The Lottery Ticket Hypothesis and value-based Deep Reinforcement Learning

Auteur : Debes, Baptiste
Promoteur(s) : Louppe, Gilles
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en science des données, à finalité spécialisée
Année académique : 2021-2022
URI/URL : http://hdl.handle.net/2268.2/13872

University of Liège
Faculty of Applied Sciences
Department of Electrical Engineering and Computer Science

The Lottery Ticket Hypothesis and value-based Deep Reinforcement Learning

A dissertation submitted in partial fulfillment of the requirements for the degree of
Master of Science in Data Science and Engineering

by

BAPTISTE DEBES

Supervisor:
Prof. Gilles Louppe

Academic Year 2021-2022

Abstract

The Lottery Ticket Hypothesis (LTH) suggests that randomly initialized overparametrized neural networks contain subnetworks which - when trained in isolation - are able to perform better than similar subnetworks whose architecture and weights are drawn randomly. Subnetworks matching the Lottery Ticket Hypothesis are referred to as winning tickets because they are the winners of the initialization lottery. An algorithm called Iterative Magnitude Pruning (IMP) was introduced to discover winning tickets. Finding well-performing sparse neural networks is especially interesting because of the potential large reduction in memory footprint and global computational burden. These combined may lead to an important reduction of the energy required to perform a same task. Deep Reinforcement Learning (DRL) has introduced algorithm capable of solving complex tasks (dynamic system control, Atari games, board games, ...). In this work we study the combination of deep reinforcement learning and the lottery ticket hypothesis. We focus on two algorithms namely Double Deep Q-Networks (DDQN) and Soft-Actor-Critic (SAC) which both belong to the fruitful class of value-based methods. We provide the third independent confirmation - in the context of deep reinforcement learning - of the existence of subnetworks matching the Lottery Ticket Hypothesis using Iterative Magnitude Pruning. Our experiments were carried on standard classic control as well as pixel-based environments. We provide experiments and guidelines regarding some important hyperparameters. We suggest a potential ability of winning tickets to robustly preserve low rank embeddings of the environment's state space. Some of ours results suggest that tickets found using IMP seem closer than expected to subnetworks that could be found using so-called structured pruning methods. Our experiments also showcase the ability of winning tickets to render inactive useless input variables while keeping good performance on the task. This result along with others indicate a potential ability of winning tickets to be used as feature importance extractors. Finally, a variant of Iterative Magnitude Pruning is introduced which we call pooled pruning. We suggest this variant could be beneficial for multi-networks algorithms such as Soft-Actor-Critic.

Acknowledgments

I would like to express my gratitude to my advisor Professor Gilles LOUPPE for guiding me while giving me the freedom to carry out this research my way. I am grateful I had this opportunity to try research on a topic I wish to spend the next few years of my life working on.
I would also like to thank Matthia SABATELLI for introducing me to the lottery ticket hypothesis and suggesting its combination with deep reinforcement learning. I am grateful for his early involvement and guidance.

Contents

Chapter 1

Introduction

Deep reinforcement learning has been a rapidly advancing field during the last ten years. Using deep neural networks as universal function approximators has allowed the emergence of algorithms capable of learning tasks which were previously unsolvable (Mnih et al. 2013, Hessel et al. 2017, Lillicrap et al. 2019, Silver et al. 2016, Schrittwieser et al. 2019). Combined with availability of cheaper powerful and some important algorithmic breakthroughs, deep reinforcement learning (DRL) has emerged as an incredibly prolific area for both research and the industry (Luong et al. 2019, Kiran et al. 2020, Zoph and Le 2016). It is now possible to build agent able to learn the control of complex physical systems for robotics and automation. Deep reinforcement learning agents have even been able to reach or surpass human performance in game-based benchmarks (Badia et al. 2020, Silver et al. 2016).

The Lottery Ticket Hypothesis (LTH) has been introduced recently in [Frankle and Carbin, 2018] offers some fresh air in the literature on model compression and network pruning. The LTH suggests the existence of special sparse initializations which - when trained in isolation - are able to obtain significantly better performance than some other sparse initializations with commensurate number of parameters. An algorithm called Iterative Magnitude Pruning (IMP) has been introduced and is able to extract sparse initializations matching this hypothesis. These especially well performing initial parameters are referred to as *winning tickets* since then won the lottery of initializations. The LTH was first introduced in the context of supervised learning for image classification. However, using iterative magnitude pruning, research has successfully shown the existence of winning tickets in many areas such as Natural Language Processing and Deep Reinforcement Learning (Yu et al. 2020).

The Lottery Ticket Hypothesis and model sparsification in general bring many benefits. On the one hand, the existence of sparse but well-performing initializations suggest a large gap for improvement in the field of initialization schemes. On the other hand, it may be a response to the existing trends of increase in model sizes and the adoption of neural networks on low power hardware (IoT, mobile phones, drones, ...). Indeed, removing a large fraction of model parameters may reduce memory footprint as well as the number of floating point operations required to use a model. Both of these improvements may help to dramatically reduce energy consumption (Han et al. 2015).

This work studies the existence of winning tickets in the context of deep reinforcement learning. Our study focuses on a fruitful class of DRL algorithms called value-based methods. Two algorithms in particular are going to be discussed namely Double Deep Q-Networks (DDQN) and Soft-Actor-Critic (SAC) which are both standards in the field of value-based DRL. In this work we confirm previous results on the existence of winning tickets for this class of algorithms. Additionally, we propose to dissect and observe some key properties exhibited by the winning tickets found using iterative magnitude pruning.

Outline

This work is divided in five core chapters. In Chapter 2, the two algorithms DDQN and SAC are introduced. An important effort of this work has been conducted on the understanding and implementation of these methods. For this reason - in this chapter - the two algorithms will be introduced from the ground and from first principles. This chapter is dedicated to interested readers such that knowledge on value-based DRL is sufficient to skip it. In Chapter 3, the formal Lottery Ticket Hypothesis is introduced. In this chapter we also provide the procedure called IMP used to find winning tickets along with a set of features and hyperparameters known to be critical in order to find these tickets successfully. Follows Chapter 4 where we discuss the experimental framework the results of this work were generated from. We introduce, the implementation required to withstand the heavy computational burden involved by Iterative Magnitude Pruning. Then, we also discuss some critical implementation decisions, network architectures and the benchmark environments on which we experimented. In Chapter 5, the contributions and results of this work are presented. We show we were able to successfully confirm the existence of winning tickets for both DDQN and SAC. We discuss, some important hyperparameters and decisions which were introduced in the previous chapter. Subsequently, we also discuss the *sample efficiency* of the pruned agents - the ability to learn a task with a minimal number of interactions with the environment. Then follows a study on some properties of the masks found by IMP. Finally, we exhibit the ability of iterative magnitude pruning to put emphasis on some variables provided by the environment. It is able to remove useless or redundant variables while preserving a good part of the final performance and being efficient at learning the task. Finally, Chapter 6, summarizes and concludes on the observations made in this work. There we also discuss the limitations and some new exciting research questions.

Chapter 2

Reinforcement learning

2.1 What is reinforcement learning

Reinforcement learning is a field that studies problems and solution related to the development of intelligent agents taking actions in some environment and whose purpose is to maximize a cumulative reward signal.

A broad variety of reinforcement learning problems can be abstracted through the notion of Markov Decision Process to model the components of the problem. A MDP models an agent acting sequentially in an environment whose dynamics can be stochastic. The actions are drawn from a mapping from states to action which can - in the most general case - be a probability distribution. This mapping is called a *policy*. The reward signal is the only feed-back received by the agent regarding its behaviour and how it is expected to act. More visually, the agent acts according to a perception-action loop as depicted in Figure 2.1. When the loop is unrolled, the sequence of states, actions and rewards is a Markov chain.

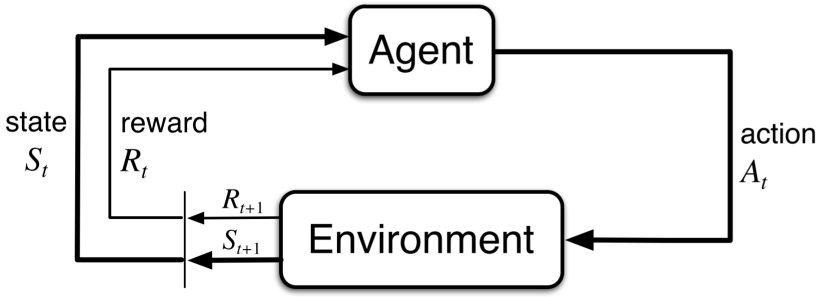

Figure 2.1: The perception-action loop. Describes the sequential nature of the problem. The agent acts upon its environment based on the perception it has of it. Afterward, the environment updates its state and send a reward signal to the agent. Scheme from [Sutton and Barto, 2018, Chapter 3.1]

Markov Decision Process Formally, a Markov Decision Process is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P})$ whose components are

- A state-space $s \in \mathcal{S}$
- An action-space $a \in \mathcal{A}$
- A probability distribution over next states and immediate rewards

$$\mathcal{P} \triangleq P(S_{t+1} = s',\, R_t = r \,|\, S_t = s,\, A_t = a) \tag{2.1}$$

which is sometimes split between $P(S_{t+1} = s' \,|\, S_t = s,\, A_t = a)$ and $P(R_t = r \,|\, S_t = s,\, A_t = a)$.

In this work we will work transparently with both finite and infinite-horizons MDPs. As will be discussed later an additional component called the *discount factor* is needed. Thus, a *discounted* Markov Decision Process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \gamma)$ with $\gamma \in [0, 1]$.

Equation 2.1 illustrates the markovian property of the abstraction. New rewards and states depends only on immediate values of the states and actions and not on values preceding this moment, this is the *first-order property*. Furthermore, the probability distribution is independent of time, the dynamics is always the same. This is called the *stationarity property*.

In this work we implicitly make the assumption that the environment is fully observable such that there is not hidden internal states to be described. Hence, the state variables are always considered to be sufficient to obtain the first-order property.

In many developments, the reward signal is only used through its expectation $\mathbb{E}_{P(r \,|\, s,\, a)}[r] \triangleq R(s, a)$. The reward signal is often assumed to be bounded such that $R(s,\, a) \in [-R_{\text{MAX}},\, R_{\text{MAX}}]$. For these reasons, there is another slightly different definition for a discounded Markov Decision Process which is the 5-tuple $(\mathcal{S},\, \mathcal{A},\, \mathcal{P},\, \mathcal{R},\, \gamma)$ with $\mathcal{P}$ the state dynamics $P(S_{t+1} = s' \,|\, S_t = s,\, A_t = a)$ and $\mathcal{R}$ the expected reward signal. We will use both ways to formalize a discounted MDP throughout this work.

2.2 The reinforcement learning problem

In this section we formalize the reinforcement learning problem, its objective and the components required to model the problem. The purpose of reinforcement learning is to derive a mapping from states to actions called *policy* which maximizes the expected sum of reward. In the most general case, the policy can be considered as a probability distribution over actions given the state. This distribution will be written $\pi(a|s)$. The reinforcement learning objective is an expectation over the environment dynamics as well as the policy distribution. We denote $V^{\pi}(s)$ the *state value function* of policy π as the expected sum of reward starting from state s and following policy π. More formally we write,

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\substack{a_t \sim \pi(\,.\,|s_t) \\ s_{t+1} \sim p(\,.\,|s_t,\, a_t)}} \left[\sum_{t=0}^{T} R(s_t, a_t) \Big| s_0 = s \right] \tag{2.2}$$

This quantity is well defined and useful only for finite-horizons tasks. In order to enforce convergence in the case of infinite horizon tasks we make use of a discount factor $\gamma \in [0, 1]$ which will weight down rewards received later into the future. Thus the state-value function is defined as

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\substack{a_t \sim \pi(\,.\,|s_t) \\ s_{t+1} \sim p(\,.\,|s_t,\, a_t)}} \left[\lim_{T \longrightarrow \infty} \sum_{t=0}^{T} \gamma^t \, R(s_t, a_t) \Big| s_0 = s \right] \tag{2.3}$$

If the reward is assumed to be bounded in absolute value by $R_{\text{MAX}} \geq 0$, this sum can be shown to be upper bounded by $\frac{R_{\text{MAX}}}{1-\gamma}$. The state-value function define in equation 2.3 is said to be the *discounted reward* state-value function. It can also be defined similarly for the finite horizon case. In this work we will exclusively make use of the discounted reward version.

A convenient way to write the value-function for both finite and infinite horizon is to consider the new objects $\rho^{\pi}(s_t)$ and $\rho^{\pi}(s_t,\, a_t)$ the *state marginal* and *state-action marginal*. These are the probability distributions to observe state s_t or state-action pair $(s_t,\, a_t)$ at time t by acting according to $\pi(\,.\,|\,s)$. It appears - using the state-action marginal - the value function $V^{\pi}(s)$ for finite-horizon can be written as

$$V^{\pi}(s) = \sum_{t=0}^{T} \gamma^t \mathop{\mathbb{E}}_{\rho^{\pi}(s_t,\, a_t)} R(s_t,\, a_t) \tag{2.4}$$

Another useful way to write the state-value function is to decompose the expected immediate reward from the ones coming afterward. In the case of finite-horizon and discounted reward we get,

$$\begin{aligned} V^{\pi}(s) &= \mathop{\mathbb{E}}_{\substack{a_t \sim \pi(\,.\,|s_t) \\ s_{t+1} \sim p(\,.\,|s_t, a_t)}} \left[R(s_0, a_0) + \sum_{t=1}^{T} \gamma^t \; r(s_t, a_t) \Big| s_0 = s \right] \\ &= \mathop{\mathbb{E}}_{\substack{a_t \sim \pi(\,.\,|s_t) \\ (s_{t+1}, r_t) \sim p(\,.\,|s_t, a_t)}} \Big[R(s_0, a_0) + \gamma \; V^{\pi}(s_1) \Big| s_0 = s \Big] \end{aligned} \tag{2.5}$$

Equation 2.5 introduces a decomposition of the value function equation as an expected reward plus an expectation over this same estimator. This idea is especially interesting because it will appear in most of the algorithms covered in the next section as well as in the two practical algorithms studied in this work. Using an estimator in some locations of the state space - through the expectation - in order to compute an estimate of this estimator in another location is called *bootstrapping*. It is to be noted that in the case of equation 2.5, this decomposition is exact.

Reinforcement learning objective Here we discuss the quantity an optimal agent maximizes. This quantity is called the *reinforcement learning objective*. However, it appears this notion of optimality coexists with another one. We will call this other notion *standard optimality* because it is mostly used in well-established references such as [Sutton and Barto, 2018]. Whether these two notions are overlapping will not be a consideration.

A first way to define the reinforcement learning objective is to first define a notion of trajectory and trajectory distribution. We define a trajectory as a sequence of states and actions which is written as $\tau \triangleq (s_0, a_0, s_1, a_1, ...)$. The distribution over trajectories of policy π is denoted $p^{\pi}(\tau)$. The notion of trajectory also requires to define a distribution over initial state $P(s_0)$. It is also useful to denote the discounted cumulative reward of a trajectory as $R(\tau) \triangleq \sum_{t=0}^{T} \gamma^t r_t$ with $T \to \infty$ in the infinite-horizon case. From this new notion we can define the reinforcement learning objective as

$$J(\pi) \triangleq \mathop{\mathbb{E}}_{\tau \sim p^{\pi}(\tau)} [R(\tau)] \tag{2.6}$$

Using equation 2.6 we can define the reinforcement maximization problem as finding a policy that maximizes the reinforcement objective given a class of potential policies. Formally, this is written as

$$\pi^* \in \arg\max_{\pi' \in \Pi} \mathop{\mathbb{E}}_{\tau \sim p^{\pi'}(\tau)} [R(\tau)] \tag{2.7}$$

A policy π is optimal given a policy class Π and a distribution over initial states $P(s_0)$ if there exists no other policy $\pi' \in \Pi$ such that

$$\mathop{\mathbb{E}}_{\tau \sim p^{\pi'}(\tau)} [R(\tau)] \quad > \quad \mathop{\mathbb{E}}_{\tau \sim p^{\pi}(\tau)} [R(\tau)] \tag{2.8}$$

Standard optimality The standard optimality as can be found in (Russell and Norvig 2009, p. 650 ; Ernst et al. 2005 ; Sutton and Barto 2018, p. 62-63), considers a policy π to be optimal in policy class Π if

$$V^{\pi}(s) \;\geq\; V^{\pi'}(s) \quad \forall\, \pi' \in \Pi, \;\; \forall\, s \in \mathcal{S} \tag{2.9}$$

Consequently, with this form of optimality, the reinforcement learning problem is defined as finding a policy π^* such that

$$V^{\pi^*}(s) = \max_{\pi \in \Pi} \; V^{\pi}(s) \quad \forall\, s \in \mathcal{S} \tag{2.10}$$

Another way to look at this form of policy optimality is through the *Bellman optimality equation*. On the one hand, the idea of an optimal policy is that there exists no other one whose state-value function is larger *for any initial states*. On the other hand, Bellman optimality equation proposes to impose a form of self-consistency on the state-value function. Indeed, an optimal optimal policy can be viewed as a policy that acts optimally in every move. Hence, a policy which is optimal takes an action that leads to the most discounted reward immediately and in the future. Let's denote an optimal value function by $V^* = V^{\pi^*}$, then this idea can be written down as,

$$V^*(s) = \max_a \mathop{\mathbb{E}}_{s_{t+1} \sim p(\,.\,|s_t, a_t)} \Big[R(s_t, a_t) + \gamma\, V^*(s_{t+1}) \,\Big|\, s_t = s,\; a_t = a \Big] \tag{2.11}$$

2.3 Some definitions

In this section we provide some additional definitions and concepts that will be used throughout this work.

2.3.1 The state-action value function

In equations 2.2 and 2.3 the state value function was introduced. Here we define the *state-action value function* which is similar except it locks the first action in the sequence. It is written as $Q^\pi(s, a)$ and is defined as follows for discounted rewards and a finite horizon,

$$Q^\pi(s, a) \triangleq \mathop{\mathbb{E}}_{\substack{a_t \sim \pi(\,.\,|s_t) \\ s_{t+1} \sim p(\,.\,|s_t, a_t)}} \Big[\sum_{t=0}^{T} R(s_t, a_t) \,\Big|\, s_0 = s,\; a_0 = a \Big] \tag{2.12}$$

The state value function and state-action value function are related. It is even more obvious that the state-action value function is equivalent to the state value function except that the first action is locked by looking at the following,

$$Q^\pi(s, a) \triangleq \mathop{\mathbb{E}}_{s_{t+1} \sim p(\,.\,|s_t, a_t)} \Big[R(s,\; a) + \gamma\, V^\pi(s_1) \,\Big|\, s_0 = s,\; a_0 = a \Big] \tag{2.13}$$

Optimality criterion An optimality criterion similar to 2.11 can be written for state-action value function. It can be written as

$$Q^*(s, a) = \mathop{\mathbb{E}}_{s_{t+1} \sim p(\,.\,|s_t, a_t)} \Big[R(s_t, a_t) + \gamma\; \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \Big| s_t = s,\; a_t = a \Big] \tag{2.14}$$

2.4 Dynamic programming

In this section we describe a general scheme to solve tabular Markov Decision Processes (MDPs) whose dynamics are perfectly known. This scheme is the foundation of reinforcement learning algorithm for which the two previous assumptions may not hold.

2.4.1 The general computational scheme

The algorithm mentioned is this section are all based on the use of either a state-value function or state-action value function. These algorithms will alternate between two steps. The first one being a form of *policy evaluation* where the performance of the policy is evaluated. The second one is called a *policy improvement* step where the policy is updated such that its performance should improve or stay equal. This idea is depicted in Figure 2.2 These two steps will be described in Section 2.4.2-2.4.3. Subsequently, these two concepts will be used in two concrete algorithms to solve a MDP. The first one is called *policy iteration*, the second is coined *value-iteration*.

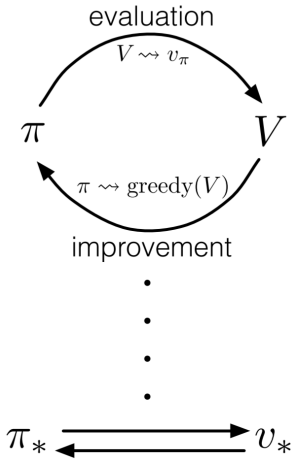

Figure 2.2: The generalized policy iteration scheme as introduced in [Sutton and Barto, 2018, Chapter 4.6]. The scheme alternate between a policy evaluation step where the performance of the policy is assessed by fitting a value function. Then, the policy is improved according to that value function in such a way that its performance should improve or at least stay equal.

