        :return: (th.Tensor) Taken action according to the policy
        """
        latent_pi, _, latent_sde, * \
            _ = self._get_latent(observation)
        distribution = self._get_action_dist_from_latent(
            latent_pi, latent_sde)
        return distribution.get_actions(
            deterministic=deterministic)

    def evaluate_actions(
        self, obs: th.Tensor, actions: th.Tensor
    ) -> Tuple[th.Tensor, th.Tensor, th.Tensor, th.Tensor]:
        """
        Evaluate actions according to the current policy,
        given the observations.

        :param obs: (th.Tensor)
```

```
        :param actions: (th.Tensor)
        :return: (th.Tensor, th.Tensor, th.Tensor) estimated value, log likelihood of taking th
            entropy of the action distribution, and the rnd error
        """
        latent_pi, latent_vf, latent_sde, latent_rnd, latent_distill = self._get_latent(
            obs
        )
        distribution = self._get_action_dist_from_latent(
            latent_pi, latent_sde)
        log_prob = distribution.log_prob(actions)
        values = self.value_net(latent_vf)
        rnd_diff = (latent_rnd - latent_distill) ** 2
        return values, log_prob, distribution.entropy(), rnd_diff


class CnnRndPolicy(RndPolicy):
    """
    CnnPolicy class (with both actor and critic) for A2C and derivates (PPO).

    :param observation_space: (gym.spaces.Space) Observation space
    :param action_space: (gym.spaces.Space) Action space
    :param lr_schedule: (Callable) Learning rate schedule (could be constant)
    :param net_arch: ([int or dict]) The specification of the policy and value networks.
    :param device: (str or th.device) Device on which the code should run.
    :param activation_fn: (Type[nn.Module]) Activation function
    :param ortho_init: (bool) Whether to use or not orthogonal initialization
    :param use_sde: (bool) Whether to use State Dependent Exploration or not
    :param log_std_init: (float) Initial value for the log standard deviation
    :param full_std: (bool) Whether to use (n_features x n_actions) parameters
        for the std instead of only (n_features,) when using gSDE
    :param sde_net_arch: ([int]) Network architecture for extracting features
        when using gSDE. If None, the latent features from the policy will be used.
        Pass an empty list to use the states as features.
    :param use_expln: (bool) Use ``expln()`` function instead of ``exp()`` to ensure
        a positive standard deviation (cf paper). It allows to keep variance
        above zero and prevent it from growing too fast. In practice, ``exp()`` is usually enou
```

```
    :param squash_output: (bool) Whether to squash the output using a tanh function,
        this allows to ensure boundaries when using gSDE.
    :param features_extractor_class: (Type[BaseFeaturesExtractor]) Features extractor to use.
    :param features_extractor_kwargs: (Optional[Dict[str, Any]]) Keyword arguments
        to pass to the feature extractor.
    :param normalize_images: (bool) Whether to normalize images or not,
         dividing by 255.0 (True by default)
    :param optimizer_class: (Type[th.optim.Optimizer]) The optimizer to use,
        ``th.optim.Adam`` by default
    :param optimizer_kwargs: (Optional[Dict[str, Any]]) Additional keyword arguments,
        excluding the learning rate, to pass to the optimizer
    """

    def __init__(
        self,
        observation_space: gym.spaces.Space,
        action_space: gym.spaces.Space,
        lr_schedule: Callable,
        net_arch: Optional[List[Union[int, Dict[str, List[int]]]]] = None,
        device: Union[th.device, str] = "auto",
        activation_fn: Type[nn.Module] = nn.Tanh,
        ortho_init: bool = True,
        use_sde: bool = False,
        log_std_init: float = 0.0,
        full_std: bool = True,
        sde_net_arch: Optional[List[int]] = None,
        use_expln: bool = False,
        squash_output: bool = False,
        features_extractor_class: Type[BaseFeaturesExtractor] = NatureCNN,
        features_extractor_kwargs: Optional[Dict[str, Any]] = None,
        rnd_rand_arch: Optional[List[int]] = None,
        rnd_distill_arch: Optional[List[int]] = None,
        rnd_latent_dim: Optional[int] = None,
        normalize_images: bool = True,
        optimizer_class: Type[th.optim.Optimizer] = th.optim.Adam,
        optimizer_kwargs: Optional[Dict[str, Any]] = None,
```

```
    ):
        super(CnnRndPolicy, self).__init__(
            observation_space,
            action_space,
            lr_schedule,
            net_arch,
            device,
            activation_fn,
            ortho_init,
            use_sde,
            log_std_init,
            full_std,
            sde_net_arch,
            use_expln,
            squash_output,
            features_extractor_class,
            features_extractor_kwargs,
            rnd_rand_arch,
            rnd_distill_arch,
            rnd_latent_dim,
            normalize_images,
            optimizer_class,
            optimizer_kwargs,
        )


class AERndPolicy(CnnRndPolicy):
    """
        MMDVae Policy class (with both actor and critic) for A2C and derivates (PPO).

        :param observation_space: (gym.spaces.Space) Observation space
        :param action_space: (gym.spaces.Space) Action space
        :param lr_schedule: (Callable) Learning rate schedule (could be constant)
        :param net_arch: ([int or dict]) The specification of the policy and value networks.
        :param device: (str or th.device) Device on which the code should run.
        :param activation_fn: (Type[nn.Module]) Activation function
```

```
        :param ortho_init: (bool) Whether to use or not orthogonal initialization
        :param use_sde: (bool) Whether to use State Dependent Exploration or not
        :param log_std_init: (float) Initial value for the log standard deviation
        :param full_std: (bool) Whether to use (n_features x n_actions) parameters
            for the std instead of only (n_features,) when using gSDE
        :param sde_net_arch: ([int]) Network architecture for extracting features
            when using gSDE. If None, the latent features from the policy will be used.
            Pass an empty list to use the states as features.
        :param use_expln: (bool) Use ``expln()`` function instead of ``exp()`` to ensure
            a positive standard deviation (cf paper). It allows to keep variance
            above zero and prevent it from growing too fast. In practice, ``exp()`` is usually
        :param squash_output: (bool) Whether to squash the output using a tanh function,
            this allows to ensure boundaries when using gSDE.
        :param features_extractor_class: (Type[BaseFeaturesExtractor]) Features extractor to us
        :param features_extractor_kwargs: (Optional[Dict[str, Any]]) Keyword arguments
            to pass to the feature extractor.
        :param normalize_images: (bool) Whether to normalize images or not,
             dividing by 255.0 (True by default)
        :param optimizer_class: (Type[th.optim.Optimizer]) The optimizer to use,
            ``th.optim.Adam`` by default
        :param optimizer_kwargs: (Optional[Dict[str, Any]]) Additional keyword arguments,
            excluding the learning rate, to pass to the optimizer
        """

    def __init__(
        self,
        observation_space: gym.spaces.Space,
        action_space: gym.spaces.Space,
        lr_schedule: Callable,
        net_arch: Optional[List[Union[int, Dict[str, List[int]]]]] = None,
        device: Union[th.device, str] = "auto",
        activation_fn: Type[nn.Module] = nn.Tanh,
        ortho_init: bool = True,
        use_sde: bool = False,
        log_std_init: float = 0.0,
        full_std: bool = True,
```

```
    sde_net_arch: Optional[List[int]] = None,
    use_expln: bool = False,
    squash_output: bool = False,
    features_extractor_class: Type[NatureAE] = NatureMMD,
    features_extractor_kwargs: Optional[Dict[str, Any]] = None,
    rnd_rand_arch: Optional[List[int]] = None,
    rnd_distill_arch: Optional[List[int]] = None,
    rnd_latent_dim: Optional[int] = None,
    normalize_images: bool = True,
    optimizer_class: Type[th.optim.Optimizer] = th.optim.Adam,
    optimizer_kwargs: Optional[Dict[str, Any]] = None,
):
    super(AERndPolicy, self).__init__(
        observation_space,
        action_space,
        lr_schedule,
        net_arch,
        device,
        activation_fn,
        ortho_init,
        use_sde,
        log_std_init,
        full_std,
        sde_net_arch,
        use_expln,
        squash_output,
        features_extractor_class,
        features_extractor_kwargs,
        rnd_rand_arch,
        rnd_distill_arch,
        rnd_latent_dim,
        normalize_images,
        optimizer_class,
        optimizer_kwargs,
    )
    self.features_extractor: NatureAE = self.features_extractor
```

```
def extract_features(self, obs: th.Tensor) -> th.Tensor:
    """
    Preprocess the observation if needed and extract features.

    :param obs: (th.Tensor)
    :return: (th.Tensor)
    """
    assert self.features_extractor is not None, "No feature extractor was set"
    preprocessed_obs = preprocess_obs(
        obs, self.observation_space, normalize_images=self.normalize_images
    )
    return self.features_extractor.encode(
        preprocessed_obs)

def _get_latent(
    self, obs: th.Tensor
) -> Tuple[th.Tensor, th.Tensor, th.Tensor, th.Tensor, th.Tensor, th.Tensor]:
    """
    Get the latent code (i.e., activations of the last layer of each network)
    for the different networks.