2.4.2 Value evaluation

The purpose of value-evaluation is, given a policy π and some known environment dynamics, to obtain $V^\pi(s)$. As the MDP is assumed to be finite, equation 2.5 expectation can be expressed explicitly as,

$$\begin{aligned} V^\pi(s) &= \mathop{\mathbb{E}}_{\substack{a_t \sim \pi(\,.\,|s_t) \\ (s_{t+1}, r_t) \sim p(\,.\,,.\,|s_t, a_t)}} \Big[r(s_0, a_0) + \gamma\, V^\pi(s_1) \Big| s_0 = s \Big] \\ &= \sum_a \pi(a|s) \sum_{s',\, a} p(s',\ r \,|\, s,\ a)\, [\, r + \gamma\, V^\pi(s')\,] \end{aligned} \tag{2.15}$$

One way to solve equation 2.15 is to apply an iterative method that simply applies equation 2.15 as an update rule over the current estimate V_k^π. Thus we get an algorithm called *iterative policy evaluation* which applies the update rule provided in equation 2.16 on any state $s \in S$.

$$V_{k+1}^\pi(s) = \sum_a \pi(a|s) \sum_{s',\, r} p(s',\ r \,|\, s,\ a)\, [\, r + \gamma\, V_k^\pi(s')\,] \tag{2.16}$$

An equivalent way to define this update is to apply an operator $\mathcal{T}_v^\pi$ called the *Bellman backup operator* for policy π. It is define as

$$\mathcal{T}_v^\pi V(s) \triangleq \mathop{\mathbb{E}}_{(s',r) \sim p(\,.\,,\,.\,|\,s,a)} [r + \gamma V(s')] \tag{2.17}$$

Hence, value evaluation can be seen as repeatedly applying operator $\mathcal{T}_v^\pi$ on the current estimate $V_k^\pi(s)$ on some states. Several choices can be made on the order of the updates. The full state-space may be updated in one-shot, which requires a second array to store the value table. Otherwise, states may be updated in any order. This order could simply be sequential, looping in a regular pattern, random or following some kind of exploration of the environment. This version does not require a second array and is thus referred as *inplace*. A full algorithm pseudo-code for the inplace version is provided in [Sutton and Barto, 2018, p. 75]. Values are initialized arbitrarily. Then, it applies update 2.16 iteratively on each state by sweeping through the state space. Updates are performed until some kind of convergence criterion is satisfied.

The state-action version An update similar to equation 2.16 can be derived for the state-action value function. This time the Bellman backup operator is defined as

$$\mathcal{T}_q^\pi Q(s,\,a) \triangleq R(s,\,a) + \gamma \mathop{\mathbb{E}}_{s' \sim p(s'|\,s,a)} \left[\mathop{\mathbb{E}}_{a' \sim \pi(\,.\,|\,s)} [Q(s', a')] \right] \tag{2.18}$$

and the value evaluation algorithm will perform updates such that $Q_{k+1}^\pi = T_q^\pi Q_k^\pi$.

2.4.3 Policy improvement

In order to introduce the idea of policy improvement we find it easier to first present an object called the *advantage function*. This function is simply defined for a policy π as

$$A^\pi(s,a) \triangleq Q^\pi(s,a) - V^\pi(s) \tag{2.19}$$

$$= Q^\pi(s,a) - \mathop{\mathbb{E}}_{a \sim \pi(\,.\,|s)} [Q^\pi(s,\,a)] \tag{2.20}$$

This function assesses how good is action a followed by acting according to the policy compared to the *average move*. The average move is the expectation of the Q-function of π over a the distribution on actions induced by π. Hence, the larger $A^\pi(s,a)$, the better is action a in state s compared to the expected performance of π starting from state s.

Since $A(s,a)$ gives an idea of whether a policy performance increases by taking an action instead of acting according to the policy, it seems very natural to use this object in order to improve π. This principle can be summarized through the following update equation which is one way to do,

$$\pi'(a_t|s_t) = \begin{cases} 1 & \text{if } \mathrm{a}_t = \arg\max_{a_t} A^\pi(s_t,\,a_t) \\ 0 & \text{otherwise} \end{cases} \tag{2.21}$$

With this update all the probability mass is moved toward the action that maximizes the advantage. For a deterministic policy it simply changes the value of the policy in a given state. It turns out that $\arg\max_{a_t} A^\pi(s_t,\,a_t) = \arg\max_{a_t} Q^\pi(s_t,\,a_t)$. Hence, it is equivalent to look at the maximum of the Q-function. It appears that update equation 2.21 - either using $Q^\pi(s,\,a)$ or $A^\pi(s,\,a)$ - is ensured to generate a policy $\pi'(s,\,a)$ which is better than or a least as good as $\pi(s,\,a)$. We write this as $V^{\pi'}(s) \geq V^\pi(s)$ for any $s \in S$. This result follows from the *policy improvement theorem* as described in [Sutton and Barto, 2018, Chapter 4.2].

2.4.4 Policy iteration

The scheme introduced by *policy iteration* is the basis of all the reinforcement learning algorithms that will be covered in this work. As introduced in Section 2.4.1, dynamic programming usually alternates between a policy evaluation and a policy improvement step. At the first iteration, policy evaluation starts with a random guess for $V^\pi(s)$ which is then replaced by $V_k^\pi(s)$ for iteration $k+1$. Policy improvement is carried on until the policy becomes stable which is guaranteed to happen. Indeed, since the MDP is assumed to be finite, there is a finite number of policies. Moreover, policy improvement is a process that increases or at least keeps performance equal. Thus, the policy must converge to an optimal policy. One might wonder how to connect policy evaluation which outputs $V^\pi(s)$ to policy improvement which requires $A^\pi(s,a)$ or $Q^\pi(s,a)$. This is simply done by using identity 2.13 since the dynamics $p(s_{t+1},\ r_t|s_t, a_t)$ is known. More details and pseudo-code can be found in [Sutton and Barto, 2018, Chapter 4.3].

2.4.5 Value iteration

Value iteration is a relaxation of policy iteration. It appears that value evaluation only converges in the limit, for this reason a convergence criterion was set to cut the computation short. Value iteration simply cuts the value estimation step after *one* iteration. In this case, the policy can be made implicit, derived from the value function. For every state and action, the algorithm alternates between the two steps

1.
$$Q_{k+1}^\pi(s,\ a) = \sum_{s',\ r} p(s',\ r\,|\,s,\ a)\,[\,r + \gamma\, V_k^\pi(s')\,] \tag{2.22}$$

2.
$$V_{k+1}^\pi(s) = \max_a Q_{k+1}^\pi(s,\ a) \tag{2.23}$$

which can be written in one update equation that looks a lot like Bellman optimality criterion in equation 2.11

$$V_{k+1}^\pi(s) = \max_a \sum_{s',\ r} p(s',\ r\,|\,s,\ a)\,[\,r + \gamma\, V_k^\pi(s')\,] \tag{2.24}$$

Value-iteration will be the foundation for algorithms that do not assume the dynamics is available and learn from experiences.

2.5 Learning from observations only

In this section we introduce the basic concepts necessary understand how to learn optimal behaviour from observations. The first important distinction is to be made between methods which learns a model similar to 2.1 and those which do not. The former are called model-based methods, the latter are coined model-free methods. In this work we will only make use of some model-free methods. Consequently, we only mention the model-based methods for the sake of completeness. Among the model-free methods, a second distinction regards the object to be learned. Some methods directly learn a policy as their object of interest i.e the policy gradient based methods. Some other methods get their policy as a by-product of the learning of some kind of value function. These methods are said to be value-based. Finally, some algorithms called actor-critic based will learn both a value-function and a policy.

The new general scheme In order to learn from observations only, another step has to be added to the scheme introduced in Figure 2.2: the generation of samples from the environment. This is depicted in Figure 2.3. The two algorithms used in this work - which are about to be introduced - are instances of this scheme. It is to be noted that these steps may happen concurrently.

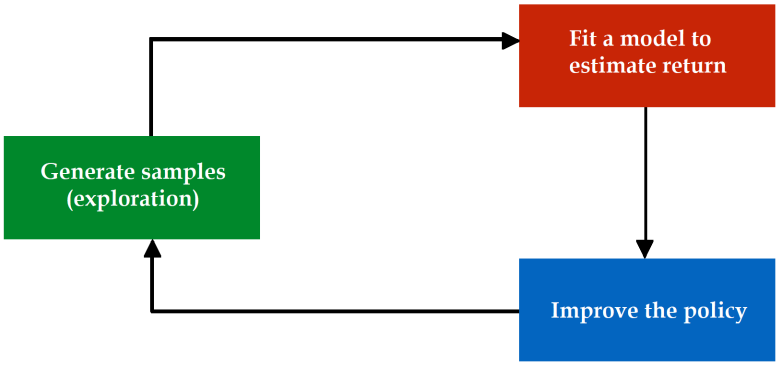

Figure 2.3: Redraw of a scheme from [Levine, 2021a]. Three steps scheme for model-free reinforcement learning. A step generates samples from the environment. These samples are used to improve a model estimating returns. This model is somehow used to improve a policy. This policy may be implicit or explicit.

2.5.1 Off-policy methods and on-policy methods

Among the methods which learn a solution to problem 2.7 from interactions with the environment, two different assumptions regarding the origin of the experience can be made. A distinction is made between methods which assume the experiences used to improve the policy come from an agent using the current policy from those which do not assuming anything. The former are called on-policy methods and require *fresh* experience every time a change is made to the policy. The latter are named off-policy and have no restriction on the policy that generated the experiences. In this case, this experience generating policy is usually coined as the *behaviour policy*.

2.5.2 One method for discrete-actions environments

In this section we will go through the conceptual steps which have to be carried on to get from dynamic programming as introduced in Section 2.4 to DDQN - one of the two algorithms we study in this work. Dynamic programming as introduced earlier relied on two assumptions. The first one is the finiteness of the state-action space and the possibility to store arrays which scales with it on a computer. The second is the access to the dynamics of the environment. In Section 2.5.2.1, the first assumption is lifted and the use of approximators is introduced. Subsequently, in Section 2.5.2.2, the second assumption is removed as well by the use of samples drawn from the environment. Then, in Section 2.5.2.3, the first practical algorithm combining previous ideas is presented. Finally, in Section 2.5.2.5, one of the two algorithms studied in this work is provided - Double Deep-Q Networks.

2.5.2.1 From value-iteration to fitted-value iteration

Here we assume that the environment dynamics are still available. In equation 2.24, every state and action must be looped through and at least one number must be stored for every location. In order to avoid this, fitted-value iteration relies on the generalization ability of an approximator such as a neural network. One way to do so it to define a loss to be minimized. The update equation 2.23, takes the current $Q^{\pi}_{k+1}(s,\ a)$ and updates the value function in a greedy fashion. In the case of fitted-value iteration, an estimate of the value function $V^{\pi}_{\theta}(s)$ is trained against an estimate of $\arg\max_a Q^{\pi}(s,\ a)$. A batch of states is selected. Using these, the update equation 2.25 for the parameters of the state value function can be computed. More details can be found in [Levine, 2021b].

$$\theta_{k+1} = \arg\min_{\theta} \frac{1}{2} \sum_{i} \left\| V_{\theta}(s_i) \ - \ \max_{a_i} \left(\mathop{\mathbb{E}}_{(s'_i,\, r_i) \sim p(.\,,\,.\,|\, s_i,\, a_i)} [\, r + \gamma\, V_{\theta_k}(s'_i)\,] \right) \right\|^2 \tag{2.25}$$

Two issues arise from this update equation. Firstly, it is not obvious how the states s_i are selected. Secondly, even if the model is known, computing the expectation could be intractable or prohibitively expensive. Both these issues will be tackled in the next section.

2.5.2.2 From fitted-value iteration to fitted Q-iteration

In this section, the second assumption regarding the availability of the dynamics model is removed as well. We are only left with one solution which is to interact with the environment. One might be tempted to work directly with fitted-value iteration. However, there is one big issue with the target in equation 2.25 that prevents its ease of use. Indeed, the maximum enforces to get values for every actions, which requires to be able to reset the environment in the same state and observe outcomes for every action. The inside, $\mathbb{E}\left[r + \gamma\, V_{\theta_k}(s'_i)\right]$ can be estimated using samples but getting these is the hard part. A solution is to work with a state-action value function which conditions the action. Given a batch of state-action pairs $(s_i,\, a_i)$, an equation similar to 2.25 for $Q_\theta(s,\, a)$ can be written as

$$\theta_{k+1} = \arg\min_{\theta} \frac{1}{2} \sum_i \left\| Q_\theta(s_i,\, a_i) \;-\; \mathop{\mathbb{E}}_{(s'_i,\, r_i) \sim p(.\,,\, .\,|\, s_i,\, a_i)} \left[r + \gamma \max_{a'_i} Q_{\theta_k}(s'_i, a'_i) \right] \right\|^2 \tag{2.26}$$

This time, the expectation is much easier to estimate from samples. These will consist of tuples of (state, action, next state, reward) which will be denoted by $(s,\, a,\, s',\, r)$. It appears that equation 2.26 minimizes a loss which is called the *Bellman error*. It can be obtained by substracting the right-hand side from Bellman optimality criterion for Q-functions as provided in 2.14. In the case of a Bellman error equal to 0 - and a perfect estimation of the sampled quantities - the policy derived from $Q_{\theta_k}(s,\, a)$ could be said to be optimal. In any other case not much can be said in general about the performance of the policy without empirically testing it. More details and pseudo-code for fitted-Q iteration can be found in [Ernst et al., 2005] which implements the supervised regression by using random forests. To the best of our knowledge, the usage of neural networks as approximators for fitted-Q learning was introduced in [Riedmiller, 2005]. The question of how to get the samples $(s,\, a,\, s',\, r)$ still arises. Fitted-Q iteration does not make any assumption regarding the origin of the samples. Consequently it is an off-policy algorithm. The authors of [Ernst et al., 2005] propose two behaviour policies. The first one is a *random policy* - sampling action randomly - which works on some environments but might not work on more complex tasks. The second suggested way to get the samples is to use the current version of the policy derived from $Q_{\theta_k}(s,\, a)$. This requires to deal with the exploration-exploitation dilemma discussed in Section 2.6.

2.5.2.3 From fitted Q-iteration to Q-learning and online fitted Q-iteration

Q-learning Q-learning is an algorithm introduced in [Watkins and Dayan, 1992]. As Fitted Q-iteration it is an off-policy algorithm which requires to samples $(s,\, a,\, s',\, r)$ tuples from the environment. In order to introduce this algorithm we need to take a step back and assume again to be in a tabular scenario. The move from fitted value-iteration to fitted Q-iteration allowing to take samples from the environment can also be done in the case of the value-iteration algorithm introduced in Section 2.4.5. Indeed, instead of updating $V^\pi_k(s)$ as in equation 2.24, one can update $Q^\pi_k(s,\, a)$, which would be written as

$$Q^\pi_{k+1}(s,\, a) = \sum_{s',\, r} p(s',\, r\,|\, s,\, a) \left[r + \gamma \max_{a'} Q^\pi_k(s', a') \right] \tag{2.27}$$

The right-hand-side of this update equation can be estimated from samples of the environment. However the update has to be performed on the whole state-action space. This would require to be able to perform action a in state s enough to get a good estimation. Even though we are working in a tabular scenario we might want to avoid to evaluate $Q^\pi(s,\, a)$ on the whole state-action space but only on some locations. One idea is to evaluate only state-action pairs that have been explored letting a default value for unexplored locations. One strategy to estimate 2.27 could be to compute estimates for $p(s'\,|\,r)$ and $\mathbb{E}[r]$. This process is known as *estimating the structure of the underlying MDP* [Ernst, 2020]. This would require to take the dataset of $(s,\, a,\, s',\, r)$ tuples and count the transitions $(s, a) \to s'$. Furthermore, the observed rewards would have to be aggregated with $(s,\, a)$ as a key. While doable this process might be tedious since it could require a large amount of samples to compute proper estimations.

A second idea naturally comes from equation 2.26. Indeed, one way to look at this update equation is - given some samples $(s,\, a,\, s',\, r)$ - the objective is to reduce the gap between the current estimates

$Q_{\theta_k}(s_i,\, a_i)$ and their greedy estimates $r_i + \gamma \max_{a'_i} Q_{\theta_k}(s'_i,\, a'_i)$. In a tabular scenario one way to do this could simply to perform the following update for every $(s_i,\, a_i,\, s'_i,\, r_i)$

$$Q(s_i,\, a_i) \longleftarrow (1-\alpha)\, Q(s_i,\, a_i) + \alpha \left[r_i + \gamma \max_{a'_i} Q(s'_i,\, a'_i) \right] \tag{2.28}$$

The coefficient α is the *learning rate* and can vary as more updates are performed. We define $\delta(s_i\,, a_i) \triangleq r_i + \max_{a'_i} Q(s'_i,\, a'_i) - Q(s_i,\, a_i)$ and call this quantity the *temporal difference*. Thus the update equation simply rewrites as

$$\begin{aligned} Q(s_i,\, a_i) \longleftarrow\ & Q(s_i,\, a_i) + \alpha \left[r_i + \gamma \max_{a'_i} Q(s'_i,\, a'_i) - Q(s_i,\, a_i) \right] \\ =\ & Q(s_i,\, a_i) + \alpha\, \delta(s_i,\, a_i) \end{aligned} \tag{2.29}$$

This update equation is the core idea of *Q-learning* as introduced in [Watkins and Dayan, 1992]. Under some assumptions on the learning rate and the visiting of every state-action pairs, Q-learning is proven to be converging to the optimal policy.

Online fitted Q-iteration Combining ideas from fitted Q-iteration and Q-learning, an online version of fitted Q-learning can be derived. Equation 2.26 seeks to find the minimizer of the *Bellman error* given a set of sampled tuples. Instead one could take an individual tuple and perform a small update one sample after the other. This leads to an online update equation which is written as

$$\begin{aligned} \theta_{k+1} &= \theta_k + \alpha\, \nabla_\theta Q_\theta(s_i,\, a_i) \left[r_i + \gamma \max_{a'_i} Q_{\theta_k}(s'_i,\, a'_i) - Q_{\theta_k}(s_i,\, a_i) \right] \\ &= \theta_k + \alpha\, \nabla_\theta Q_\theta(s_i,\, a_i)\, \delta_{\theta_k}(s_i, a_i) \end{aligned} \tag{2.30}$$

This update is used in the *online fitted Q-iteration* algorithm which is also sometimes called *deep Q-learning*. Pseudo-code for online fitted Q-iteration is provided in Algorithm 1.

Algorithm 1: Online fitted Q-iteration

Initialize Q_{θ_0} with random weights
for episode $= 1, M$ **do**
  Initialize the environment and take $s_0 \sim P(s_0)$
  for $t = 0, T-1$ **do**
    Select and execute action a_t
    Observe s'_t and r_t
    Update θ_k according to:
    if s'_t is a terminal state **then**
      $\theta_{k+1} = \theta_k + \alpha\, \nabla_\theta Q_\theta(s_t, a_t) \left[r_t + \gamma \max_{a'_t} Q_{\theta_k}(s'_t,\, a'_t) - Q_{\theta_k}(s_t,\, a_t) \right]$
    else
      $\theta_{k+1} = \theta_k + \alpha\, \nabla_\theta Q_\theta(s_t, a_t) \left[r_t - Q_{\theta_k}(s_t,\, a_t) \right]$
    Update state: $s_{t+1} \leftarrow s'_t$
    Update iteration: $k \leftarrow k+1$
  end
end

An important aside has to be made regarding equation 2.30, the term $r_i + \max_{a'_i} Q_{\theta_k}(s'_i, a'_i)$ was taken as independent with respect to θ. Thus no gradient flows with respect to that part of the Bellman residual. This is one way to work. Another way could be to use a method called *residual gradients* that would consider this term to depend on θ thus this part would need to be differentiated as well. A review of residual methods can be found in [Zhang et al., 2019]. From all these observations, some people including the authors of [Zhang et al., 2019] describe 2.30 as a *semi-gradient* update equation since it is not the gradient of a proper objective. These considerations are outside the scope of this introduction to deep reinforcement learning and will be left behind.

2.5.2.4 DQN

DQN was introduced in the seminal paper [Mnih et al., 2013]. As Q-learning, DQN is an off-policy algorithm. It aims at solving issues which arose with online fitted-Q-iteration. This paper reintroduced already existing ideas for a new problem at the time. These ideas are the use of a delayed *target network* and a *replay buffer*.