    :param obs: (th.Tensor) Observation
    :return: (Tuple[th.Tensor, th.Tensor, th.Tensor, th.Tensor, th.Tensor]) Latent codes
        for the actor, the value function, for gSDE function, and the decoded observation
    """
    # Preprocess the observation if needed
    features = self.extract_features(obs)
    latent_pi, latent_vf = self.mlp_extractor(features)

    # Features for sde
    latent_sde = latent_pi
    if self.sde_features_extractor is not None:
        latent_sde = self.sde_features_extractor(
            features)
```

```
        latent_rnd = self.rnd_net.forward(features)
        latent_distill = self.rnd_distill_net(features)

        return latent_pi, latent_vf, latent_sde, latent_rnd, latent_distill, features

    def evaluate_actions(
        self, obs: th.Tensor, actions: th.Tensor
    ) -> Tuple[th.Tensor, th.Tensor, th.Tensor, th.Tensor, th.Tensor]:
        """
        Evaluate actions according to the current policy,
        given the observations.

        :param obs: (th.Tensor)
        :param actions: (th.Tensor)
        :return: (th.Tensor, th.Tensor, th.Tensor) estimated value, log likelihood of taking th
            entropy of the action distribution, and the rnd error
        """
        (
            latent_pi,
            latent_vf,
            latent_sde,
            latent_rnd,
            latent_distill,
            features,
        ) = self._get_latent(obs)
        distribution = self._get_action_dist_from_latent(
            latent_pi, latent_sde)
        log_prob = distribution.log_prob(actions)
        values = self.value_net(latent_vf)
        rnd_diff = (latent_rnd - latent_distill) ** 2
        ae_loss = self.features_extractor.compute_loss(
            obs, features)

        return values, log_prob, distribution.entropy(), rnd_diff, ae_loss

    def forward(
```

```
        self, obs: th.Tensor, deterministic: bool = False
    ) -> Tuple[th.Tensor, th.Tensor, th.Tensor, th.Tensor]:
        """
        Forward pass in all the networks (actor and critic)

        :param obs: (th.Tensor) Observation
        :param deterministic: (bool) Whether to sample or use deterministic actions
        :return: (Tuple[th.Tensor, th.Tensor, th.Tensor, th.Tensor])
            action, value, log probability of the action and distillation error
        """
        (
            latent_pi,
            latent_vf,
            latent_sde,
            latent_rnd,
            latent_distill,
            *_,
        ) = self._get_latent(obs)
        # Evaluate the values for the given observations
        values = self.value_net(latent_vf)
        distribution = self._get_action_dist_from_latent(
            latent_pi, latent_sde=latent_sde
        )
        actions = distribution.get_actions(
            deterministic=deterministic)
        log_prob = distribution.log_prob(actions)
        rnd_diff = ((latent_rnd - latent_distill)
                    ** 2).mean(dim=1)
        return actions, values, log_prob, rnd_diff


register_policy("RndPolicy", RndPolicy)
register_policy("CnnRndPolicy", CnnRndPolicy)
```

C.2.2 Algorithm

```
import time
from typing import List, Tuple, Type, Union, Callable, Optional, Dict, Any

import gym
from gym import spaces
import torch as th
import torch.nn.functional as F

# Check if tensorboard is available for pytorch
# TODO: finish tensorboard integration
# try:
#     from torch.utils.tensorboard import SummaryWriter
# except ImportError:
#     SummaryWriter = None
import numpy as np

from stable_baselines3.common import logger
from stable_baselines3.common.base_class import BaseRLModel
from stable_baselines3.common.type_aliases import GymEnv, MaybeCallback
from stable_baselines3.common.buffers import RolloutBuffer
from stable_baselines3.common.utils import explained_variance, get_schedule_fn
from stable_baselines3.common.vec_env import VecEnv
from stable_baselines3.common.callbacks import BaseCallback

from hierarchical_methods.ppo.policies import RndPolicy, AERndPolicy


class RndPPO(BaseRLModel):
    """
    Proximal Policy Optimization algorithm (PPO) (clip version)

    Paper: https://arxiv.org/abs/1707.06347
    Code: This implementation borrows code from OpenAI Spinning Up (https://github.com/openai/s
    https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail and
    and Stable Baselines (PPO2 from https://github.com/hill-a/stable-baselines)
```

```
Introduction to PPO: https://spinningup.openai.com/en/latest/algorithms/ppo.html

:param policy: (RndPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, ...)
:param env: (Gym environment or str) The environment to learn from (if registered in Gym, c
:param learning_rate: (float or callable) The learning rate, it can be a function
    of the current progress (from 1 to 0)
:param n_steps: (int) The number of steps to run for each environment per update
    (i.e. batch size is n_steps * n_env where n_env is number of environment copies running
:param batch_size: (int) Minibatch size
:param n_epochs: (int) Number of epoch when optimizing the surrogate loss
:param gamma: (float) Discount factor
:param gae_lambda: (float) Factor for trade-off of bias vs variance for Generalized Advanta
:param clip_range: (float or callable) Clipping parameter, it can be a function of the curr
    (from 1 to 0).
:param clip_range_vf: (float or callable) Clipping parameter for the value function,
    it can be a function of the current progress (from 1 to 0).
    This is a parameter specific to the OpenAI implementation. If None is passed (default),
    no clipping will be done on the value function.
    IMPORTANT: this clipping depends on the reward scaling.
:param ent_coef: (float) Entropy coefficient for the loss calculation
:param vf_coef: (float) Value function coefficient for the loss calculation
:param max_grad_norm: (float) The maximum value for the gradient clipping
:param use_sde: (bool) Whether to use generalized State Dependent Exploration (gSDE)
    instead of action noise exploration (default: False)
:param sde_sample_freq: (int) Sample a new noise matrix every n steps when using gSDE
    Default: -1 (only sample at the beginning of the rollout)
:param target_kl: (float) Limit the KL divergence between updates,
    because the clipping is not enough to prevent large update
    see issue #213 (cf https://github.com/hill-a/stable-baselines/issues/213)
    By default, there is no limit on the kl div.
:param tensorboard_log: (str) the log location for tensorboard (if None, no logging)
:param create_eval_env: (bool) Whether to create a second environment that will be
    used for evaluating the agent periodically. (Only available when passing string for the
:param policy_kwargs: (dict) additional arguments to be passed to the policy on creation
:param verbose: (int) the verbosity level: 0 no output, 1 info, 2 debug
:param seed: (int) Seed for the pseudo random generators
```

```
    :param device: (str or th.device) Device (cpu, cuda, ...) on which the code should be run.
        Setting it to auto, the code will be run on the GPU if possible.
    :param _init_setup_model: (bool) Whether or not to build the network at the creation of the
    :param use_ae: (bool) Whether the model is using an AutoEncoder policy
    :param ae_coef: (float) Loss coef for the auto encoder
    """

    def __init__(
        self,
        policy: Union[str, Type[RndPolicy]],
        env: Union[GymEnv, str],
        learning_rate: Union[float, Callable] = 3e-4,
        n_steps: int = 2048,
        batch_size: Optional[int] = 64,
        n_epochs: int = 10,
        gamma: float = 0.99,
        gae_lambda: float = 0.95,
        clip_range: float = 0.2,
        clip_range_vf: Optional[float] = None,
        ent_coef: float = 0.0,
        vf_coef: float = 0.5,
        rnd_coef: float = 0.5,
        max_grad_norm: float = 0.5,
        use_sde: bool = False,
        sde_sample_freq: int = -1,
        target_kl: Optional[float] = None,
        tensorboard_log: Optional[str] = None,
        create_eval_env: bool = False,
        policy_kwargs: Optional[Dict[str, Any]] = None,
        verbose: int = 0,
        seed: Optional[int] = None,
        device: Union[th.device, str] = "auto",
        _init_setup_model: bool = True,
        use_ae: bool = False,
        ae_coef: float = 0.5,
    ):
```

```
super(RndPPO, self).__init__(
    policy,
    env,
    RndPolicy,
    learning_rate,
    policy_kwargs=policy_kwargs,
    verbose=verbose,
    device=device,
    use_sde=use_sde,
    sde_sample_freq=sde_sample_freq,
    create_eval_env=create_eval_env,
    support_multi_env=True,
    seed=seed,
)

self.batch_size = batch_size
self.n_epochs = n_epochs
self.n_steps = n_steps
self.gamma = gamma
self.gae_lambda = gae_lambda
self.clip_range = clip_range
self.clip_range_vf = clip_range_vf
self.ent_coef = ent_coef
self.vf_coef = vf_coef
self.rnd_coef = rnd_coef
self.max_grad_norm = max_grad_norm
self.rollout_buffer = None
self.target_kl = target_kl
self.tensorboard_log = tensorboard_log
self.tb_writer = None
self.use_ae = use_ae
self.ae_coef = ae_coef