Target networks While not stated explicitly this way in [Mnih et al., 2013] the authors made a design change on update equation 2.30. As mentioned earlier the term $r_i + \max_{a'_i} Q_{\theta_k}(s'_i, a'_i)$ was considered as fixed with respect to θ. However, the *gradient* for the update still depends on the quantity that is being changed. Another way to phrase this is to say that the target $r_i + \max_{a'_i} Q_{\theta_k}(s'_i, a'_i)$ depends on θ_k which implies the target is somehow moving as θ is updated. This is said to cause training instabilities. One way to solve this issue is to take a frozen version of the Q-function approximator $Q_{\theta'_k}$. This way the target does not change for every update. The network $Q_{\theta'_k}$ is called a *target network*. It may be updated every K iterations. However in this work we choose to use a method called *Polyak averaging* which is simply written as

$$\theta'_{k+1} \longleftarrow \tau\, \theta'_k + (1 - \tau)\, \theta_k \tag{2.31}$$

with $\theta'_0 = \theta_0$ and $\tau \in [0, 1]$ but usually *close* to 1.

Experience replay buffer Looking at Algorithm 1, three observations can be made. Firstly, sequential updates will mostly use sequential states. Another way to phrase this is to say there is a state correlation in the updates. The authors of [Mnih et al., 2013] claimed this way of ordering the updates to be inefficient and unsafe since it could lead to "unwanted feedback loops". Secondly, each sampled tuple (s_i, a_i, s'_i, r_i) is used once and discarded which might be sample inefficient. Thirdly, the gradient estimator uses only one sample. Drawing one sample at a time from the environment does not allow to compute a better estimator by lowering variance through averaging.

From these observations the authors of [Mnih et al., 2013] suggested the use of an *experience replay buffer* which stores the last N sampled experiences. Hence, samples are drawn uniformly from this buffer which breaks state-wise correlation and allows sample reuse. This also gives the possibility to compute averaged gradient estimators. The buffer will be written $\mathcal{D}$. Samples draw from the buffer will be denoted by $(s_i, a_i, s'_i, r_i) \sim \mathcal{D}$.

Combining target networks and experience replay When combined and applied on the online fitted Q-iteration scheme, target networks and experience replay lead to the DQN algorithm as described in [Mnih et al., 2013]. Pseudo-code of DQN is provided in Algorithm 2

Algorithm 2: Deep Q-learning with experience replay and polyak averaged target networks

Inputs:
B: the batch size, integer ≥ 1
N: the replay buffer size, integer ≥ 1
τ, the target network update rate, float $\in [0, 1]$
α, the learning rate, float > 0

Initialization:
Initialize Q_{θ_0} with random weights
Initialize $Q_{\theta'_0}$ with $\theta'_0 = \theta_0$
Initialize experience replay memory for size N

Iterations:
for episode $= 1, M$ **do**
  Initialize the environment and take $s_0 \sim P(s_0)$
  for $t = 0, T-1$ **do**
    Select and execute action a_t
    Observe s'_t and r_t
    Store (s_t, a_t, s'_t, r_t) into $\mathcal{D}$

    Sample uniformly a minibatch of transitions $(s_i,\, a_i,\, s'_i,\, r_i)$ of size B from $\mathcal{D}$

    Compute the targets.
    for $i = 1,\, B$ **do**
      $$y_i = \begin{cases} r_i & \text{if } s'_i \text{ is terminal} \\ r_i + \gamma \max_{a'_i} Q_{\theta'_k}(s'_i,\, a'_i) & \text{otherwise} \end{cases}$$
    end

    Update θ_k according to:
    $\theta_{k+1} = \theta_k + \alpha \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta Q_\theta(s_t, a_i) \left[y_i - Q_{\theta_k}(s_i,\, a_i) \right]$

    Update the target network:
    $\theta'_{k+1} = \tau\, \theta'_k + (1 - \tau)\, \theta_k$

    Update state: $s_{t+1} \leftarrow s'_t$
    Update iteration: $k \leftarrow k + 1$
  end
end

2.5.2.5 From DQN to DDQN

In order to obtain one of the two algorithms studied in this work, a last step must be taken. A well known issue with Q-learning is called the *overestimation bias*. There is a systematic positive bias for $Q_{\theta_k}(s,\, a)$ with respect to the *true* Q-function of the greedy policy induced by $Q_{\theta_k}(s,\, a)$. In other words if $\pi_{\theta_k}(s) = \arg\max_a Q_{\theta_k}(s,\, a)$ then $Q_{\theta_k}(s,\, a)$ tends to be larger than $Q^{\pi_{\theta_k}}(s,\, a)$. In the tabular scenario, a review of the issue and a workaround can be found in [Hasselt, 2010]. In a few words, the problem is said to be mostly explained by a double use of Q_{θ_k} for the estimation of $\max_{a'_i} Q_{\theta_k}(s'_i,\, a'_i)$. Indeed, one observes that $\max_{a'_i} Q_{\theta_k}(s'_i,\, a'_i) = Q_{\theta_k}(s'_i,\, \arg\max_{a'_i} Q_{\theta_k}(s'_i, a'_i))$. Thus, this estimator is refereed as a *single estimator* by [Hasselt, 2010]. The author showed that - in the tabular scenario - using an independent estimate $Q'_{\theta_k}(s,\, a)$ to perform the evaluation could mitigate the issue. The estimator is written as $Q'_{\theta_k}(s,\, \arg\max_{a'_i} Q_{\theta_k}(s'_i, a'_i))$. Since it uses two independent estimators, this scheme is called *double estimator*.

The observation on the maximization has been generalized in [van Hasselt et al., 2015]. The authors showed that any error in the estimation of $Q^\pi(s,\, a)$ by $Q_k(s,\, a)$ could lead to an overestimation bias. Since this work uses neural network Q_θ to approximate Q^π it is natural that the estimation will contain

errors. The authors showed empirically that systematic overestimation biases was observed with DQN and proposed *Double DQN*. This implies to use two networks Q_{θ_A} and Q_{θ_B} whose estimation should be as independent as possible. The authors suggested to use $\theta_A = \theta$ and $\theta_B = \theta'$ where θ' is the target network introduced in DQN. Even though θ and θ' are far from being independent, the authors showed a good reduction in the overestimation bias and a substantial increase in the performance of the policy in comparative benchmarks.
In conclusion, DDQN is only a very slight modification on top of DQN. The only change is to be made on the evaluation of the targets y_i as noted in Algorithm 2. The change is written bellow,

$$r_i + \gamma \max_{a'_i} Q_{\theta'_k}(s'_i,\, a'_i) \Longrightarrow r_i + \gamma\, Q_{\theta'_k}\left(s'_i,\, \arg\max_{a'_i} Q_{\theta_k}(s'_i, a'_i)\right) \tag{2.32}$$

2.5.3 One method for continuous-action environments and discrete-action environments: Soft-Actor Critic

One typical issue with DDQN and DQN is related to the way actions are handled. Usually $Q_\theta(s,\, a)$ is a neural networks which takes as input the state s and outputs a vector $\mathbf{a}_\theta(s)$ for every action. This leads to two observations. Firstly, actions are regarded as independent where $\mathbf{a}_\theta(s)$ can be seen as a kind of classifier. However, in many tasks the action space has some "structure". For example in the case of a robot actuators, commands are continuous values that can be compared. If they were to be discretized and considered as separate classes as in the classical DDQN implementation, this ordering would be lost. A second issue is related to the scalability of $\arg\max_{a'} Q_\theta(s',\, a')$. This requires to perform a complete look-up of $\mathbf{a}_\theta(s)$. In the case of large action spaces the cost of this operation might be prohibitive. For these reasons other methods were introduced. The algorithm presented in this section is one of them. It solves the poor scalability with respect to the size of the action space in some cases. It builds upon a slightly different framework called soft-MDP which will be briefly introduced in Section 2.5.3. Then the general algorithm called *soft actor-critic* (SAC) will be introduced in Section 2.5.3. Subsequently a simplification of the general SAC algorithm for discrete action spaces whose size is not prohibitive will be presented in Section 2.5.3

Soft-MDP and a new objective

In Section 2.1, the standard MDP was introduced. It appears that in subsequent section - for most considerations - the rewards were only used through $\mathbb{E}_{P(r|s,\,a)}\,[r] \triangleq R(s,\, a)$. The soft-MDP is a modification over the standard MDP that has interest only for stochastic policies $\pi(\,.\,|\,s)$. It simply modifies the expected reward signal as

$$R(s,\, a) \Longrightarrow R(s,\, a) + \lambda\, H\left(\pi(\,.\,|\,s)\right)$$

where $H\left(\pi(\,.\,|\,s)\right)$ is the entropy of the policy distribution. This leads to a modification of the reinforcement learning problem which is written

$$\pi^*_{\text{RegEnt}} \;\in\; \arg\max_{\pi' \in \Pi} \sum_{t=0}^{\infty} \gamma^t \mathop{\mathbb{E}}_{\rho^\pi(s_t,\, a_t)} \left[\, R(s_t,\, a_t) + \lambda\, H(\pi(\,.\,|\,s_t))\,\right] \tag{2.33}$$

where $\rho^\pi(s_t,\, a_t)$ is the state-action marginal of the trajectory distribution induced by policy π. This new objective enforces an optimal policy to make transitions that will lead to large rewards and large policy entropy now and in the future. It requires to maximize the rewards along with the policy entropy on the whole trajectory [Haarnoja et al., 2017].

Some definitions

Soft value functions Here we define the soft Q-function $Q^\pi_{\text{soft}}(s,\, a)$ as well as the soft value-function $V^\pi_{\text{soft}}(s)$. In order to keep the notations short and stay consistent with [Haarnoja et al., 2017], we will use the notation $\tau \sim \pi$ to denote the trajectory $\tau = (s_0,\, a_0,\, s_1\,, a_1,\, ...)$ follows $\pi(\,.\,|s)$ and the transition dynamics. We will add $s_0 = s$, $a_0 = a$ to denote the trajectory starts from $(s,\, a)$.

The soft value-function is defined as

$$V^{\pi}_{\text{soft}}(s) \triangleq \mathop{\mathbb{E}}_{\substack{\tau \sim \pi \\ s_0 = s}} \left[\sum_{t=0}^{\infty} \gamma^t \left[R(s_t,\, a_t) + \lambda\, H(\pi(\,.\,|\,s_t)) \right] \right] \tag{2.34}$$

The soft Q-function is defined

$$Q^{\pi}_{\text{soft}}(s,\, a) \triangleq R(s,\, a) + \mathop{\mathbb{E}}_{\substack{\tau \sim \pi \\ s_0 = s,\, a_0 = a}} \left[\sum_{t=1}^{\infty} \gamma^t \left[R(s_t,\, a_t) + \lambda\, H(\pi(\,.\,|\,s_t)) \right] \right] \tag{2.35}$$

where the first entropy term is not accounted for since the first action is not drawn from $\pi(\,.\,|\,s)$.

A new max operator: $\widetilde{\max}_\lambda$ In the subsequent section, we will make use of a new operator $\widetilde{\max}_\lambda$. It operates as a kind of softmax. If its parameter $\lambda \to 0$, then $\widetilde{\max}_\lambda$ behaves like a regular max. Here we give the definition in the continuous case as provided in [Haarnoja et al., 2017]. For the discrete case we follow [Poupart, 2020].

Continuous $\widetilde{\max}_\lambda$

$$\widetilde{\max_{\mathbf{x}}}_{\lambda}\, f(\mathbf{x}) \triangleq \lambda \log \int \exp\left(\frac{f(\mathbf{x})}{\lambda} \right) \mathrm{d}\mathbf{x} \tag{2.36}$$

Discrete $\widetilde{\max}_\lambda$

$$\widetilde{\max_{a}}_{\lambda}\, f(a) \triangleq \lambda \log \sum_{a} \exp\left(\frac{f(a)}{\lambda} \right) \tag{2.37}$$

Soft Bellman optimality An equivalent of the Q-function Bellman optimality 2.14 can be written for the soft Q-function. This is called the *soft Bellman equation*.

$$Q^{*}_{\text{soft}}(s,\, a) = R(s,\, a) + \mathop{\mathbb{E}}_{s' \sim p(\,.\,|\,s,\,a)} \left[\gamma\ \widetilde{\max_{a'}}_{\lambda}\ Q^{*}_{\text{soft}}(s', a') \right] \tag{2.38}$$

Soft greedy policy In the case of deterministic policies in the standard MDP, the greedy policy $\pi_{\text{greedy}}(s) : \mathcal{S} \to \mathcal{A}$ was equal to $\arg\max_a Q(s,\, a)$. An equivalent $\pi_{\text{greedy}}(\,.\,|, s) : \mathcal{S} \times \mathcal{A} \to [0, 1]$ for soft MDP and stochastic policies can be defined.

Continuous greedy policy

$$\pi_{\text{greedy}}(\,.\,|\,s) \;=\; \frac{\exp\left(Q_{\text{soft}}(s,\,.\,)/\lambda\right)}{\int \exp\left(Q_{\text{soft}}(s,\, a)/\lambda\right) \mathrm{d}\mathbf{a}} \tag{2.39}$$

Discrete greedy policy

$$\pi_{\text{greedy}}(\,.\,|\,s) \;=\; \frac{\exp\left(Q_{\text{soft}}(s,\,.\,)/\lambda\right)}{\sum_a \exp\left(Q_{\text{soft}}(s,\, a)/\lambda\right)} \tag{2.40}$$

The proof that - in the discrete action case - the equation 2.40 is indeed *the* greedy policy with respect to Q_{soft} is available in [Poupart, 2020].

Policy improvement theorem In an ealier section, we discussed the *policy improvement theorem*. It was in the context of perfectly know standard MDPs and deterministic policies. There exists an equivalent for soft MDPs and stochastic policies. It shows that the greedy policy defined in 2.39 and 2.40 lead to a policy that has a soft Q-function larger than (or equal to) the policy induced by the previous soft Q-function. Formally the soft policy improvement theorem as provided in [Poupart, 2020] is

> **Theorem 2.5.1** *Let $Q^{\pi_i}_{soft}(s,\, a)$ be the soft Q-function of π_i. Let $\pi_{i+1}(\,.\,|\,s) = \pi_{greedy}(\,.\,|\,s)$ the soft greedy policy based on $Q^{\pi_i}_{soft}(s,\, a)$. Then $Q^{\pi_{i+1}}_{soft}(s,\, a) \geq Q^{\pi_i}_{soft}(s,\, a)\ \ \forall s,\, a \in \mathcal{S}, \mathcal{A}$.*

Soft policy iteration

In this section we briefly introduce the soft policy iteration algorithm as presented in [Haarnoja et al., 2018]. This is necessary to understand the soft actor-critic that will be introduced in the coming sections. In the same way the *policy improvement theorem* was key for the policy iteration algorithm, Theorem 2.5.1 leads to an equivalent *soft policy iteration* algorithm. Following policy iteration, the soft variant iterates between *soft policy evaluation* and *soft policy improvement* which we now introduce.

Soft policy evaluation In order to make things more obvious we explicitly draw the parallel between soft policy evaluation and standard policy evaluation. As when it was first introduced we assume a tabular scenario and perfectly known dynamics. An equivalent of the operator $\mathcal{T}_q^\pi$ for performing policy evaluation can be derived for the soft MDP. According to [Haarnoja et al., 2018] it is defined as

$$\mathcal{T}^\pi_{q,\,\text{soft}}\, Q(s,a) = R(s,\,a) + \gamma \mathop{\mathbb{E}}_{s' \sim p(s'|\, s,\, a)} \left[\mathop{\mathbb{E}}_{a' \sim \pi(\,.\,|\,s)} \left[Q(s',a') - \log\left(\pi(a'\,|\,s')\right) \right] \right] \tag{2.41}$$

Thus, soft policy evaluation is the process of iteratively applying $Q^\pi_{k+1} = \mathcal{T}^\pi_{q,\,\text{soft}} Q^\pi_k$. It is shown - under some conditions - to converge to the true soft Q-function as $k \to \infty$.

Soft policy improvement In Theorem 2.5.1, a policy defined greedily from a soft Q-function was shown to have equal or better performance. However, the authors of [Haarnoja et al., 2018] claim this process to be intractable in general. Consequently, they restrict the policies $\pi(\,.\,|\,s)$ to some set of policies Π. Thus a specific version of Theorem 2.5.1 is proposed. We call it the *tractable policy improvement theorem*. Since the class of policies is restricted, the update toward the greedy policy is constrained. The authors suggest the use of the KL divergence projection. Thus the update they propose is

$$\pi_{k+1}(\,.\,|\,s) = \mathop{\arg\min}_{\pi' \,\in\, \Pi} \mathrm{D}_{\mathrm{KL}} \left(\pi'(\,.\,|s) \,\middle\|\, \pi_{k,\,\text{greedy}}(\,.\,|\,s) \right) \tag{2.42}$$

They show that - under some conditions - the updated policy has equal or better performance compared to the previous policy.

General Soft actor-critic

Soft actor critic or SAC builds upon the soft policy iteration algorithm introduced previously. It tackles three issues related to that algorithm. Firstly, soft policy iteration can only be performed exactly for the tabular scenario. Secondly, the convergence of soft policy evaluation is only in the limit. Thirdly, soft policy iteration requires to know the dynamics. For these reasons SAC suggests to run one step of policy evaluation and one step of policy improvement. Furthermore, the policy $\pi(\,.\,|\,s)$ and soft Q-function $Q^\pi_{\text{soft}}(s,\,a)$ will be parametrized by two neural networks, $\pi_\theta(\,.\,|\,s)$ and $Q_\phi(s,\,a)$ respectively. Lastly, SAC as DQN/DDQN will work by sampling from the environment, storing in a replay buffer and minimize some kind of loss. In this section we will briefly give the required details to understand the algorithm as well as its pseudo-code.

The losses to minimize Exactly as in DQN / DDQN, soft actor critic will sample tuples (s_t, a_t, s'_t, r_t) from the environment. It will store and then sample them from the experience replay buffer. These samples will be used to perform gradient step on two losses since there are two networks to be updated. The loss for the soft Q-function network will be denoted $J_Q(\theta)$ and the loss for the policy network will be written as $J_\pi(\phi)$. As DQN/DDQN, soft actor critic will use target networks for the target computations. These will be updated using polyak averaging.

The soft Q-network loss $J_Q(\theta)$ The soft Q-network will be trained in order to minimize the *soft Bellman residual* according to samples drawn from the replay buffer $\mathcal{D}$. In order to make the equations more readable we first introduce the target function $\hat{Q}_\theta(s,\,a)$ it is simply one application of the soft Bellman backup operator. It is defined as

$$\hat{Q}_\theta(s,\, a) \triangleq R(s,\, a) + \gamma \mathop{\mathbb{E}}_{s' \sim p(s'|\, s,\, a)} \left[\mathop{\mathbb{E}}_{a' \sim \pi_\phi(\,.\,|\, s)} \left[Q_\theta(s', a') - \log\left(\pi_\phi(a' \,|\, s')\right) \right] \right] \tag{2.43}$$

As for DQN/DDQN, soft actor critic uses a target network for the target computation. The target Q-network is denoted $Q_{\bar{\theta}}(s,\, a)$.

$$J_Q(\theta) = \mathop{\mathbb{E}}_{(s_t,\, a_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\theta(s_t,\, a_t) - \hat{Q}_{\bar{\theta}}(s_t,\, a_t) \right)^2 \right] \tag{2.44}$$

This loss can be estimated from samples (s_t, a_t, s_{t+1}, r_t) . These will be drawn from the replay buffer. Then the action a_{t+1} (written a' above) is simply drawn from the policy network. In practice only one sample a_{t+1} is drawn from $\pi_\phi(\,.\,|\, s_{t+1})$

The policy network loss $J_\pi(\phi)$ For the policy loss, we first look at the KL divergence written in equation 2.42. The constant denominator of π_{greedy} can be discarded since it does not impact the gradient with respect to ϕ. Then - up to a constant multiplier λ - the KL divergence has the same gradient as $\mathbb{E}_{a \sim \pi_\phi} \left[\lambda \log\left(\pi_\phi(a \,|\, s)\right) - Q_\theta(s,\, a) \right]$.