if _init_setup_model:
    self._setup_model()
```

```
        if use_ae:
            assert isinstance(self.policy, AERndPolicy)

    def _setup_model(self) -> None:

        self._setup_lr_schedule()
        self.set_random_seed(self.seed)

        self.rollout_buffer = RolloutBuffer(
            self.n_steps,
            self.observation_space,
            self.action_space,
            self.device,
            gamma=self.gamma,
            gae_lambda=self.gae_lambda,
            n_envs=self.n_envs,
        )

        self.policy = self.policy_class(
            self.observation_space,
            self.action_space,
            self.lr_schedule,
            use_sde=self.use_sde,
            device=self.device,
            **self.policy_kwargs,
        )
        self.policy = self.policy.to(self.device)

        self.clip_range = get_schedule_fn(self.clip_range)
        if self.clip_range_vf is not None:
            if isinstance(self.clip_range_vf, (float, int)):
                assert self.clip_range_vf > 0, (
                    "`clip_range_vf` must be positive, "
                    "pass `None` to deactivate vf clipping"
                )
```

```
            self.clip_range_vf = get_schedule_fn(
                self.clip_range_vf)


    def collect_rollouts(
        self,
        env: VecEnv,
        callback: BaseCallback,
        rollout_buffer: RolloutBuffer,
        n_rollout_steps: int = 256,
    ) -> bool:


        assert self._last_obs is not None, "No previous observation was provided"
        n_steps = 0
        rollout_buffer.reset()
        # Sample new weights for the state dependent
        # exploration
        if self.use_sde:
            self.policy.reset_noise(env.num_envs)


        callback.on_rollout_start()


        while n_steps < n_rollout_steps:
            if (
                self.use_sde
                and self.sde_sample_freq > 0
                and n_steps % self.sde_sample_freq == 0
            ):
                # Sample a new noise matrix
                self.policy.reset_noise(env.num_envs)


            with th.no_grad():
                # Convert to pytorch tensor
                obs_tensor = th.as_tensor(
                    self._last_obs).to(self.device)
                actions, values, log_probs, rnd_diff = self.policy.forward(
```

```
                obs_tensor)
        actions = actions.cpu().numpy()

        # Rescale and perform action
        clipped_actions = actions
        # Clip the actions to avoid out of bound error
        if isinstance(self.action_space,
                      gym.spaces.Box):
            clipped_actions = np.clip(
                actions, self.action_space.low, self.action_space.high
            )

        new_obs, rewards, dones, infos = env.step(
            clipped_actions)

        if callback.on_step() is False:
            return False

        self._update_info_buffer(infos)
        n_steps += 1
        self.num_timesteps += env.num_envs

        if isinstance(self.action_space,
                      gym.spaces.Discrete):
            # Reshape in case of discrete action
            actions = actions.reshape(-1, 1)

        rewards += rnd_diff.cpu().numpy()
        rollout_buffer.add(
            self._last_obs, actions, rewards, dones, values, log_probs
        )
        self._last_obs = new_obs

    rollout_buffer.compute_returns_and_advantage(
        values, dones=dones)
```

```
        callback.on_rollout_end()

        return True

    def train(self, n_epochs: int,
              batch_size: int = 64) -> None:
        # Update optimizer learning rate
        self._update_learning_rate(self.policy.optimizer)
        # Compute current clip range
        clip_range = self.clip_range(self._current_progress)
        # Optional: clip range for the value function
        if self.clip_range_vf is not None:
            clip_range_vf = self.clip_range_vf(
                self._current_progress)

        entropy_losses, all_kl_divs = [], []
        pg_losses, value_losses = [], []
        clip_fractions = []
        rnd_losses = []

        # train for gradient_steps epochs
        for epoch in range(n_epochs):
            approx_kl_divs = []
            # Do a complete pass on the rollout buffer
            for rollout_data in self.rollout_buffer.get(
                    batch_size):
                actions = rollout_data.actions
                if isinstance(self.action_space,
                              spaces.Discrete):
                    # Convert discrete action from float to
                    # long
                    actions = rollout_data.actions.long().flatten()

                # Re-sample the noise matrix because the log_std has changed
                # TODO: investigate why there is no issue with the gradient
                # if that line is commented (as in SAC)
```

```
if self.use_sde:
    self.policy.reset_noise(batch_size)

(
    values,
    log_prob,
    entropy,
    rnd_diff,
    *ae_loss,
) = self.policy.evaluate_actions(rollout_data.observations, actions)
if self.use_ae:
    ae_loss = ae_loss[0]
else:
    ae_loss = 0

values = values.flatten()
# Normalize advantage
advantages = rollout_data.advantages
advantages = (advantages - advantages.mean()) / (
    advantages.std() + 1e-8
)

# ratio between old and new policy, should
# be one at the first iteration
ratio = th.exp(
    log_prob - rollout_data.old_log_prob)
# clipped surrogate loss
policy_loss_1 = advantages * ratio
policy_loss_2 = advantages * th.clamp(
    ratio, 1 - clip_range, 1 + clip_range
)
policy_loss = - \
    th.min(policy_loss_1,
           policy_loss_2).mean()

# Logging
```

```
pg_losses.append(policy_loss.item())
clip_fraction = th.mean(
    (th.abs(ratio - 1) > clip_range).float()).item()
clip_fractions.append(clip_fraction)

if self.clip_range_vf is None:
    # No clipping
    values_pred = values
else:
    # Clip the different between old and new value
    # NOTE: this depends on the reward
    # scaling
    values_pred = rollout_data.old_values + th.clamp(
        values - rollout_data.old_values,
        - clip_range_vf, clip_range_vf
    )
# Value loss using the TD(gae_lambda) target
value_loss = F.mse_loss(
    rollout_data.returns, values_pred)
value_losses.append(value_loss.item())

# Entropy loss favor exploration
if entropy is None:
    # Approximate entropy when no analytical
    # form
    entropy_loss = -log_prob.mean()
else:
    entropy_loss = -th.mean(entropy)

entropy_losses.append(entropy_loss.item())

# rnd
rnd_loss = rnd_diff.mean()
rnd_losses.append(rnd_loss.item())

loss = (
```

```
                policy_loss
                + self.ent_coef * entropy_loss
                + self.vf_coef * value_loss
                + self.rnd_coef * rnd_loss
                + ae_loss * self.ae_coef
            )

            # Optimization step
            self.policy.optimizer.zero_grad()
            loss.backward()
            # Clip grad norm
            th.nn.utils.clip_grad_norm_(
                self.policy.parameters(), self.max_grad_norm
            )
            self.policy.optimizer.step()
            approx_kl_divs.append(
                th.mean(rollout_data.old_log_prob
                        - log_prob).detach().cpu().numpy()
            )

        all_kl_divs.append(np.mean(approx_kl_divs))

        if (
            self.target_kl is not None
            and np.mean(approx_kl_divs) > 1.5 * self.target_kl
        ):
            print(
                f"Early stopping at step {epoch} due to reaching max kl: {np.mean(approx_kl
            )
            break

    self._n_updates += n_epochs
    explained_var = explained_variance(
        self.rollout_buffer.returns.flatten(
        ), self.rollout_buffer.values.flatten()
    )
```

```
        logger.logkv("n_updates", self._n_updates)
        logger.logkv("clip_fraction",
                     np.mean(clip_fraction))
        logger.logkv("clip_range", clip_range)
        if self.clip_range_vf is not None:
            logger.logkv("clip_range_vf", clip_range_vf)

        logger.logkv("approx_kl", np.mean(approx_kl_divs))
        logger.logkv("explained_variance", explained_var)
        logger.logkv("entropy_loss",
                     np.mean(entropy_losses))
        logger.logkv("policy_gradient_loss",
                     np.mean(pg_losses))
        logger.logkv("value_loss", np.mean(value_losses))
        if hasattr(self.policy, "log_std"):
            logger.logkv("std", th.exp(
                self.policy.log_std).mean().item())
        logger.logkv("rnd_loss", np.mean(rnd_losses))

    def learn(
        self,
        total_timesteps: int,
        callback: MaybeCallback = None,
        log_interval: int = 1,
        eval_env: Optional[GymEnv] = None,
        eval_freq: int = -1,
        n_eval_episodes: int = 5,
        tb_log_name: str = "PPO",
        eval_log_path: Optional[str] = None,
        reset_num_timesteps: bool = True,
    ) -> "PPO":

        iteration = 0
        callback = self._setup_learn(
            eval_env,
```

```
    callback,
    eval_freq,
    n_eval_episodes,
    eval_log_path,
    reset_num_timesteps,
)

# if self.tensorboard_log is not None and SummaryWriter is not None:
#     self.tb_writer = SummaryWriter(log_dir=os.path.join(self.tensorboard_log, tb_log_r

callback.on_training_start(locals(), globals())

while self.num_timesteps < total_timesteps:

    continue_training = self.collect_rollouts(
        self.env, callback, self.rollout_buffer, n_rollout_steps=self.n_steps
    )

    if continue_training is False:
        break

    iteration += 1
    self._update_current_progress(
        self.num_timesteps, total_timesteps)