Then, we want the KL divergence to be minimized for all the state space. One way to approximate this is to draw states from the replay buffer and use the KL divergence estimator in expectation. Thus the policy loss is

$$J_\pi(\phi) = \mathop{\mathbb{E}}_{s_t \sim \mathcal{D}} \left[\mathop{\mathbb{E}}_{a \sim \pi_\phi(\,.\,|\, s_t)} \left[\lambda \log\left(\pi_\phi(a_t \,|\, s_t)\right) - Q_\theta(s_t,\, a_t) \right] \right] \tag{2.45}$$

It turns out the inside expectation is based on π_ϕ which depends on ϕ. Thus, computing the gradient with respect to ϕ is not straightforward. The author of [Zhou et al., 2018] suggest to use the reparametrization trick. In order to sample from $\pi_\phi(\,.\,|s)$ the transformation $a = f_\phi(\epsilon; s)$ is used. The noise vector ϵ is usually sampled from a Gaussian. Thus the final form of the policy loss - which can be estimated from samples - is

$$J_\pi(\phi) = \mathop{\mathbb{E}}_{\substack{s_t \sim \mathcal{D} \\ \epsilon_t \sim \mathcal{N}}} \left[\lambda \log\left(\pi_\phi(f_\phi(\epsilon_t\,;\, s_t) \,|\, s_t)\right) - Q_\theta(s_t,\, f_\phi(\epsilon_t\,;\, s_t)) \right] \tag{2.46}$$

Automatic temperature $\boldsymbol{\lambda}$ adjustment In the second version of SAC which was provided in [Zhou et al., 2018], the authors suggested the parameter λ which scales the entropy regularization could be adjusted on the go. Indeed, a single value will not be the best fit for tasks whose reward scale differs. It may even not be optimal to use a constant temperature across the same task learning process. Thus, they propose an update scheme for the parameter λ. The derivation is more involved and available in their paper. Here we give the loss $J(\lambda)$ in a style similar to $J_Q(\theta)$ and $J_\pi(\phi)$.

$$J(\lambda) = \mathop{\mathbb{E}}_{s_t \sim \mathcal{D}} \left[\mathop{\mathbb{E}}_{a_t \sim \pi_\phi(\,.\,|\, s_t)} \left[-\lambda \log\left(\pi_\phi(a_t \,|\, s_t)\right) - \lambda \bar{\mathcal{H}} \right] \right] \tag{2.47}$$

This objective is to be minimized and is straightforward to estimate from samples. It requires the selection of the constant $\bar{\mathcal{H}}$ which is called the *entropy target*. For continuous actions the authors used Zhou et al. [2018] used $\bar{\mathcal{H}} = -\dim(\mathcal{A})$ where $\dim(\mathcal{A})$ is the number of dimensions of the action space.

Double $Q(s, a)$ estimator In Section 2.5.2.5, we introduced DDQN. This algorithm tackled the so-called *overestimation bias* encountered by DQN. It appears that a similar issue arises with soft actor critic. This problem turns out to be general for value-based actor-critic algorithms as suggested in [Fujimoto et al., 2018]. The solution they advocate uses two Q-networks trained concurrently using the same samples but different initialization. The new estimator for $Q(s, a)$ that is supposed to be less prone to overestimation bias is defined as $Q_{\theta_1, \theta 2}(s, a) \triangleq \min_{i=1,2} Q_{\theta_i}(s, a)$. The implications for soft actor critic are changes in the the Q-network loss $J_Q(\theta)$ and in the policy loss $J_\pi(\phi)$. In the new versions the estimator $Q_{\theta_1, \theta 2}(s, a)$ is used in place of $Q_\theta(s, a)$.

Full pseudo-code Here we wrap-up everything mentioned previously. The soft actor critic follows a pattern similar to DQN/DDQN. Firstly, there is a experience collection process where the policy is sampled and tuples are store in the replay buffer. Then the losses are approximated by drawing a batch of samples from the buffer. Finally, networks are updated using polyak averaging. The loss gradient is computed using backpropagation. Optimization is carried on using any optimizer such as SGD, Adam, RMSProp. The full pseudo-code for the soft actor critic algorithm with reparametrization trick is provided in Algorithm 3.

Enforcing action bounds On many continuous environments, the actions are bounded. In the most common benchmarks as well as the environments used in this work, the action bounds are of the kind $a \in [-b\,; b]$ with b usually equal to one. A simple way to obtain such bounds for the actor network is to use $b\tanh(x)$ as output function. This squashing function involves some calculations to get the final density function. The details are provided in Appendix A.1. The final expression for the log of the actor distribution is $\log(\pi_\phi(a \,|\, s)) = \log(\mu(u \,|\, s)) - \sum_{i=1}^{d} \log(1 - \tanh^2(u_i))$.

Algorithm for discrete actions

In order to use Algorithm 3 for discrete actions, one could simply parametrize a discrete distribution which allows to use the reparametrization trick. One way to do so is to use the *Gumbel-Softmax* distribution [Jang et al., 2017]. This approach might work but it is not optimal. The author of [Christodoulou, 2019] suggests to forget about the reparametrization trick and simply output a discrete distribution using a softmax layer at the end of the network. Indeed, the reparametrization trick was necessary to draw differentiable samples. These differentiable samples were used to approximate the expectations $\mathbb{E}_{a \sim \pi_\phi}[\,.\,]$. This is not necessary with a discrete distribution whose expectation may be computed exactly. The output of the Q-network is also changed to a vector of Q-values, one for every action. In this case, the distribution over actions simply becomes a vector of probabilities. Thus, expectations can be computed as a dot product. In the following equations, we provide the changes to be made to the loss equations. The full pseudo-code for the discrete version of SAC is provided in Algorithm 5 in Appendix A.3.

The target for $J_Q(\theta)$ becomes

$$\hat{Q}_\theta(s, a) \triangleq R(s, a) + \gamma \mathop{\mathbb{E}}_{s' \sim p(\,.\,|\,s, a)} \left[\pi_\phi(s')^T \left[Q_\theta(s') - \log\left(\pi_\phi(s')\right)\right]\right] \tag{2.48}$$

The policy loss $J_\pi(\phi)$ is now

$$J_\pi(\phi) = \mathop{\mathbb{E}}_{s_t \sim \mathcal{D}} \left[\pi_\phi(s_t)^T \left[\lambda \log\left(\pi_\phi(\,s_t)\right) - Q_\theta(s_t)\right]\right] \tag{2.49}$$

Finally, the temperature loss $J(\lambda)$ is written

$$J(\lambda) = \mathop{\mathbb{E}}_{s_t \sim \mathcal{D}} \left[\pi_\phi(s_t)^T \left[-\lambda \log\left(\pi_\phi(s_t)\right) - \lambda\,\bar{\mathcal{H}}\right]\right] \tag{2.50}$$

Algorithm 3: General soft actor critic with reparametrization trick

Inputs:
B: the batch size, integer ≥ 1
N: the replay buffer size, integer ≥ 1
τ, the target networks update rate, float $\in [0, 1]$
α_Q, α_ϕ, α_λ, the learning rate, float > 0
$\overline{\mathcal{H}}$, the entropy target, float < 0

Initialization:
Initialize $Q_{\theta_{1,0}}$ and $Q_{\theta_{2,0}}$ with random weights
Initialize $Q_{\bar{\theta}_{1,0}}$ and $Q_{\bar{\theta}_{2,0}}$ with $\bar{\theta}_{1,0} = \theta_{1,0}$ and $\bar{\theta}_{2,0} = \theta_{2,0}$
Initialize experience replay memory for size N

Iterations:
for episode $= 1, M$ **do**
  Initialize the environment and take $s_0 \sim P(s_0)$
  for $t = 0, T-1$ **do**
    Draw the action from the policy network and act
    $a_t = f_{\phi_k}(\epsilon\,;\, s_t)$ with $\epsilon \sim \mathcal{N}$
    Observe s'_t and r_t
    Store (s_t, a_t, s'_t, r_t) into $\mathcal{D}$

    Sample uniformly a minibatch of transitions $(s_i,\, a_i,\, s'_i,\, r_i)$ of size B from $\mathcal{D}$

    Draw the noises and compute the targets
    for *i = 1, B* **do**
      $\epsilon_i \sim \mathcal{N}$
      $\epsilon'_i \sim \mathcal{N}$
      $$y_i = \begin{cases} r_i & \text{if } s'_i \text{ is terminal} \\ r_i + \gamma \left(\min_{j=1,2} \left[Q_{\bar{\theta}_{j,k}}(s'_i,\, f_{\phi_k}(\epsilon'_i\,;\, s'_i)) \right] - \log\left(\pi_{\phi_k}\left(f_{\phi_k}(\epsilon'_i\,;\, s'_i) \,|\, s'_i \right) \right) \right) & \text{otherwise} \end{cases}$$
    end

    Update $\theta_{1,k}$ and $\theta_{2,k}$ according to:
    $\theta_{i,k+1} = \theta_{i,k} - \alpha_Q \, \nabla_{\theta_{i,k}} \left[\frac{1}{2B} \sum_{j=1}^{B} \left(Q_{\theta_{i,k}}(s_j,\, a_j) - y_j \right)^2 \right]$ for i $\in \{1, 2\}$

    Update ϕ according to:
    $\phi_{k+1} = \phi_k - \alpha_\pi \, \nabla_{\phi_k} \left[\frac{1}{B} \sum_{j=1}^{B} -\lambda \, \log\left(\pi_{\phi_k}(f_{\phi_k}(\epsilon_j\,;\, s_j) \,|\, s_j) \right) - \min_{i=1,2} Q_{\theta_{i,k}}(s_j,\, f_{\phi_k}(\epsilon_j\,;\, s_j)) \right]$
    Update λ according to:
    $\lambda_{k+1} = \lambda_k - \alpha_k \, \nabla_{\lambda_k} \left[\frac{1}{B} \sum_{j=1}^{B} -\lambda_k \log\left(\pi_{\phi_k}(f_{\phi_k}(\epsilon_j\,;\, s_j) \,|\, s_j) - \lambda_k \, \overline{\mathcal{H}} \right] \right.$

    Update the target networks:
    $\theta'_{i,k+1} = \tau \, \theta'_{i,k} + (1 - \tau) \, \theta_{i,k}$ for i $\in \{1, 2\}$

    Update state: $s_{t+1} \leftarrow s'_t$
    Update iteration: $k \leftarrow k + 1$
  end
end

2.6 Exploration and exploitation

Exploration is a problem that arises as soon as one hast to learn from samples. Reinforcement learning algorithms need samples from a broad set of trajectories. Most often, an algorithm will be unable to learn an optimal policy for an area of the state space it has never seen before. Consequently, many strategies may be applied to explore region of the state-action space which matter in order to learn a good policy. It may sometimes depend on whether the algorithm is off-policy or on-policy.

A strategy which works well for both cases is to use the current policy to carry exploration and collect samples. However one important trade-off arises. It is called the exploration-exploitation trade-off [Sutton and Barto, 2018, p. 3]. A reinforcement learning agent is supposed to act depending on the knowledge it has from the environment (the exploitation). At some point in training, the agent will map states to action actions according to samples it has already seen. However, in order to find a better policy, the agent might need samples from trajectories it will not cover since it is not designed to act sub-optimally (the exploration). One way to solve this issue is to add random noise on actions such that part of the decision is independent from the agent. However, neither following the agent nor exploring randomly are the optimal choice. In other words, a trade-off between exploitation and exploration has to be found.

Both DDQN and SAC are off-policy algorithm which means they make no assumption on the origin of the samples used for learning. Hence, adding noise on top of the action given by the current policy is not an issue. Now we will briefly discuss what are the most usual exploration strategies for DDQN and SAC. These are also the ones used in this work.

Exploration with DDQN

Epsilon-greedy exploration Epsilon-greedy is perhaps the simplest form of random noise. It works for any discrete action policy. Epsilon-greedy policy samples a random action with probability ϵ or act according to the policy with probability ϵ (greedy action). Thus, from the policy $\pi(\,.\,|\,s)$ and with $|\mathcal{A}|$ the number of actions, the new policy is $\pi^\epsilon(a\,|\,s) = \frac{\epsilon}{\mathcal{A}} + (1-\epsilon)\,\pi(a\,|\,s)$. The policy in the case of DDQN is deterministic and selects the action with largest Q-value such that

$$\pi^\epsilon(a\,|\,s) = \begin{cases} \frac{\epsilon}{|\mathcal{A}|} + 1 - \epsilon & \text{if a} = \arg\max_{a'} Q_\theta(s,\, a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{2.51}$$

It appears that it is often desirable to decrease the noise as the agent improves. Several options exist to decrease ϵ, one of them is provided in A.2

Exploration with SAC

Soft-actor critic takes the problem of exploration the other way around. Instead of adding noise independent to the agent decision, SAC changes the objective to enforce exploration. Thus soft-actor critic is said to naturally incorporate exploration especially with an auto-tune entropy temperature [Zhou et al., 2018]. Furthermore, since SAC is off-policy one may also add any form of exploration on top of it.

2.7 N-step returns

In this section we introduce a slight modification on top of DDQN and SAC. This modification is transparent in the sense it does not change the inner workings of these algorithms. In every earlier section of this chapter we made use of a concept introduced in equation 2.5 which is called *bootstrapping*. Schematically, the idea is to use an already available estimator and rewards to improve the original estimator. Practically it was used in DDQN as well as in SAC. In the case of DDQN, the *target* was computed as $y = r + \gamma\, \max_{a'} Q_{\theta'}(s',\, a')$ where $(s,\, a,\, s',\, r)$ was sampled from the replay buffer. When we roll back across the preceding sections we observe this target definition originates from value-evaluation as well as value-iteration introduced in Section 2.4.2 and 2.4.5. The updates used in these algorithm were themselves based on the bootstrapping equation 2.5. However, one could argue that cutting the rewards

and using the estimator after only one step was an arbitrary choice. In some sense, it was. Indeed, taking back the original equation, one could perfectly write

$$\begin{aligned} V^{\pi}(s) &= \mathop{\mathbb{E}}_{\substack{a_t \sim \pi(\,.\,|\,s_t) \\ s_{t+1} \sim p(\,.\,|s_t, a_t)}} \left[R(s_0, a_0) + \sum_{t=1}^{T} \gamma^t \, r(s_t, a_t) \Big| s_0 = s \right] \\ &= \mathop{\mathbb{E}}_{\substack{a_t \sim \pi(\,.\,|s_t) \\ s_{t+1} \sim p(\,.\,|\,s_t, a_t)}} \left[R(s_0, a_0) + \gamma \, V^{\pi}(s_1) \Big| s_0 = s \right] \\ &= \mathop{\mathbb{E}}_{\substack{a_t \sim \pi(\,.\,|s_t) \\ s_{t+1} \sim p(\,.\,|\,s_t, a_t)}} \left[\sum_{t=0}^{N-1} \gamma^t \, R(s_t,\, a_t) + \gamma^N \, V^{\pi}(s_N) \Big| s_0 = s \right] \end{aligned} \tag{2.52}$$

With this last equality, building an estimator would use the first N rewards and then rely back on the estimator. This idea is called the *N-step returns*. This scheme has the advantage of - somehow - relying more on samples from the environment. Since the estimator is weighted by γ^N a less accurate value estimator may be less harmful. This is especially relevant at the beginning of training when value networks have not undergone many updates. It appears this idea fits well with DDQN and SAC for the Q-networks updates. The changes to perform to the pseudo-code are minor. Only two parts are to be modified. The first one is the storing of the tuples inside the replay buffer. Instead of storing the individual rewards, one simply keep a discounted sum of the N last observed rewards. The second one is in the target computations where the value of the target estimators such as $\max'_a Q_{\theta'}(s',\, a')$ are now weighted by γ^N instead of N.

An important aside is to be made regarding N-step returns. Indeed, the changes performed to DDQN and SAC as just introduced are not perfectly sound. Even though these algorithms are off-policy, relying on up to N samples breaks their off-policyness. Hence, the updates are no longer *correct*. However, in practice it appears that ignoring this issue is reasonable. Consequently, when used in this work, N-step returns will be performed without further correction. A discussion on this issue and how to make the N-step return estimator more theoretically sound can be found in [Munos et al., 2016].

Chapter 3

The Lottery Ticket Hypothesis

3.1 The lottery ticket hypothesis

3.1.1 Original idea

The lottery ticket hypothesis was introduced by [Frankle and Carbin, 2018]. Here we provide the first definition of the lottery ticket hypothesis as given in this paper.

> **The Lottery Ticket Hypothesis**. *A randomly-initialized, dense neural network contains a subnetwork that is initialized such that — when trained in isolation — it can match the test accuracy of the original network after training for at most the same number of iterations.*
>
> Frankle and Carbin 2018

We are now going to dissect this hypothesis and rephrase the core ideas. It was first derived in the context of supervised learning. That is why it refers to a *"test accuracy"*. The hypothesis states that given a dense neural network architecture initialized randomly as it is most often the case - using an initialization sheme. There exists a *subnetwork* - a network obtained by removing weights from the network layers. This subnetwork when trained *in isolation* - as one would train a full network - obtains performance on par with the original network in terms of accuracy thus in a similar number of iterations. A subnetwork exhibiting these properties is said to be the winners of the *initialization lottery*. It is thus coined as a *winning ticket*. It appears this first definition of the lottery ticket hypothesis has been followed by a more cautious notion of winning ticket - later compared to a randomly initialized subnetwork as discussed in [Frankle et al., 2019]. Furthermore, the definition will be once more modified, this time regarding the notion of random initialization in order to make it applicable to a wider range of models. All these ideas will be defined more formally in Section 3.1.2.

Outline This chapter is divided into two parts. In the first part, we are going to focus on the lottery ticket hypothesis in general. In Section 3.1.2, we are going to study the formal aspects of the LTH. Subsequently, in Section 3.1.3, the algorithm introduced by [Frankle and Carbin, 2018] to find a subnetwork matching the LTH will be uncovered. Subsequently, a technique to find tickets more reliably will be introduced in Section 3.1.5. Finally the first half of this chapter will end with a brief set of reasons motivating the quest for well performing subnetworks.

In the second part of this chapter, the lottery ticket hypothesis in the context of deep reinforcement learning will be introduced. We will start in Section 3.2.1 with summary of the state of knowledge at the time of writing. Then, in Section 3.2.2, we will discuss how deep reinforcement learning is a different problem compared to supervised learning. We will discuss what might be the impact of these differences regarding the lottery ticket hypothesis.

3.1.2 Formal definition

In this section we provide the formal definition of the lottery ticket hypothesis. We first define a feed-forward neural network $\mathcal{N}$ with initialization parameters $\theta_0 \sim \mathcal{D}_\theta$. where $\mathcal{D}_\theta$ is the distribution over initial parameters. It is defined by the initialization scheme. The neural network $\mathcal{N}$ of parameters θ defines a functional mapping which is written $f(x\,;\theta)$. We define the operation that removes weights from the neural network as *pruning*. It is equivalent to setting the pruned weights to 0. Performing this action is also equivalent to multiplying element-wise the network parameters θ by a binary mask $m \in \{0,1\}^{|\theta|}$. This element-wise product is written $m \odot \theta$. The lottery ticket hypothesis is all about *initialization* parameters. Thus the object of interest will often be $m \odot \theta_0$.

Formal definition Here we provided a more formal definition of the LTH for supervised learning. It is borrowed almost as is from [Frankle and Carbin, 2018]. A neural network network with initial parameters θ_0 is optimized using stochastic gradient descent. It obtains mimimum validation loss l and test accuracy a at iteration j. A second neural network with initial parameters $m \odot \theta_0$ is trained similarly - on the same training set. This masked neural network reaches minimum validation loss l' and test accuracy a' at iteration j'. The lottery ticket hypothesis predicts that $\exists\, m$ which obtains commensurate accuracy ($a' \geq a$) in commensurate training time ($j' \leq j$) with a number of active weights significantly less than the full network size ($\|m\|_0 \ll |\theta|$).

Sparsity of mask Here we provide the notion of mask sparsity. It is defined by $S_m \triangleq \frac{|\theta| - \|m\|_0}{|\theta|}$ and is the ratio of inactive weights under the size of the neural network in parameters count. Similarly the *remaining ratio* is defined $R_m \triangleq \frac{\|m\|_0}{|\theta|} = 1 - S_m$. In order to avoid confusion with [Frankle and Carbin, 2018], R_m is what the authors - confusingly - called the *sparsity of the mask* and denoted P_m.

A more practical definition As said in the introduction the first notion of winning ticket has later been subject to change. The formal definition as provided above appears not to be very useful practically. Firstly, because it is vague regarding $\|m\|_0 \ll |\theta|$. Secondly, because reaching the performance of the full network might be a too hard target for networks whose sparsity is too large. This objective could hide the fact that a particular combination of initialization and mask $m \odot \theta_0$ is still somewhat special even though it does not compete with the full model. Another target - said to be more reasonable - is a so-called *random ticket* (Morcos et al. 2019, Frankle et al. 2019). A random subnetwork or random ticket as coined in the literature could be several things. On the one hand, a random ticket as referred to in [Frankle and Carbin, 2018] and [Frankle et al., 2019] is a winning ticket $m \odot \theta_0$ whose parameters have been reset such that it becomes $m \odot \theta_0'$. On the other hand, a random ticket is defined by [Morcos et al., 2019] as a subnetwork whose parameters θ_0' have been drawn randomly and whose mask m' was obtained as a permutation of m. Because the cost for finding random subnetworks may be negligible - especially when both the mask and the weights are selected randomly - they are expected to be good lower-bounds on the *compression-accuracy trade-off* [Gale et al., 2019]. Usually this notion of random mask is to be compared to a winning ticket with the same number of active weights such that $\|m\|_0 = \|m'\|_0$. When two tickets are compared we implicitly mean we compare the performance of these tickets after training - using them for initialization.

Using all these concepts, one could define a *good winning ticket* as a sparse initialization that beats the performance of a random ticket with the same number of active weights. At low sparsity, a random ticket can be expected to reach the same performance as the full network as shown empirically in [Frankle and Carbin, 2018]. Thus this criterion is close to the LTH for low sparsities. Sparsities at which a random subnetwork is able to compete with the full model are said to be *trivial spartisies* by [Frankle et al., 2019]. Indeed, since the cost of finding such a random subnetwork might be negligible, they might be said not to be really special. Sparsities for which the performance of a random subnetwork collapses are said to be *nontrivial*. Thus a sparse initialization whose performance after training seats between the full model and a random ticket is interesting. This will be the situation will be looking for throughout this work. The notion of random subnetwork will have to be specified more precisely though.

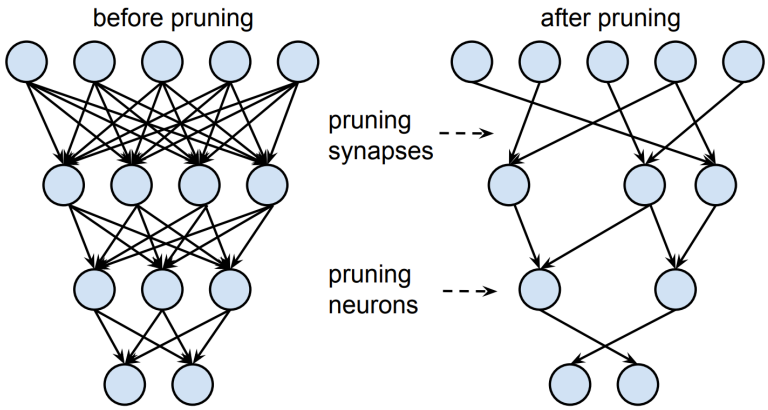

Figure 3.1: Network pruning as depicted in [Han et al., 2015]. **Left.** A multilayer perceptron has full connections. **Right.** The same network has some of its connections (synapses) as well as some of its neurons pruned. This last pruning can be considered as a special case of connections pruning.

3.1.3 Finding a winning ticket

Criterion: magnitude pruning

Along with the notion of LTH, the authors of [Frankle and Carbin, 2018] came with a process to find a sparse initialization which exhibits properties matching the LTH. The criterion they advocate to select which weights to prune is to look at the magnitude of the weights after training - breaking ties randomly. The weights whose L_1 norm are the largest after training are considered to be more valuable and impactfull. This heuristic choice leads to a criterion that allows to remove the less important weights. Since it is based on magnitude, it is referred to as *magnitude pruning*. A depiction of the idea of subnetworks and connections pruning can be found in Figure 3.1. Magnitude based pruning was formalized through the notion of *mask criterion* or *mask score* in [Zhou et al., 2019]. A mask criterion allows to compare and sort weights. We define a network initial parameters as θ_0 and its parameters after training as θ_f. The ith element of a set of parameters θ is written θ^i. Generally, a mask criterion can be depend on both the initial and final value of the parameter. Thus we define a mask criterion as $M(\theta_0^i, \theta_f^i) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Hence, the mask criterion corresponding to magnitude pruning is $M(\theta_0^i, \theta_f^i) = |\theta_f^i|$.

Operation on the remaining weights

From the magnitude pruning criterion, a pruning mask m can be extracted. Another contribution of [Frankle and Carbin, 2018] is to simply set the remaining weights to their original values. This operation is referred to as *rewinding*.

One shot pruning

The combination of magnitude pruning and rewinding naturally leads to an algorithm to find tickets. One needs to specify a pruning rate $p\%$. A network is initialized randomly and trained up to convergence. Then it is pruned, removing the $p\%$ weights having the lowest magnitude. The remaining weights are rewound to their original initializations. This algorithm reaches the desired pruning rate in one step. Hence, it has been coined as *one-shot* by [Frankle and Carbin, 2018].

Iterative magnitude pruning

One-shot magnitude pruning reaches the desired sparsity in one step. However, even though the magnitude of the weights at the end of training might be a good indicator of their importance, selecting all the weights to be removed once might not be reasonable. Indeed, the L_1 norm being a noisy signal, the larger the pruning rate, the larger the number of connections mistakenly removed. In order to alleviate this issue, [Frankle and Carbin, 2018] already introduced a variant of one-shot pruning. This algorithm is called *iterative magnitude pruning* (IMP). It simply applies one-shot pruning n times, removing $p^{\frac{1}{n}}\%$ of the weights at each iteration. Every new iteration $j+1$ simply uses $m_j \odot \theta_0$ as its initial parameters.

These parameters are used for training and only non-pruned weights are considered for removal. It appears one-shot pruning is just a special case of IMP with $n = 1$. More formally, iterative magnitude pruning is provided in Algorithm 4 using $M(\theta_0^i, \theta_f^i) = |\theta_f^i|$.

Algorithm 4: Iterative magnitude pruning

Inputs:
$p\%$, the pruning rate, float $\in [0, 100]$
M, a masking criterion: $M : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$
$\mathcal{N}$, a neural network architecture which defines a mapping $f(x\,; \theta)$
$\mathcal{D}_0$ a distribution over initializations for $\mathcal{N}$
$\mathcal{L}$, a training algorithm, $\mathcal{L} : \mathbb{R}^{|\theta|} \to \mathbb{R}^{|\theta|}$

Outputs:
$m \odot \theta_0$, a lottery ticket, $m \odot \theta_0 \in \mathbb{R}^{|\theta|}$

Initialization:
Draw the initial parameters with $\theta_0 \sim \mathcal{D}_0$
Initialize the pruning mask $m = \mathbf{1}^{|\theta_0|}$

Iterations:
for $j = 1, ..., M$ **do**
  Train the neural network, keeping the mask frozen
  $m \odot \theta_f \leftarrow \mathcal{L}(m \odot \theta_0)$

  Compute the mask scores of non-pruned weights using
  $s^i \leftarrow M(\theta_0^i, \theta_f^i)$ for $0 \leq i < |\theta|$

  For the $(100 - p)\%$ top scores of non-pruned weights (break ties randomly):
  $m^i \leftarrow 1$
  For the $p\%$ bottom scores of non-pruned weights:
  $m^i \leftarrow 0$
end

Global pruning and layerwise pruning

Layerwise pruning removes a same ratio of weights for every layers. It ranks the weights scores layer per layer. Global pruning gives more flexibility by pooling all the layers parameters. Hence, every layer may have a different pruning rate. This idea was already introduced [Frankle and Carbin, 2018] as they observed that using global pruning on deeper networks allowed to reach larger sparsities. They explain this observation by the diversity of layer sizes in such networks. They claim smaller layers become bottlenecks. The idea is that - with layerwise pruning - one has to remove weights at the same ratio for every layer. However - for small layers - removing $p\%$ of its weights might have more impact than removing that same percentage on a larger layer. This claim was also supported experimentally in [Morcos et al., 2019] and [Paganini and Forde, 2020] where they observed global pruning allowed to reach larger sparsities before collapsing to a random subnetwork performance. They also note that - with global pruning - IMP leads to very different pruning rates across the layers. A similar observation that allowing model sparsity to be different between layers helps the performance was also made in the context of architecture and compression search [He and Han, 2018].

Discussion: Structured pruning and unstructured pruning

Magnitude pruning is said to be an *unstructured* way of removing weights from a neural network since it removes connections one by one. Methods removing units in dense layers or channels in CNNs are coined as *structured*. The L_1 magnitude unstructured pruning method has been compared to L_1, L_2, L_∞ structured pruning in [Paganini and Forde, 2020]. They carried experiments on supervised image classification using conv-nets. Their results tends to show that L_1 unstructured pruning is superior to any structured variant in their test scenario. They observed L_1 structured tickets to be able to reach larger sparsities before collapsing performance-wise.

Discussion: Other criterion choices

Although the L_1 norm of the trained weights as a criterion may seem arbitrary it has been since backed-up by empirical comparisons with other structured pruning criteria. In [Zhou et al., 2019], the authors compared the L_1 norm of the trained weights as well as the untrained weights. They also studied how comparing the initial and the trained weights could lead to a pruning criterion. Their experiments were performed on supervised classification with fully connected networks as well as conv-nets. One of their main results supports the pruning of the lowest L_1 norm weights.

3.1.4 Learning rate warmup

Along with the lottery ticket hypothesis and itterative magnitude pruning the authors of [Frankle and Carbin, 2018] came with an additional trick to find winning tickets more reliably and potentially more rapidly. Their experimental setup involved Lenet-300-100 architecture on MNIST dataset [Lecun et al., 1998]. They were also able to find winning tickets on conv-nets using scaled down-versions of VGG [Simonyan and Zisserman, 2015] tailored for the CIFAR10 dataset [Krizhevsky, 2009]. Within these two scenarios, the standard IMP algorithm as introduced was able to extract subnetworks that - when trained in isolation - were able to keep full network accuracy for large pruning rates. It was respectively $R_m \geq 3.6\%$ for LeNet and $R_m \geq 2\%$ for the scaled-down VGG models. However, when applying IMP to more involved architectures the observations are more nuanced. They experimented on a full special purposed VGG-19 architecture [Liu et al., 2018] as well as a Resnet-18 architecture [He et al., 2015a]. They observed the discovery of winning tickets to be very sensitive to the learning rate used for training. Indeed, they often had to use values one order of magnitude lower than the ones usually advocated for their architecture/dataset combination. They observed tickets found with larger learning rate not to perform better than the same subnetworks with reinitialized remaining parameters (random reinit). This observation was latter supported by Liu et al. [2018] with similar experimental conditions. In order to find ticket without lowering the learning rate too much - penalizing the learning speed - they suggested the use of a *warmup strategy*. This technique simply involves to start at a very low learning rate and increases linearly in the number of training steps k up to a target rate α_t. The usual learning rate scheme is then applied without change. This warmup procedure allowed the authors to find winning tickets while minimizing the change in training settings required to do so.

3.1.5 Late rewinding: finding a ticket stably

As mentioned in the introduction of this chapter, it appears the definition of the lottery ticket hypothesis had to be changed one more time. Following the introduction of the LTH, work has been done to try to apply the LTH for real-world architectures [Gale et al., 2019]. The authors applied standard IMP on Resnet-50 for ImageNet [Russakovsky et al., 2014] and Transformer architecture [Vaswani et al., 2017] for language translation. They were unable to find tickets exhibiting the properties of the LTH. In a follow-up work, the authors of the LTH exposed a variation of IMP that is said to make it more reliable, especially on larger architectures [Frankle et al., 2019]. This variation simply involves not to rewind to the initialization parameters θ_0 but to the parameters obtained once after k learning iterations θ_k. Since the parameters are reset to latter parameters, this trick is named *late resetting* or *late rewinding*. According to the authors, the number of iterations k should be much smaller than the total number of training iterations ($k \ll K$). As a vocabulary aside, if $k > 0$ and the ticket performs better than a random ticket at similar sparsity, this ticket is called a *matching ticket*. The special case $k = 0$ is the usual *winning ticket*. The authors support their idea of late resetting with a tool they introduced coined *instability analysis*. Their discussion is involved and outside the scope of this work. This new IMP variant implies to change the lottery ticket hypothesis. Hence, the new and more general lottery ticket hypothesis allows for the lucky initialization to be parameters obtained after some training. It also involves to change the IMP algorithm. The new version is highly redundant with Algorithm 4 and is provided in Appendix B.1: Algorithm 6.

3.1.6 On the interest of finding and researching on winning tickets

As will be made obvious in subsequent sections, IMP is an excruciatingly expensive process. The usual number of IMP iterations practitioners employ is 31. This implies to scale the total neural network

training time by a similar amount. Hence, one might wonder what are the motivations behind the application of this technique in the first place and the research on topics related to it in a second place. This actually draws the line between what could already be made available and what might be awaited for in the future.

(Near) Immediate benefits A winning ticket is a sparse initialization which - trained in isolation - is expected to keep good performances. As such, finding a winning ticket can be seen as an instance of model compression. Hence, many of the practical benefits of compression could follow. For example, in the recent years, many models have become prohibitively large for wide spread adoption. The model may require too much memory or simply be too time consuming to be trained by smaller research teams. Hence, network compression through IMP could be greatly beneficial.

However, an important point has to be made regarding the structure of the sparse model extracted by IMP. Indeed, as said before, a winning ticket found by IMP is *unstructured*. Consequently, the weights matrices are sparse but have no constraints on where to put the zeros. In the case of linear layers, the weights matrices could be stored in sparse-matrix formats. Depending on the model sparsity, this could potentially save storage and compute time through optimized sparse matrices operations (sparse matrix vector products). Standard libraries such as PyTorch [Paszke et al., 2019] and TensorFlow [Abadi et al., 2015] have been slow at integrating optimizations for sparse models even though work seems to have been done in that direction. At the time of writing, TensorFlow seems to be the most advanced in that regard with automatic tools to select the optimal sparse format and optimize operations [Li and Ablavatski, 2021]. However, this is not clear whether they would improve memory/time performance for unstructured sparsity. Furthermore, they are - at the moment - only available on CPU. The lower memory footprint of pruned networks could make possible their load into faster but less plentiful memory which could reduce energy computation and latency even more. However, exploiting the sparsity of a pruned model to its full potential might require custom accelerators as suggested in [Han et al., 2016]. Oppositely, structured pruning removes entire neurons or entire CNN channels. Hence, usual software and hardware implementations can immediately spare operations and thus save on compute time.

Research interests Research motivations for the LTH are plentiful. Here we provided an non-exhaustive list of them. In [Frankle and Carbin, 2018], the authors points toward an observed better generalization on supervised image classification. Winning tickets have been shown to be - in some scenarios - re-usable across datasets and optimizers in [Morcos et al., 2019]. This could potentially mitigate the large cost of IMP by simply reusing a winning ticket for different tasks. They also suggest the possibility to parametrize a distribution over winning masks. This would allow to sample new - datasets independent - generic masks. This also suggests there is a gap to be filled between current initializations schemes and what is achievable with winning tickets initialization. In [Paganini and Forde, 2020] the authors have introduced a parallelizable alternative to standard IMP which they showed to be competitive for some sparsities.

3.2 The lottery ticket hypothesis and deep reinforcement learning

In this section we discuss the lottery ticket hypothesis in the context of deep reinforcement learning. We start in Section 3.2.1 by exploring the results already available in that scenario. Then in Section 3.2.2 we briefly point out the core differences between deep reinforcement learning and deep supervised learning which might be interesting regarding the LTH.

3.2.1 Results available

To the best of our knowledge [Yu et al., 2020] and [Vischer et al., 2021] are the only two published works investigating the LTH in the context of deep reinforcement learning. In this section we will summarize the core results introduced by these papers. Firstly, we will briefly summarize the choice of algorithm, architecture and general setup each paper proposed. This will be useful in order to compare with our own experimental setup.

Papers setups

In the first paper studying the LTH and DRL (Yu et al. 2020), the experiments were performed on classic control and atari games from the Arcade Learning Environment [Bellemare et al., 2012]. The RL algorithm they used was A2C [Mnih et al., 2016]. This algorithm is policy-gradient based and thus optimizes directly an estimator of equation 2.6. Furthermore, A2C is an on-policy algorithm which exhibits a stochastic policy. For classic control they used MLPs with 3 hidden layers with 128 or 256 units. In the case of Atari games they used a sequence of convolution/max-pooling and finally some linear layers. The authors performed IMP with 20 iterations removed the 20% weights with lowest magnitude. The pruning was performed globally and included both weights and biases. They compared their tickets to random subnetworks which we assume to be a combination of a random mask and a random initialization.

Finally, the second paper from [Vischer et al., 2021] considered two algorithms, one from the value-based family (DDQN) and one from the policy-gradient based family (PPO). Proximal Policy Optimization or PPO was introduced in [Schulman et al., 2017]. It is an on-policy algorithm with actor-critic architecture and it exhibits a stochastic policy. PPO works for both discrete and continuous action spaces. The authors pruned 20% of the weights globally and performed a number of IMP iterations - we assume to be - equal to 31 (the authors said to follow Frankle and Carbin [2018]). The authors did not specify whether they pruned both weights and biases. For PPO, the authors pruned both the actor and the critic. The authors experimented on classic control tasks and simplified Atari games (MinAtar) from [Young and Tian, 2019] for pixel control. They worked 2 hidden layers MLP with between 128 and 512 units width for classic control and between 512 and 1024 units for pixel control. They also experimented with CNNs architecture scaled for MinAtar as specified in [Young and Tian, 2019].

A summary of the important features from the two reference paper is provided in Table 3.1 for quick look-up.

Feature	Yu et al. 2020	Vischer et al. 2021	This work
IMP steps	20	31	31
Pruning rate	20%	20%	20%
Algorithms	A2C	DDQN, PPO	DDQN, SAC
Classic control	OpenAI Gym	OpenAI Gym & PyByllet Gym	OpenAI Gym & PyByllet Gym
Pixel control	ALE	MinAtar	MinAtar
Prune bias	Yes	Unknown	No
Global pruning	Yes	Yes	Yes
Late rewinding	Yes	No	Yes

Table 3.1: Table of important features from the two reference papers Yu et al. 2020 & Vischer et al. 2021. Comparison with this work.

The LTH holds with DRL The authors of [Yu et al., 2020] were the first to show the existence of winning tickets in the context of DRL. They were able to find winning tickets on 3 classic control tasks as well as 9 Atari games. On classic control the results were very neat with winning tickets able to withstand larger pruning rates than random tickets. On Atari, their results were more contrasted. They were sometimes able to observe large differences with the random baseline but on 4 out of 9 tasks the winning tickets did not perform significantly better than random tickets.

The existence of winning tickets for DRL was confirmed a second time in [Vischer et al., 2021]. They found tickets for continuous and discrete classic control. For pixel control, they found tickets both using MLPs and CNNs.

Late-rewinding The late-rewinding trick was tested by [Yu et al., 2020] where they used the parameters obtained after $4-5\%$ of the normal number of training iterations. On classic control, the trick helped at best and never made the collapse in performance happen sooner in terms of sparsity. On the Atari games, they tested three games and showed very significant positive impact of the late-rewinding trick. On the last game, the influence of late-rewinding was negligible but not really harmful. Oppositely, [Vischer et al., 2021] did not find late rewinding to be helpful.

One-shot pruning One-shot pruning was found to be unable to find winning tickets in two out of three classic control tasks by [Yu et al., 2020]. Iterative magnitude pruning is helpful for the three experiments. Hence, their results suggest IMP combined with late-rewinding could be a good combination for DRL.

The input mask is specialized In Vischer et al. 2021, the input mask - the mask of the input layer - was found to be specialized for every task. They observed, IMP to be able to remove redundant variables by pruning every weight connected to them. They found IMP focused on important area of the image and put less weights on non-influential or redundant pixels.

Sensitivity to initialization scheme The IMP procedure was found to be relatively sensitive to the weight initialization scheme. Indeed, for some initialization schemes the larger the number of inputs of a layer the lower the magnitude of the weights (i.e the so-called *Kaiming family*. He et al. 2015b). Oppositely, some schemes do not exhibit that property (i.e *Xavier initialization* Glorot and Bengio 2010). Consequently, the authors found Kaiming initialization to bias the weights magnitude of the input mask because it tends to have lower number of entries.

3.2.2 Differences between supervised learning and value-based DRL

In this section, we briefly highlight some differences between supervised image classification and the class of deep reinforcement learning algorithms we used in this work. These are which - we humbly think - might have an impact for finding and studying winning tickets.

Network overparametrization Overparametrization is a core idea for explaining the LTH and the existence of winning tickets [Frankle and Carbin, 2018]. They hypothetized that the more overparameterized a neural network, the more likely the existence of a well-performing subnetwork. Even though this notion of overparametrization seems not to be defined formally, one may wonder whether the networks used in DRL are as overparametrized as their supervised image classification counterparts.

It appears the common network sizes used in DRL have very large room for improvement. In both [Ota et al., 2021] and [Neal and Mitliagkas, 2019] the authors suggested using larger layers (up to 2048 units) could improve performance and sample efficiency. Their observations apply to both value-based and policy-gradient algorithms. However, [Ota et al., 2021] also found increasing the depth of the networks not to be improving agent performance thus because of training instability. Anyway, the first observation could suggest the common scale of networks used in the field is not as overaprametrized as it could be. This might have an impact on the ability to find winning tickets or the maximum pruning rate achievable before performance collapse.

The learning distribution evolves With supervised learning a loss is minimized on a training set which is fixed. Hence, the training distribution can be said to be constant. Oppositely, deep reinforcement learning algorithms train from samples whose distribution may vary. In general, the training distribution evolves as the agent learns the task and collects different samples. It also evolves with the exploration scheme which may change over the course of training (i.e epsilon-greedy with decay). As pointed out in [Vischer et al., 2021] it is not obvious whether IMP will be able to identify tickets which are valid on the whole course of training as the agent and exploration evolves.