    # Display training infos
    if (
        self.verbose >= 1
        and log_interval is not None
        and iteration % log_interval == 0
    ):
        fps = int(self.num_timesteps
                  / (time.time() - self.start_time))
        logger.logkv("iterations", iteration)
        if len(self.ep_info_buffer) > 0 and len(
                self.ep_info_buffer[0]) > 0:
```

```
                logger.logkv(
                    "ep_rew_mean",
                    self.safe_mean(
                        [ep_info["r"]
                            for ep_info in self.ep_info_buffer]
                    ),
                )
                logger.logkv(
                    "ep_len_mean",
                    self.safe_mean(
                        [ep_info["l"]
                            for ep_info in self.ep_info_buffer]
                    ),
                )
            logger.logkv("fps", fps)
            logger.logkv("time_elapsed", int(
                time.time() - self.start_time))
            logger.logkv("total␣timesteps",
                         self.num_timesteps)
            logger.logkv("episodes", len(
                self.ep_info_buffer))
            logger.dumpkvs()

        self.train(self.n_epochs,
                   batch_size=self.batch_size)

        # For tensorboard integration
        # if self.tb_writer is not None:
        #     self.tb_writer.add_scalar('Eval/reward', mean_reward, self.num_timesteps)

    callback.on_training_end()

    return self

def get_torch_variables(
        self) -> Tuple[List[str], List[str]]:
```

```
        """
        cf base class
        """
        state_dicts = ["policy", "policy.optimizer"]

        return state_dicts, []
```

C.3 Modules

C.3.1 Nature AutoEncoder

```
from typing import Tuple, Optional
from abc import abstractmethod

import gym
import torch as th
from torch import nn
from torch.nn import functional as F
import torchlayers as tl

from stable_baselines3.common.policies import BaseFeaturesExtractor
from stable_baselines3.common.preprocessing import is_image_space


def nature_decoder():
    return nn.Sequential(nn.Flatten(), )


class Reshaper(nn.Module):
    def __init__(self, shape):
        super().__init__()
        self._features_dim = shape

    def forward(self, observations: th.Tensor) -> th.Tensor:
        return observations.reshape(
            observations.shape[0], *self.features_dim)
```

```
    @property
    def features_dim(self):
        return self._features_dim


class NatureAE(BaseFeaturesExtractor):
    """
    CNN from DQN nature paper: https://arxiv.org/abs/1312.5602

    :param observation_space: (gym.Space)
    :param features_dim: (int) Number of features extracted.
        This corresponds to the number of unit for the last layer.
    """

    def __init__(self, observation_space: gym.spaces.Box,
                 features_dim: int = 512):
        super(NatureAE, self).__init__(
            observation_space, features_dim)
        # We assume CxWxH images (channels first)
        # Re-ordering will be done by pre-preprocessing or
        # wrapper
        assert is_image_space(observation_space), (
            'You should use NatureMMD '
            f'only with images not with {observation_space} '
            '(you are probably using `CnnPolicy` instead of `MlpPolicy`)')
        n_input_channels = observation_space.shape[0]
        self.encode = nn.Sequential(
            nn.Conv2d(n_input_channels, 32,
                      kernel_size=8, stride=4, padding=0),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4,
                      stride=2, padding=0),
            nn.ReLU(),
            nn.Conv2d(64, 32, kernel_size=3,
                      stride=1, padding=0),
```

```
            nn.ReLU(),
            nn.Flatten(),
            tl.Linear(features_dim)
        )

        # Compute shape by doing one forward pass
        tl.build(self.encode, th.as_tensor(
            observation_space.sample()[None]).float())

        self.decode = nn.Sequential(
            tl.Linear(128), nn.ReLU(), Reshaper((2, 8, 8)),
            tl.ConvTranspose(64, kernel_size=3,
                             stride=1, padding=0), nn.ReLU(),
            tl.ConvTranspose(32, kernel_size=4,
                             stride=2, padding=1), nn.ReLU(),
            tl.ConvTranspose(n_input_channels,
                             kernel_size=8, stride=4, padding=0)
        )
        tl.build(self.decode, self.encode(th.as_tensor(
            observation_space.sample()[None]).float()))

    def forward(
            self, observations: th.Tensor) -> Tuple[th.Tensor, th.Tensor]:
        enc = self.encode(observations)
        dec = self.decode(enc)
        return enc, dec

    def compute_loss(self, observations: th.Tensor,
                     features: Optional[th.Tensor] = None) -> th.Tensor:
        if features is None:
            features, dec = self(observations)
        else:
            dec = self.decode(features)

        dec = dec.reshape(observations.shape[0], -1)
        loss = F.mse_loss(observations.reshape(
```