Chapter 4

Experimental framework

In this chapter, the experimental framework upon which the results of this work were extracted from is introduced. In Section 4.1, we start by introducing the training mechanism used for the two *off-policy* algorithms we considered. Subsequently, in Section 4.5 we discuss some non-trivial choices we made for most of our experiments. Finally, in Section 4.6, we discuss the heavy parallelism that was applied to carry on the experiments. This parallelism appeared to be crucial regarding the scale of the experiments this work present.

4.1 The training procedure: synchronous APEX

Both algorithms we considered in this work are off-policy. It implies the process generating the (s, a, r, s') tuples can be decoupled from the process sampling the replay buffer and performing the actual learning. This idea is the foundation for the APEX framework introduced in [Horgan et al., 2018]. Within this framework, the exploration is decoupled from the learning both conceptually and physically since it can be carried on different CPU cores or even machines. The authors introduced the concept of *actors*, *learner* and a *centralized replay buffer*.

The *actors* are distributed processes running each a version of the environment and every part of the agent that is necessary to use its policy. Hence, every actor has an independent version of the neural networks and their parameters. The *learner* is an independent process that samples tuples from the replay buffer and performs the update. The new parameters are sent regularly to the actors allowing the exploration to improve. The idea of APEX is simple and arises in many other areas of scientific computing. Since the process performing the exploration cannot be made faster in itself, several processes are spawn which allows to leverage more compute power. Deep reinforcement learning algorithms are known to require many samples to learn anything useful. In the case of the Arcade Learning Environment introduced in [Bellemare et al., 2012] - which is a very popular benchmarking set of environments - it is very common to see algorithms requiring millions or tens of millions of samples to learn a single task. In their paper, the authors of APEX presented the benefits of their approach using *hundreds* of actors and thus *hundreds* of CPU cores. This - of course - would not scale well with the number of experiments that had to be performed in this work. This will be made obvious in subsequent sections. However, we found out that only a few *actors* could already increase the speed of learning substantially. In most of our experiments we used only four actors running each on their own CPU core. Since the exploration can be performed on CPU cores that - in our case - were plentiful, we were able to give an important hardware budget to the learner process. In every of our experiment, the learning process was performed on GPU which is not so common for deep reinforcement learning. This allowed us to use important batch sizes and perform more stable learning.

4.1.1 Modifications over APEX

In this section we present briefly a set of simplifications we made over the classic APEX framework as introduced in [Horgan et al., 2018]. A scheme of our modified version of APEX is provided in Figure 4.1.

The replay buffer In their paper, the authors of APEX used a *centralized replay buffer* based on *prioritized experience replay*. It is an important concept introduced in [Schaul et al., 2016]. Briefly, it focuses the learning on samples that the algorithm has more trouble with. In order to do so it samples more frequently tuples whose predicted *Bellman residual* are larger. We tested this idea with both of the algorithms we considered and found out it had little benefits with respect to the additional computation burden. Moreover, experience replay adds a layer of complexity that might complicate the analysis of the results. For these reasons, we decided to never use the prioritized experience replay. Thus, the replay buffer is a simple list from which samples are drawn uniformly.

Synchronicity The APEX framework as it was introduced is *asynchronous*. It implies the learner will send updates on the neural networks without strong time guarantees. Furthermore, the actors processes explore as fast as they can without constraints on the number of samples they generate and the relative age of the policy they use. It is not a flaw in its own since APEX was introduced to make learning faster and leverage as much hardware as possible which it successfully did. However, these two observations prevent reproducibility. Indeed, even though we assume the number of actors is constrained as well as the number of CPUs/GPUs. There might be no guarantee on the workload already running on the machines we used as well as no constraints on the exact hardware components. It is especially relevant in our case since the computation is run on a shared cluster. Consequently, we propose to modify APEX into a framework that simply enforces synchronicity. It appears to be a very simple form of parallel exploration. With our implementation two parameters determine how samples and updates are ordered. The first one is the *parameter update frequency*. It is the number of steps of the learning process - usually the number of batch drawn and gradient updates - between two updates sent out by the learner process. The second parameter is the *ratio actor update* parameter. It is simply the number of samples to draw on each actor process between two parameter updates. The newly generated samples are not sent to the replay buffer on the go but only during parameter updates. On one side, it implies that the learner will perform updates with a given state of the replay buffer and then wait until the actors are done. On the other side, the actors will produce a given number of samples and then wait for the learner. When every job is done, the samples are sent to the replay buffer, the learner sends the new parameters to a parameter server and every task resumes. We call this version of the framework *synchronized APEX*. A scheme of the loop is depicted in Figure 4.2

Evaluation In our version of APEX, the evaluation is carried on explicitly by a process whose job is to perform a given number of episodes with the last version of the agent. These episodes are used in order to check whether some convergence or early stopping criterion is met. The stop signal is sent outward to every other job. This evaluation is performed asynchronously with the last parameters available. It does not block the other processes.

4.1.2 Parallel exploration strategy

One of the main benefit of parallel exploration for off-policy algorithms is the possibility to have different exploration strategies. Consequently, when epsilon greedy exploration is used, we set different epsilon start values for every actor. The final epsilon value is fixed as well as the decay steps. Hence, during the epsilon decay phase, every actor will explore differently which brings more diversity to the training distribution.

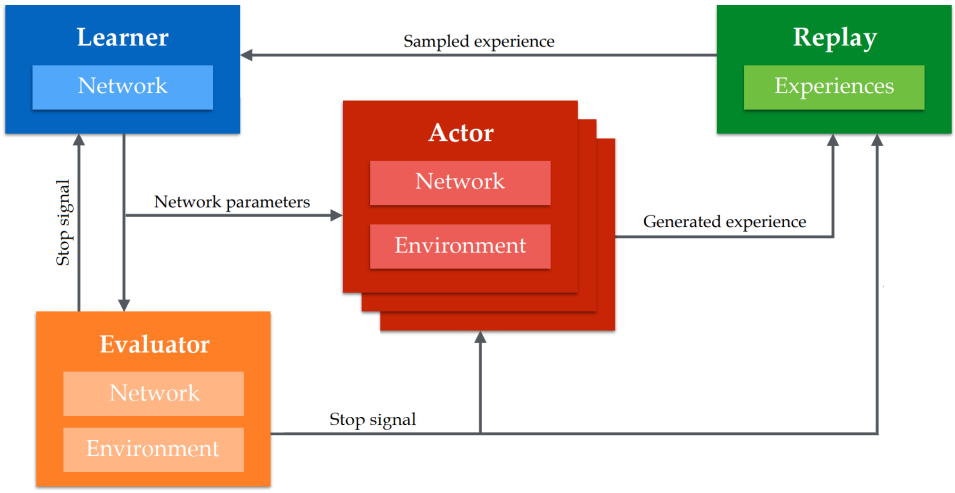

Figure 4.1: Modified version of a scheme provided in [Schaul et al., 2016]. These are the components involved in our version of APEX which we called *synchronized APEX*. There is a learner process running on GPU and actors each running on their own CPU core. The replay buffer stores the samples and run its own process as well. Parameters and samples are sent at a fixed pace which maintains order no matter what computing power is available. An additional process performs evaluation every time a new set of parameters is sent. A stopping criterion is defined and checked. The stop signal is sent out to every other job.

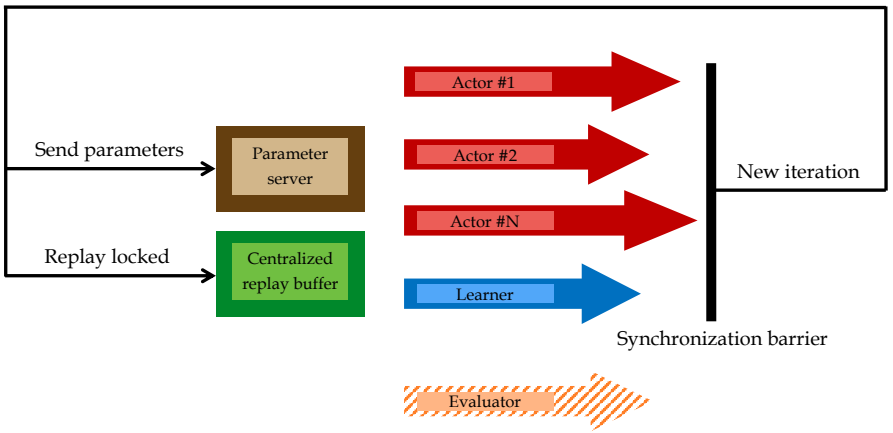

Figure 4.2: Synchronized Apex. A learner, N actors and an evaluator run concurrently. The actors fill each a local buffer of samples and wait for the synchronization barrier. The learner performs a fixed number of updates. The evaluator aggregates a fixed number of episode but does block the other processes. Once the barrier is reached, the learner sends the parameters to the parameter server, the actors send their local data to the centralized replay. Then the replay buffer is locked, the parameter server sends out the parameters. Finally, a new iteration begins.

4.1.3 Real-world benefits

In this section we try to exhibit the benefits of using parallel exploration to generate more samples in a given amount of time. In Figure 4.3, the performance of DDQN agents as training progresses is depicted for cartpole and MinAtar-SpaceInvaders. As can be seen, on the two left charts, the impact on sample efficiency - the number of environment interactions to learn a task - is not clear. On the one hand, for cartpole DDQN takes more samples to reach the maximum reward. On the other hand, for SpaceInvaders it reaches better performances for a given number of environment interactions. However, this charts relates to sample efficiency which is not the raw time computation. The time consumed can be seen on the two right charts. In the case of cartpole, even though using more actors is less sample efficient is it actually compensated by a larger number of samples generated in a given time period. In the case of SpaceInvaders it is even more clear. In this case both curves stop when the maximum number of samples is generated. With four actors APEX takes a quarter of the time to reach that limit. Critically, it does so while having better performances almost all the way.

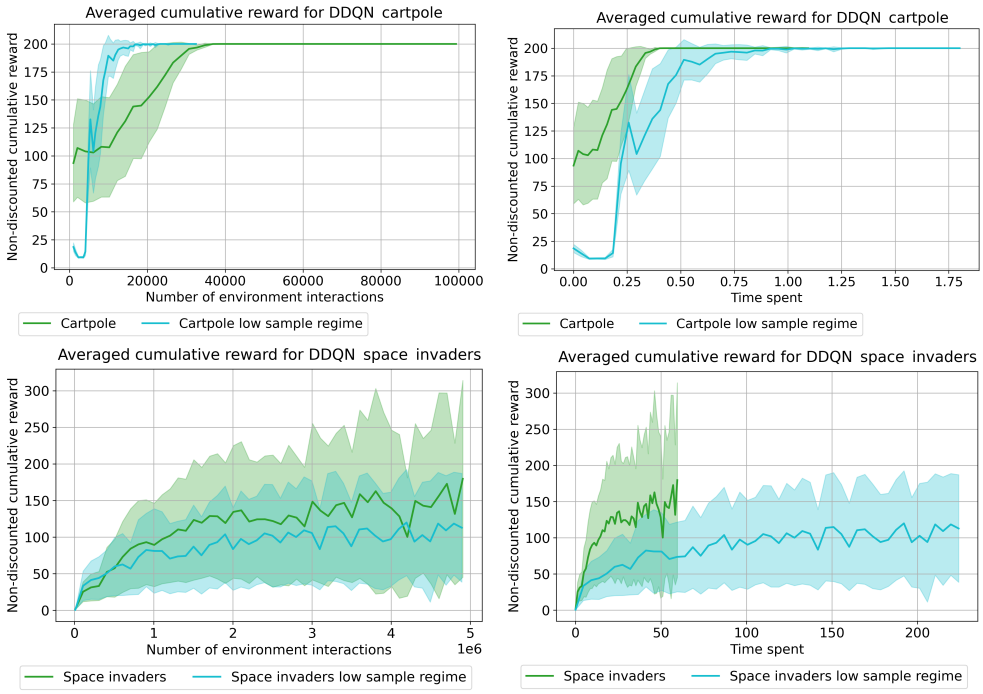

Figure 4.3: Comparison of low sample regime (one APEX actor) and the larger sample regime (4 APEX actors) with respect to the number of environment interactions or time spent in minutes. **Top left.** DDQN cartpole with one APEX actor, on the go evaluation as a function of the number of environment interactions. **Top right.** DDQN cartpole in a larger sample regime with 4 APEX actors, on the go evaluation as a function of the time spent (in minutes). **Bottom left.** Same as top left for DDQN space invaders. **Bottom right.** Same as top right for DDQN space invaders. **Observations.** Adding more actors may not make the learning more sample efficient but for equal time spent the performance of the agent is larger with more actors.

Environment name	Library	State space dim	Action space
CartPole-v0	OpenAI Gym	4	$\{0, 1\}$
Pendulum-v0	OpenAI Gym	3	$[-2\,;\,2]$
InvertedPendulumPyBulletEnv-v0	PyBullet Gym	5	$[-1\,;\,1]$
Acrobot-v1	OpenAI Gym	6	$\{0, 1, 2\}$
LunarLander-v2	OpenAI Gym	8	$\{0, 1, 2, 3\}$
InvertedDoublePendulumPyBulletEnv-v0	PyBullet Gym	9	$[-1\,;\,1]$
AntPyBulletEnv-v0	PyBullet Gym	28	$[-1\,;\,1]^8$
HalfCheetahPyBulletEnv-v0	Pybullet Gym	26	$[-1\,;\,1]^6$

Table 4.1: Classic control environment table description. OpenAI Gym: [Brockman et al., 2016]. PyBullet Gym: [Ellenberger, 2018–2019]. First column is the reference environment name for cross comparison. Then follows, the library which made the environment available. State space, number of dimensions. Action space, either continuous (with bounds) or discrete).

4.2 Environments and training settings

In this section we discuss the describe the set of environments the experiments were carried on as well as general settings used for training.

4.2.1 Classic control

Classic control involves to seek optimal behaviour for an agent acting within a physical simulation. We picked a large complexity range spanning from a simple pendulum to the walk of an ant. A description of the state space and action space of the environment we used in this work can be found in Table 4.1. Some environments are part of OpenAI Gym [Brockman et al., 2016] and others belong to PyBullet Gym [Ellenberger, 2018–2019].

In all experiments, 4 actors from synchronized apex were used as well as a batch size of 256 and a maximum number of samples equal to 750 000 or 1 000 000. The late rewinding was set to 5% of the total maximum number of samples. Adam was always used as default optimizer. For DDQN, the learning rate was usually set to $\alpha = 5 \times 10^{-4}$ and $\epsilon = 10^{-4}$. For SAC, the learning rates for the actor and the critic were $\alpha_Q = \alpha_\pi = 10^{-3}$ and $\epsilon = 10^{-4}$. The temperature parameter was learned with a learning rate $\alpha_\lambda = 10^{-4}$ and an initial value $\lambda_0 = 0.25$. For synchronized apex, the *parameter update* frequency was set to 64 and the *ratio actor update* was equal to 4. The maximum size for the replay buffer was 10^6 tuples. Epsilon-greedy exploration was used for DDQN with different parameters for each actor. No added exploration was necessary for SAC classic control. The main hyperparameters as well as the stopping criterion parameters and exploration noise can be found in Appendix B.2

4.2.2 Arcade environments

Environments from the Arcade Learning Environment were too complex to be ran within our computation budget. A simplified versions of 6 Atari games was introduced in [Young and Tian, 2019]. The library is called *MinAtar* and exhibits complex dynamics which can reasonably compared to the Atari games they emulate but with much simpler state space. Indeed, the state space is simply a $N \times 10 \times 10$ semantic grid whose channels can be interpreted. The action space is discrete with 6 choices. A description of the MinAtar environments we used in this work is available in Table 4.2.

The hyperparameters were similar to classic control. The maximum number of samples was 5×10^6. The late resetting was set to 1%. For synchronized apex, the *parameter update* frequency was set to 64 but the *ratio actor update* was equal to 8. In the case of SAC actor and critic learning rates were lowered to $\alpha_Q = \alpha_\pi = 5 \times 10^{-4}$. Furthermore, the number of steps for N-steps return was changed from 1 (normal) to 5 which helped a lot during training.

Environment name	State space dim	Action space
asterix	$4 \times 10 \times 10$	6
breakout	$4 \times 10 \times 10$	6
freeway	$7 \times 10 \times 10$	6
space_invaders	$6 \times 10 \times 10$	6

Table 4.2: Simplified arcade environments specification from MinAtar [Young and Tian, 2019] First column is the reference environment name for cross comparison. Then follows state space number of dimensions and number of discrete actions.

4.3 Network architectures

In this section we briefly expose the architectures that were used for both algorithm in classic and pixel based control. In every experiment, the networks were 3 layers with thus 3 sets of parameters to be learned. In the case of classic control, the hidden layer was 256 units in width. For pixel control, it was increased to 512 units. The ReLu activation was used for every network and every environment. In the case of pixel control, we borrowed the input layer architecture from [Young and Tian, 2019]. It is simply a set of 16 kernels of size 3×3 with stride 1. It is followed by a sequence of linear layers as for classic control. In the case of SAC for continuous actions, one Gaussian per action is parametrized such that there are two outputs, one for the mean and one for the standard deviation. The architectures can be found in Appendix B.3

4.4 Ticket evaluation

The tickets were evaluated in two different ways. Firstly they were tested throughout the experiments. Every time a new set of parameters is sent out by the learner (every *parameter update* $\times$ *ratio actor update* learning steps), between 10 and 50 episodes (depending on the task) were played by the agent. Secondly, the tickets were evaluated after training on 1000 episodes. In both cases, no noise was added on the actions. Furthermore, in the case of SAC, the action with the largest probability/density was picked instead of sampling from the distribution. For SAC discrete, it involves a simple look-up over the action distribution. For SAC continuous, we parametrized a Gaussian per action dimension. Hence, the largest probability action is the mean vector.

4.5 Implementation choices

In this section we give some details about choices we had to make regarding aspects of our experiments. Some of them are obvious, some others are less. Nonetheless, we considered these deserved to be clarified.

No pruning of the bias parameters We made the choice - for every experiment - to not prune the biases. Indeed, we assumed the number of parameters contained in the biases not to be large enough to have an impact on our experiments and their conclusions.

Pruning all the networks simultaneously In all of our experiment we decided to prune all the networks that could be pruned simultaneously and at the same rate. In the case of soft actor critic it implies to prune the two soft Q-networks and the policy network at the same rate for a given step of iterative magnitude pruning. This choice was made because it is the simplest one. One could have decided not to prune one of the two networks or to prune one slower considering it needed more parameters than the other.

Pruning of the target networks Both DDQN and SAC make use of target networks which are updated using equation 2.31. The way to handle this update with pruned network is obvious. The target networks simply use the mask of the original network. Since the mask is not changed during the

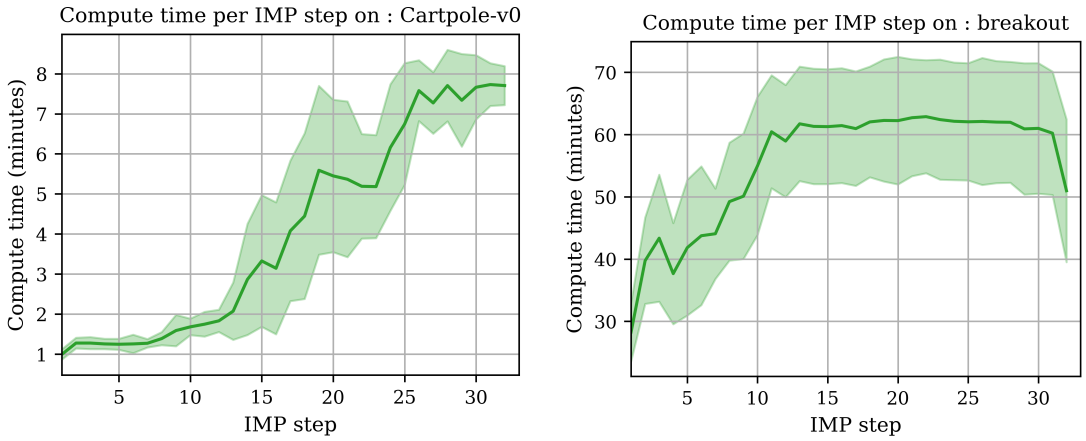

Figure 4.4: Averaged times per training step for the IMP loop. Means and standard deviations computed from 5 repetitions. **Left.** Cartpole-V0 trained with DDQN. **Right.** Breakout environment from MinAtar, trained with DDQN. Time per IMP step tends to increase - or may stay stable after some IMP steps - as training takes longer to reach the stopping criterion or the maximum number of steps. **Observations.** Training time increases as IMP progresses such that one may stop it earlier for the first IMP steps.

learning process - between iterative magnitude pruning steps - it can be kept equal and the remaining parameters are updated with polyak averaging.