```
            observations.shape[0], -1), dec)
        return loss


class NatureMMD(NatureAE):
    def __init__(self, observation_space: gym.spaces.Box,
                 features_dim: int = 32, kernels=10):
        super().__init__(observation_space, features_dim)
        self.kernels = kernels

    def compute_loss(self, observations: th.Tensor,
                     features: Optional[th.Tensor] = None) -> th.Tensor:

        if features is None:
            features, rec = self(observations)
        else:
            rec = self.decode(features)

        def gaussian_kernel(a, b):
            dim1_1, dim1_2 = a.shape[0], b.shape[0]
            depth = a.shape[1]
            a = a.view(dim1_1, 1, depth)
            b = b.view(1, dim1_2, depth)
            a_core = a.expand(dim1_1, dim1_2, depth)
            b_core = b.expand(dim1_1, dim1_2, depth)
            numerator = (
                a_core - b_core).pow(2).mean(2) / depth
            return th.exp(-numerator)

        Jrec = F.mse_loss(
            rec, observations, reduction="mean")

        kernel = th.randn(
            self.kernels, features.shape[1], requires_grad=False
        ).to(features.device)
```

```
        g1 = gaussian_kernel(kernel, kernel).mean()
        g2 = gaussian_kernel(features, features).mean()
        g3 = -2 * gaussian_kernel(kernel, features).mean()
        Jmmd = g1 + g2 + g3
        return Jrec + Jmmd
```

C.4 Utilities

C.4.1 VecNoise

```
from typing import Optional, List, Iterable
from abc import ABC, abstractmethod
import copy

import numpy as np
from stable_baselines3.common.noise import ActionNoise


class VectorizedActionNoise(ActionNoise):
    """
    A Vectorized action noise for parallel environments.

    :param base_noise: ActionNoise The noise generator to use
    :param n_envs: (int) The number of parallel environments
    """

    def __init__(self, base_noise: ActionNoise,
                 n_envs: int):
        super().__init__()
        try:
            self.n_envs = int(n_envs)
            assert self.n_envs > 0
        except (TypeError, AssertionError):
            raise ValueError(
                f"Expected n_envs={n_envs} to be positive integer greater than 0"
            )
```

```
        self.base_noise = base_noise
        self.noises = [copy.deepcopy(self.base_noise)
                       for _ in range(n_envs)]

    def reset(
            self, indices: Optional[Iterable[int]] = None) -> None:
        """
        Reset all the noise processes, or those listed in indices

        :param indices: Optional[Iterable[int]] The indices to reset. Default: None.
            If the parameter is None, then all processes are reset to their initial position.
        """
        if indices is None:
            indices = range(len(self.noises))

        for index in indices:
            self.noises[index].reset()

    def __repr__(self) -> str:
        return (
            f"VecNoise(BaseNoise={repr(self.base_noise)}), n_envs={len(self.noises)})"
        )

    def __call__(self) -> np.ndarray:
        """
        Generate and stack the action noise from each noise object
        """
        noise = np.stack([noise() for noise in self.noises])
        return noise

    @property
    def base_noise(self) -> ActionNoise:
        return self._base_noise

    @base_noise.setter
```

```
def base_noise(self, base_noise: ActionNoise):
    if base_noise is None:
        raise ValueError(
            "Expected␣base_noise␣to␣be␣an␣instance␣of␣ActionNoise,␣not␣None",
            ActionNoise,
        )
    if not isinstance(base_noise, ActionNoise):
        raise TypeError(
            "Expected␣base_noise␣to␣be␣an␣instance"
            "of␣type␣ActionNoise", ActionNoise
        )
    self._base_noise = base_noise

@property
def noises(self) -> List[ActionNoise]:
    return self._noises

@noises.setter
def noises(self, noises: List[ActionNoise]) -> None:
    # raises TypeError if not iterable
    noises = list(noises)
    assert (
        len(noises) == self.n_envs
    ), f"Expected␣a␣list␣of␣{self.n_envs}␣ActionNoises,␣found␣{len(noises)}."

    different_types = [
        i
        for i, noise in enumerate(noises)
        if not isinstance(noise, type(self.base_noise))
    ]

    if len(different_types):
        raise ValueError(
            f"Noise␣instances␣at␣indices␣{different_types}"
            + "don't␣match␣the␣type␣of␣base_noise",
            type(self.base_noise),
```

```
        )

    self._noises = noises
    for noise in noises:
        noise.reset()
```