Stop training earlier with target reward In all of our experiments we decided to define a reward target. Additionally, a maximum number of generated samples is provided. Using the *parameter update frequency* and the *ratio actor update*, it also defines a maximum number of training steps. The evaluator will check at every new update of the agent parameters whether the performance target is met. In order to do so the evaluator will roll $N_{\text{evaluator}}$ and average the discounted cumulative rewards. The target is met if this average is above or equal to the threshold thus for $N_{\text{r\_stop}}$ consecutive iterations of synchronized APEX. When the target happens to be reached - if ever - the evaluator sends a stopping signal to all the others jobs. This early stopping is especially useful in the case of simple tasks or during the first iteration of iterative magnitude pruning. Indeed, in these cases, the target is often reached very quickly. However, as pruning progresses the agent tends to take much longer to learn. This slow-down requires to define very large number of maximum learning steps / samples generated. Consequently, the target reward criterion prevents a large waste of computation time. In Figure 4.4, the large amount of time that would be wasted without this criterion is exhibited. As can be seen - as IMP progresses - training tends to take longer to reach the performance criterion of the agent. If no target reward was used, the first steps would take as long as the last IMP steps.

Always use late-rewinding In all our experiments - except when the opposite is said explicitly - we applied the late-rewinding trick. The number of training steps k for parameter rewinding is usually 5% of the number of steps for the normal training. This value is used by default. A discussion about the reason of this choice is to be found in Section 5.1.1.

Networks initialization Unless explicitly said otherwise we used the same initialization algorithm for every network every algorithm and every experiment. For dense layers, orthogonal initialization was used for the weights and zero initialization for the bias. For convolutional layers as well as transposed convolutional layers, the *delta-orthogonal* initialization scheme was applied. It is introduced in [Xiao et al., 2018]. Its usage was advocated in [Yarats et al., 2019]. Observing it worked for our experiments, we decided to generalize this choice for all of them.

4.6 Experiments organization

In this section we give details about how the different experiments were implemented. We first discuss the large computational cost involved with iterative magnitude pruning and why parallelizing the experiments was critical. Secondly, we provide more details on how we split the load for three of the main tasks to be performed to get the results discussed in subsequent chapters.

4.6.1 Heavy parallelism

In the case of iterative magnitude pruning - as discussed before - the learning has to be re-performed at every iteration. In most of our experiments, the standard 31 number of IMP steps was used. In Table 4.3, an estimation - computed from real experiments - of the time consumed to perform all the 31 IMP steps is provided for several algorithms and tasks. If we had to perform each of this experiment sequentially, the times provided would have to be multiplied by the number of experiment repetitions. This number was usually set to 5 or 10. It would clearly make the cost of experiments prohibitive. For this reason, we had to parallelize the whole IMP process by simply spawning several experiments at once.

Algorithm	Environment	Time for 31 IMP steps (minutes)
DDQN	CartPole-v0	129.43 ± 12.85
DDQN	LunarLander-v2	948.66 ± 117.63
DDQN	MinAtar-breakout	1776.65 ± 268.75
DDQN	MinAtar-asterix	1819.42 ± 253.76
SAC	Cartpole-v0	200.32 ± 20.23
SAC	inverted-pendulum-v0	731.70 ± 21.67
SAC	MinAtar-space_invaders	3763.26 ± 60.19

Table 4.3: Elapsed times to perform the 31 IMP steps for different combinations of algorithm and task. Times are given in minute, averaged with standard deviations over all the runs.

Modifications on the masks Some experiments involve to take an agent file saved on disk and perform operations on the parameters such as permuting the mask. This operation involves retraining a single agent. This operation can be even more parallelized than IMP which requires one step to be done before performing the next one. In other words, for this kind of experiments, even more compute power can be leveraged by splitting the load on more workers.

Evaluating the models We decided to save every step of IMP on disk. Thus evaluation can be performed at any moment. This operation involves spawning evaluator processes as in *synchronized APEX*. Each worker handles a file and generate a given number of episodes whose discounted reward is aggregated.

Chapter 5

Results

The topic of this work is to study the combination of deep reinforcement learning and the lottery ticket hypothesis. As mentioned in Chapter 3 - at the time of writing - two works have been published confirming the existence of the phenomenon in deep reinforcement learning (Yu et al. 2020, Vischer et al. 2021). In this chapter we reproduce some of their results and try to briefly extend in some areas. Firstly, in Section 5.1, we show we were able to find matching tickets within our experimental framework. Subsequently we discuss the importance of three hyper-parameters, namely the use of *late rewinding*, the *global pruning vs local pruning* and finally the number of iterative magnitude pruning steps (*IMP steps*). In the subsequent section we try to cast some light on properties exhibited by winning tickets found by iterative magnitude pruning. In Section 5.2 we will discuss how three aspects of the models evolve as sparsity increases. Firstly, in Section 5.2.1, we discuss the loss of expressivity of networks whose number of remaining parameters decreases. Then in Section 5.2.2, we discuss how the number of environment interactions to complete a task evolves as more parameters are removed from the models. Finally this section will end by looking at how iterative magnitude pruning keeps some layers less pruned as it goes forward. Subsequently in Section 5.3, we will surprisingly show that - in some ways - unstructured magnitude pruning recovers or get closer to structured pruning than what could maybe be expected. After that - in Section 5.4 - we will discuss pruning in the input layer and the ability of IMP to generate masks which are able to almost remove useless or redundant variables. As a last topic, we will give a few words about a variant of global magnitude pruning we call *pooled pruning* which could potentially be useful for deep reinforcement learning algorithms making use of several models (as Soft-Actor-Critic).

5.1 Confirmation of the existence

In this first section we confirm preceding results and show the existence of winning / matching tickets in the context of deep reinforcement learning. We conducted experiments on DDQN and SAC-discrete/continuous applied for classic control and pixel-based control. We followed the standard experiment introduced by [Frankle and Carbin, 2018], IMP is performed for 31 iterations and 20% of the weights with the lowest magnitude are removed at every iteration. We pruned all the layers globally and used late rewinding as explained in Section 4.5. As usually done, we compare the performance of tickets obtained through IMP with random subnetworks with matching sparsity. This will be the main criterion to assess whether tickets are winning / matching. Additionally, we aimed at reproducing results from [Vischer et al., 2021]. Hence, in order to disentangle the importance of the mask from the weights we also carried two experiments which apply transformation on the mask and weights obtained by IMP. The first one keeps the mask untouched but permute the remaining weights layerwise. This experiment is coined *keep mask permute weights* and will assess the importance of the weights on the overall effect. A last experiment will break the mask obtained by IMP through layerwise permutation (and permutation of the remaining weights). This experiment is coined *permute mask permute weights* and will assess the contribution of the mask. A summary of the four main experiments can be found in Table 5.1.

IMP variant	Retain mask	Retain weights	Retain layerwise pruning ratio	Retain layerwise distribution (initial weights)	Retain global pruning ratio
Vanilla	True	True	True	True	True
Keep mask permute weights	True	False	True	True	True
Permute mask permute weights	False	False	True	True	True
Random reinit	False	False	False	False	True

Table 5.1: Summary of the four basic variants of the IMP experiments. The vanilla experiment is the simple IMP. The three subsequent rows are variants changing the mask and/or the weights of the models obtained through IMP. Reproduction of a table from [Vischer et al., 2021].

Results

Results are depicted in Figure 5.1 and clearly suggest that iterative magnitude pruning is able to find sparse initialization which - when trained in isolation - are able to beat random subnetworks. Indeed, on every combination of environment and algorithm, there existed a fraction of remaining weights at which IMP tickets were able to surpass the performance of random subnetworks. On a few instances - especially on pixel-based control (top two rows) - subnetworks were even able to beat the performance of the unpruned networks. These are generally on environments whose reward is not maximized by the unpruned agent. Thus it suggests an improvement in terms of sample efficiency. This will be discussed in Section 5.2.2.

An important claim of [Vischer et al., 2021] was that a major part of the gap in performance between a random subnetwork and a winning ticket (the lottery effect) is to be related to the mask rather than the initial weights. While less affirmatively, our experiments also support this claim. Indeed, we often observe the loss in performance to be greater when the mask is broken (*permute mask permute weights*) compared to when the weights location of the remaining initial weights is lost (*keep mask permute weights*). However, we note the experiments showed in [Vischer et al., 2021] suggested a much larger importance of the mask than the one we observed. In Section 5.2.2 further experiments will look at sample efficiency and suggest that - for a winning ticket - the mask is indeed important. However, they will suggest the original initialization always improves the sample efficiency of a pruned agent.

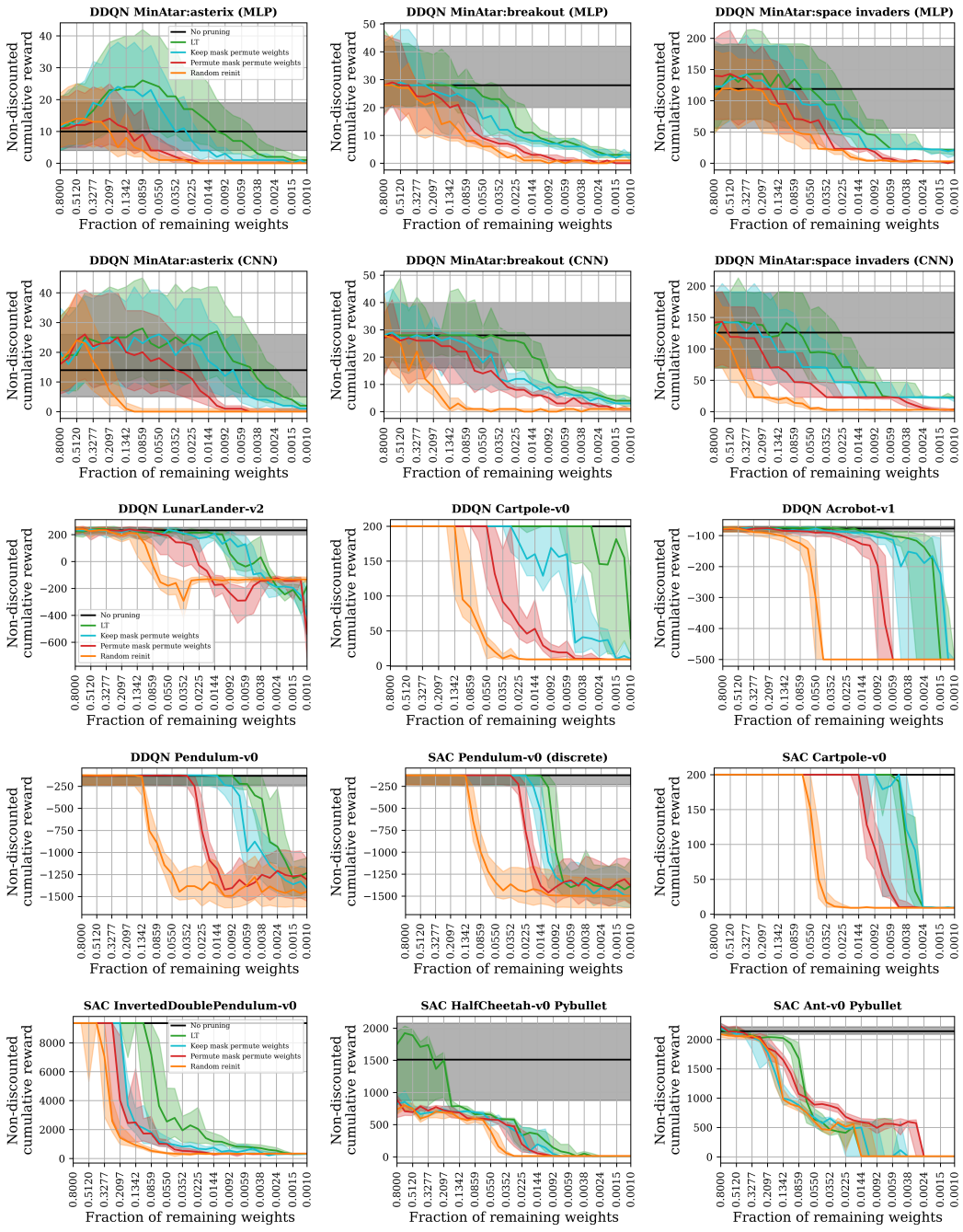

Figure 5.1: Description. Comparison of the non-discounted cumulative rewards as sparsity increases for IMP and 3 variants of the obtained tickets. Curves are medians estimated from 1000 episodes surrounded by 25% and 75% quantiles. Iterative magnitude pruning (IMP) removes a constant fraction of the remaining weights whose magnitude is the lowest. Then training is performed again. Process is repeated 31 times. Performance as sparsity increases along the IMP steps is displayed in green (LT). The performance of the full model (no pruning) is displayed in black. Three variants of the tickets obtained by IMP are generated. Firstly, the *keep mask permute weights* variant which keeps the mask and permutes the remaining weights in layerwise way (in cyan). Secondly, the *permute mask and permute weights* variant permutes the mask (layerwise) and re-assign the remaining weights (layerwise) (in red). Finally, there is a random reinit variant which is a random subnetwork with random re-initialization with commensurate sparsity of the given LT ticket the variant is based on. The x-axis is the fraction of remaining weights and is in log-scale. **Observations.** These charts suggest the existence of a lottery affect as a large gap between the winning ticket curves (green) and the random reinit curves (orange). Iterative magnitude pruning is able to find sparse initializations capable of working with less parameters. A large part of the lottery effect seems to be contained in the mask as keeping only the mask (cyan) keeps a good part of the overall effect.

5.1.1 Later rewinding: helps at best, never harms

Here we show that late rewinding can improve substantially the performance of winning tickets and when it not the case it does not degrade performance. As such, the following experiments are the reason we applied late rewinding systematically. Results are depicted in Figure 5.2, and as can be seen rewinding to later parameters helps to delay the collapse in performance. As such, it is possible to reach larger sparsities with an almost negligible additional cost.

Similar experiments were carried in [Yu et al., 2020] and [Vischer et al., 2021]. The former found late rewinding to be helpful especially for pixel-control tasks using CNN architectures. However, the last found late rewinding to be have negligible effect on the performance of winning tickets.

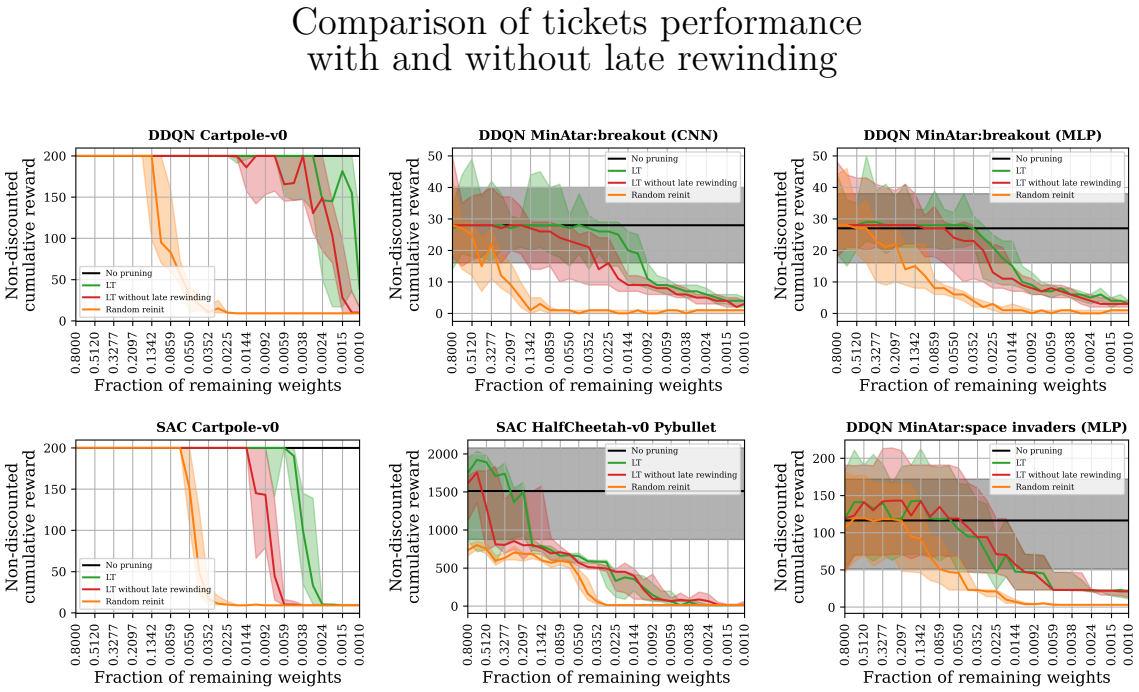

Figure 5.2: Description. Comparison of the performance of tickets whose weights are reset to their original initialization (LT without rewinding in red) and tickets whose weights are reset to a value obtained after a few training iterations (LT in green). Random reinit is a baseline and is made of a random subnetwork with random initialization. Pruning is performed globally. Performances are measured as non-discounted cumulative rewards. Curves are medians surrounded by 25% and 75 % quantiles estimated from 1000 episodes per random seed (usually 5). The x-axis is in log-scale. **Observations.** Using late-rewinding seems to be always beneficial or at least not harmful.

5.1.2 Global pruning vs layerwise pruning

Here we discuss another IMP hyperparameter whose value is often taken for granted. Indeed no previous experiments confirmed the necessity of performing pruning globally for DRL models. It is often assumed to be necessary for models having large discrepancy between layer sizes. It appears that it is the case for the models we used in this work and the models used commonly in value-based deep reinforcement learning. Indeed, it is usual to have networks whose input and/or output layers scale with the number of state variables / actions. It is also very frequent that these numbers are one or two orders of magnitude lower than the size of the hidden layers. Here we confirm what could already be expected: global pruning is necessary to obtain winning tickets which are able to perform well for large sparsities (better than random tickets). As can be seen in Figure 5.3, switching off global pruning turns out to make the collapse in performance happen sooner - sometimes much sooner (in terms of sparsity). It is especially relevant in the case of DDQN + breakout with a CNN architecture. It could suggest that the CNN input layer is very sensitive to pruning. The reason might be that it contains already fairly few parameters as can be seen in Appendix B.2. There is also the notable exception of the HalfCheetah-v0 environment with

SAC where global pruning did not seem to help. This result would deserve further inspection. As a conclusion, these experiments point toward a very favorable impact of global pruning. This is the reason we used this setting systematically throughout this work.

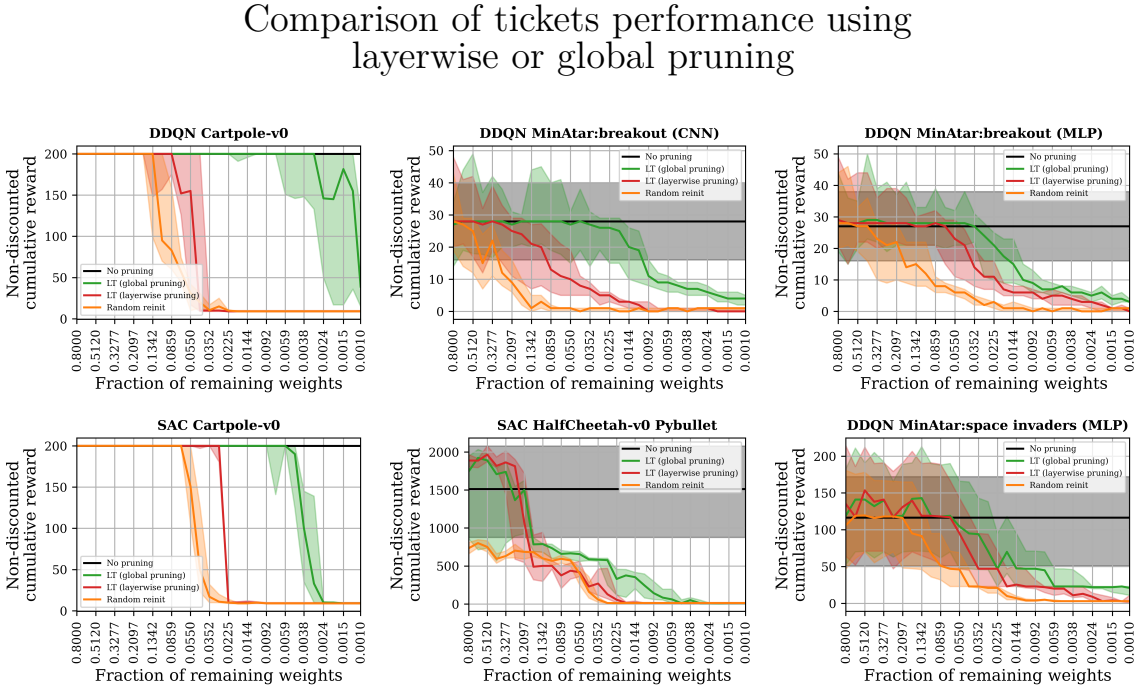

Figure 5.3: Description. Comparison of the performance of tickets whose weights are removed layerwise (same fraction for each layer) or globally (fraction of remaining weights can vary across different layers). Random reinit is a baseline and is made of a random subnetwork with random initialization. Parameters are rewound to their late resetting values. Performances are measured as non-discounted cumulative rewards. Curves are medians surrounded by 25% and 75 % quantiles estimated from 1000 episodes per random seed (usually 5). The x-axis is in log-scale. **Observations.** Experiments suggest that pruning globally may be very beneficial and give the ability to reach - sometimes - much larger sparsities. Within this experiments, using global pruning never harmed performance.

5.1.3 Influence of the number of pruning steps

Here we study the influence of a last important IMP hyperparameter: the number of pruning steps. As discussed in Section 3.2.1, performing the pruning in more than one step is often critical since the L1 magnitude heuristic can sometimes be misleading. In this work we always used 31 IMP steps since it is the standard in the literature. Here we provide a set of experiments where we picked the number of IMP steps in {5, 10, 20, 31}. In Figure 5.4, the tickets performance are displayed.

We recall that given a performance curve as the one for 31 steps, every point is not the performance of a ticket obtained after 31 pruning steps. These curves are the performances of tickets obtained sequentially from the first IMP step to the last.

From these experiments we observe that 31 steps seems always to be a good choice. Indeed, with 31 steps, IMP was often able to sustain larger sparsity before collapsing in performance. However, we observe that on some tasks with 10 or 20 steps IMP obtains tickets with performance on par compared to 31 steps. On several tasks, a large part of the lottery effect - the gap in performance between random subnetworks and winning tickets - could already be extracted with 5 or 10 IMP steps.

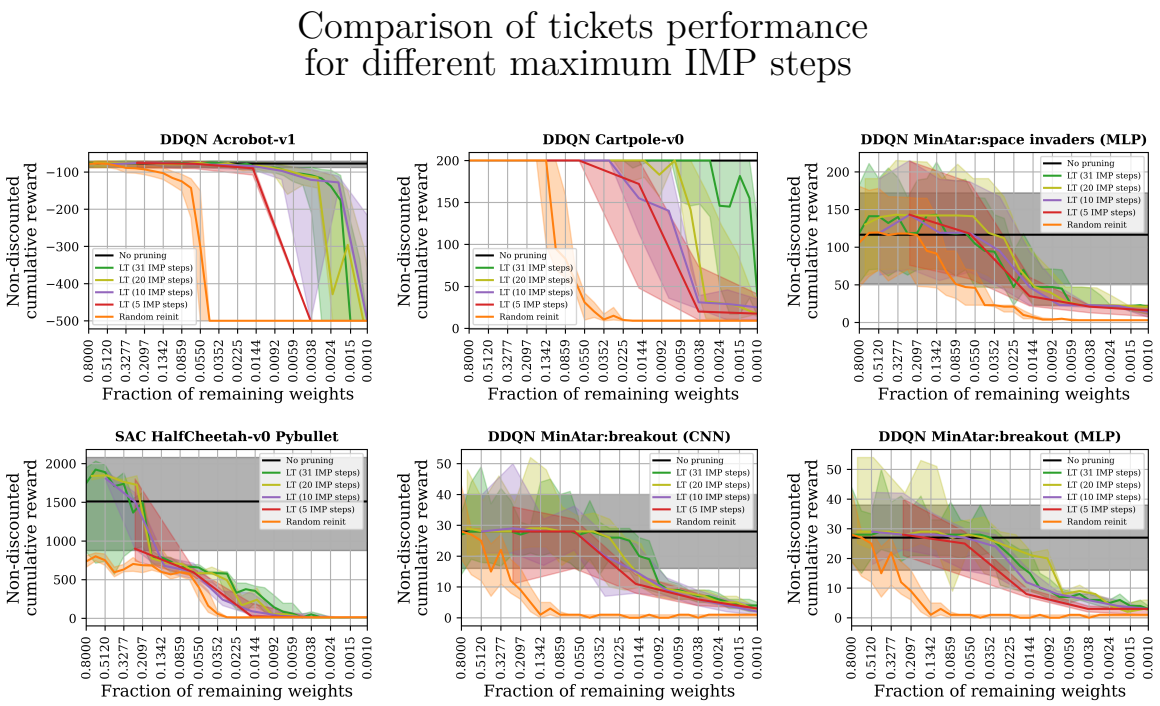

Figure 5.4: Description. Comparison of tickets performances obtained as iterative magnitude pruning progresses for different number of maximum iterations. Comparison of 5, 10, 20 and 31 IMP steps. Random reinit is a baseline and is made of a random subnetwork with random initialization. Pruning is performed globally and parameters are rewound to their late resetting values. Performances are measured as non-discounted cumulative rewards. Curves are medians surrounded by 25% and 75 % quantiles estimated from 1000 episodes per random seed (usually 5). The x-axis is in log-scale. **Observations.** Results suggest that using at least 20 IMP steps is a safe choice. On several tasks a large part of the lottery effect (compared to 31 IMP steps) could already be extracted using only 5 or 10 IMP steps.

Discussion Here we do not make claim on which value is best. Indeed these results suggest there might be a trade-off between the number of IMP steps and the final performance of the model. We note that by using at least 10 steps, we observed a large gap between the random subnetwork and the tickets. Since this is the criterion to establish whether the ticket is winning, we suggest this is the minimal working value for our experiments. Favouring safety, these results also comforted us to choose 31 IMP steps for the experiments throughout this work.

5.2 Evolution of the tickets

In this section we aim at understanding how the tickets behave as pruning goes by. This section will be divided into three parts. Firstly, we will discuss how the loss of parameters influence the expressivity of the network by looking at the embedding generated by the penultimate layer. In a second time we will observe how much individual layers are pruned as sparsity increases. This will inform us on the kind of strategy implicitly selected by iterative magnitude pruning. Finally we will discuss a very important aspect of reinforcement learning which is the *sample efficiency*. We will discuss how pruning agents affect their ability to master a task rapidly.

5.2.1 Embedding of the last layer: the s-rank

Motivations In this part we use a tool called the *effective rank* to try to understand how the number of parameters affects the ability of the networks to represent complex functions. The study of *effective rank* of the penultimate layer for deep reinforcement learning is motivated by [Kumar et al., 2020]. The author showed that value-based reinforcement learning tends to generate networks which behave as smaller *under-parametrized* networks. Here, we use their tool - the effective rank - and observations on under-parametrization to study how tickets progressively loose in expressivity and how it might affect performance. The effective rank of a matrix Φ is written $\text{srank}_\delta(\Phi)$ and is defined by $\text{srank}_\delta(\Phi) = \min\left\{k : \frac{\sum_{i=1}^{k}\sigma_i(\Phi)}{\sum_{i=1}^{d}\sigma_i(\Phi)} \geq 1-\delta\right\}$ where $\{\sigma_i\}$ are the singular values of the matrix Φ in decreasing order such that $\sigma_1 \geq ... \geq \sigma_d \geq 0$.

Let's assume we consider a Q-function network $Q_\theta(\,.\,|\,s)$ with N layers and for discrete actions such that $Q_\theta(\,.\,|\,s) = Q_\theta(s) \in \mathbb{R}^{1\times|\mathcal{A}|}$. Now, let's write the features given out by the penultimate layer as $\phi_{\theta_{1:N-1}}(s) \in \mathbb{R}^{1\times d}$. Hence, the vector of Q-values decomposes as $Q_\theta(s) = \phi_{\theta_{1:N-1}}(s)\,\theta_N$ with $\theta_N \in \mathbb{R}^{d\times|\mathcal{A}|}$ since the output is a simple linear transform from the last set of features as depicted in Figure 5.9.

The authors of [Kumar et al., 2020] suggest to consider Φ to be the matrix of stacked vectors $\phi_{\theta_{1:N-1}}(s)$ $\forall\, s \in \mathcal{S}$. Then this matrix of stacked vectors is in $\mathbb{R}^{|\mathcal{S}|\times d}$ such that - when it is multiplied by θ_N - the result is a Q-table, a matrix whose elements represent the Q-value of every combination of state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$.

With everything set, the authors claim $\text{srank}_\delta(\Phi)$ to relate to the number of *effective* unique components of the matrix Φ which form the basis for linearly approximating the Q-values. If $\text{srank}_\delta(\Phi)$ is close to d (the width of the penultimate layer) then the network is said to be able to map states to orthogonal vectors. If d is close to 0, the network maps states to a smaller subspace than what it is capable of. In other words, the s-rank relates to aliasing. It quantifies how different states are told apart. If this value is low the networks acts as a smaller one. It is said to behave as an *under-parametrized* network by [Kumar et al., 2020]. The authors showed empirically a strong association between under-parametrization and the poor performance observed for some agents. This observation was then supported empirically once more by [Ota et al., 2021].

Experimental setup We reproduced the setup from [Kumar et al., 2020] and used it to evaluate every ticket for every sparsity. Since the state space is potentially infinite, computing the singular values of Φ may be intractable. Hence, the s-rank is computed by sampling batch of states i.i.d from a replay buffer. This is motivated by [Yang et al., 2019] where the authors claimed if a large matrix with low-rank is sub-sampled the resulting matrices are likely to be low-rank as well. We acknowledge we looked away on that question and followed [Kumar et al., 2020]. The parameter δ was set to $\delta = 0.01$ as in the original paper.

For a given task, the s-ranks were computed from a same set of samples generated by the unpruned models. This is done in order to eliminate bias in the states visited since distribution may evolve as the sparsity increases (especially after performance collapse). The number of samples was 500 000 and was shared evenly between the random seeds (usually 5). For pixel-control the width of the penultimate layer was 512 and the batch-size 2048 (numbers from Kumar et al. 2020). Hence, the matrices Φ are 512×2048. For classic control the width of the penultimate layer was 256 and thus the batch-size was scaled-down to 1024. For ticket we drew 100 matrices Φ, compute their effective rank and aggregated the results. We provide results for both DDQN and SAC on classic control and pixel-based control. In

the case of Soft-Actor-Critic both the soft-q-network and actor can have their penultimate layer rank evaluated even though there is no evidence supporting the study of the latter.

Results A comparison of the sranks as sparsity increases for a large set of environments is provided in Figure 5.5. The sranks of winning tickets (LT) is compared to the values obtained by random subnetworks of equal sparsity. The values after training (solid lines) are also compared to the values before training (shaded lines). In the top half of the figure, a comparison of the ranks on 3 MinAtar environments with CNN and MLP input layer. The bottom half is a set of classic control environments for both DDQN and SAC. The vertical shaded lines are the performance collapse thresholds. These are the lowest estimated sparsities at which 10% of the lost in performance from the unpruned model to the most pruned model has been reached (10% being an arbitrary choice). In the case of IMP variants, the most pruned models are compared to models with 80% of parameters remaining. From these charts we make several observations.

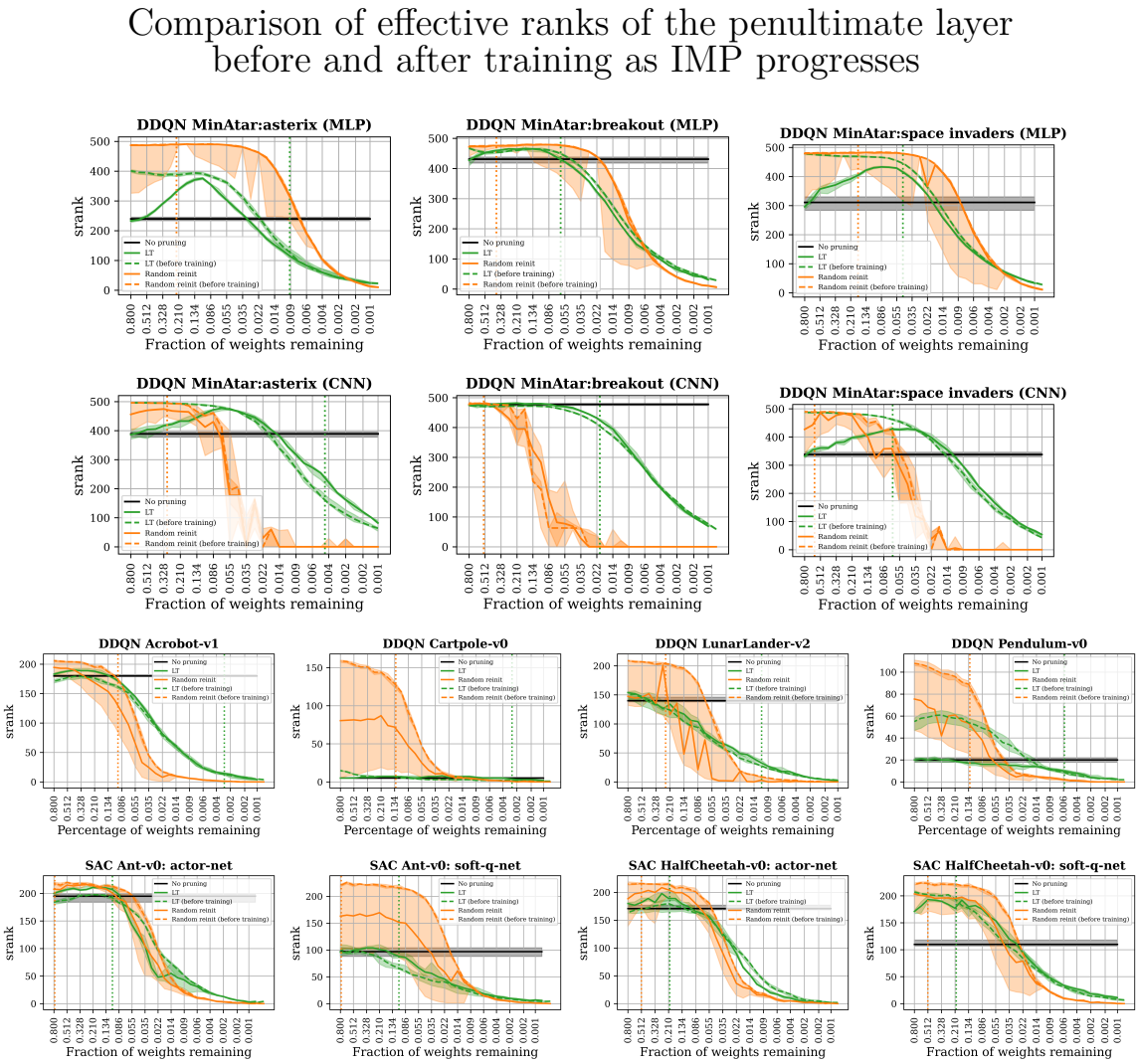

Figure 5.5: Description. Comparison of effective ranks of the penultimate layers output for tickets before and after training (in green). Baseline is a random subnetwork with random initialization. Effective ranks are computed by drawing 500 000 samples from the unpruned trained models and sampling batches of states whose size is 1024 for classic control and 2048 for pixel-based control. These states are passed through the networks and embeddings are aggregated as matrices whose effective ranks are computed. Curves are medians surrounded by 25% and 75% quantiles estimated from 100 repetitions of the previously mentioned experiments. **Observations.** Firstly, the winning tickets sranks drops for larger sparsities. Thus the last layer embedding of winning tickets seems more robust than random reinit counterparts. Secondly, winning tickets preserve low sranks on some low ranks environment (Cartpole-v0, Pendulum-v0). Finally, on some environments, the sranks increases suggesting a potential implicit training through pruning.

Firstly, for the MinAtar environments with DDQN and dense input layer (first row), the effective ranks of the random subnetworks are almost always larger than the winning tickets with same sparsities. However, the performance of the corresponding agents are lower and the performance collapse happens much sooner. This observation seems at first counter-intuitive and suggests - as could be expected - that the sranks alone cannot explain the discrepancy in performance between winning tickets and random subnetworks. Interestingly, the random subnetworks sranks does not evolve has training is performed (solid line merged with dashed line)

Secondly, on MinAtar environments we observe the sranks of the winning tickets to be increasing as IMP progresses on two out of the three environments. In the case of asterix, it seems to match well with the sparsities at which the performance of the agent increases or was the largest (as can be seen in Figure 5.1). It may suggest the networks are *undertrained* such that an increase in sranks - a more accurate embedding - leads to an increase in performance. In other words, if more training had been carried from the beginning, it is likely that the sranks as well as the performance would have been larger. This joins a general observation about winning tickets which suggests that pruning can be considered as a form of training in the sense is removes weights that would have been low anyway [Zhou et al., 2019]. In other words, when the increase in sranks and performance appears it implies there is room for improvement. This gap can be reduced by implicit training through magnitude pruning. In our opinion, this reasoning - while interesting - would need to be confirmed by more experiments.

Thirdly, training seems to almost always lower the srank. This matches with observations from [Kumar et al., 2020] where value-based DRL is known to exhibit lowering sranks networks as training goes on. However, unlike their results we did not observe this reduction in srank to always be associated with a reduction in performance. In other words, we observe that srank can decrease substantially across training while performing equally well at the task.

Fourthly, on the classic control tasks (third and fourth rows), the sranks of the winning tickets always seems to be lower than the values obtained by random subnetworks. It is especially prevalent when the random subnetwork has not yet reached the performance collapse threshold (vertical orange dashed bar). It suggests that - when both networks are considered to be equally good in terms of performance - the LT tickets provide a lower rank embedding of the state space. For example on DDQN Cartpole-v0 and DDQN Pendulum-v0, the winning tickets sranks were dramatically lower than their maximum possible values (256). It appears these are likely the tasks with the simplest state space and dynamics. They are thus likely to also have a simple Q-function. Especially, in the case of Cartpole-v0, were IMP seems to be able to obtain subnetworks which preserve low sranks for very large pruning rates. Further experiments available in Appendix C.1 suggest that the best variant to preserve low effective ranks is a combination of winning mask and winning weights.

Finally, on all classic control tasks (except Ant-v0) as well as pixel-control tasks with CNN input layer, there is a sparsity lower than the performance collapse (green vertical dashed line) above which the winning tickets sranks is larger than their random network counterparts. It suggests that the winning tickets are more robust for providing a *good* embedding of the state space as sparsity increases. Consequently, here we are possibly able to relate better performance with more robust srank. However, the srank has nothing to do with rewards. It is an estimate of state aliasing under some distribution over states (this distribution being samples generated by the unpruned models).

Discussion In this section we have experimented on the sranks of subnetworks. We compared winning/matching tickets with random subnetworks of equal sparsity. Our experiment support a possible implicit training induced by magnitude weight pruning. They also suggest the winning tickets are able to sustain low rank embeddings robustly for larger sparsities than random subnetworks. Another way to coin this observation could be to say that winning initializations may contain an inductive bias. This inductive bias is observed as the ability to alias states better than random subnetwork thus before training and as sparsity increases. In our opinion, the effective rank is an interesting tool which bypasses the notion of reward and depends only on state distribution and state aliasing.

5.2.2 Sample efficiency of the pruned agents

In this section we try to investigate whether winning tickets could be more sample efficient than the full model and if so, up to what sparsity. We will observe whether the same property holds for a random subnetwork (random mask + random reinit). Sample efficiency is to be understood as the ability of an RL algorithm to reach large rewards quickly. The efficiency is computed with respect to the number of environment interactions - the number of samples generated by the exploration. Here we will use a benchmark which is a variant of the Learning Area Curve (LCA) tailored for reinforcement learning by [Sokar et al., 2021]. The LCA is computed by going over the training curve. Let's Δ be the number of training steps and $R(t)$ the performance of the agent after t training steps. The LCA is then computed as

$$\text{LCA} \triangleq \frac{1}{\Delta}\int_0^{\Delta} R(t)\,dt = \frac{1}{\Delta}\sum_{t=0}^{\Delta} R(t) \tag{5.1}$$

More details on how the LCA scores were computed practically can be found in Appendix C.2

Results In Figure 5.6 the LCA curves for several environments are depicted. Theses charts show the LCA scores obtained by the four usual variants (winning ticket, keep mask permute weights, permute mask permute weights, random mask random weights) as pruning rate increases. From these charts we firstly observe the winnings tickets are most often the models which learn the fastest. Indeed, the LCA score seems to always decrease as more changes are performed on the winning tickets (alteration on the mask and/or weights). Here again, we denote that a large part of what makes winning tickets special (*the lottery effect*) seems to be related to the mask. However, as before we find it to be as such in a lower extent than [Vischer et al., 2021]. For these reasons, the LCA curves appear very similar to the standard performance curves showed in Figure 5.1.

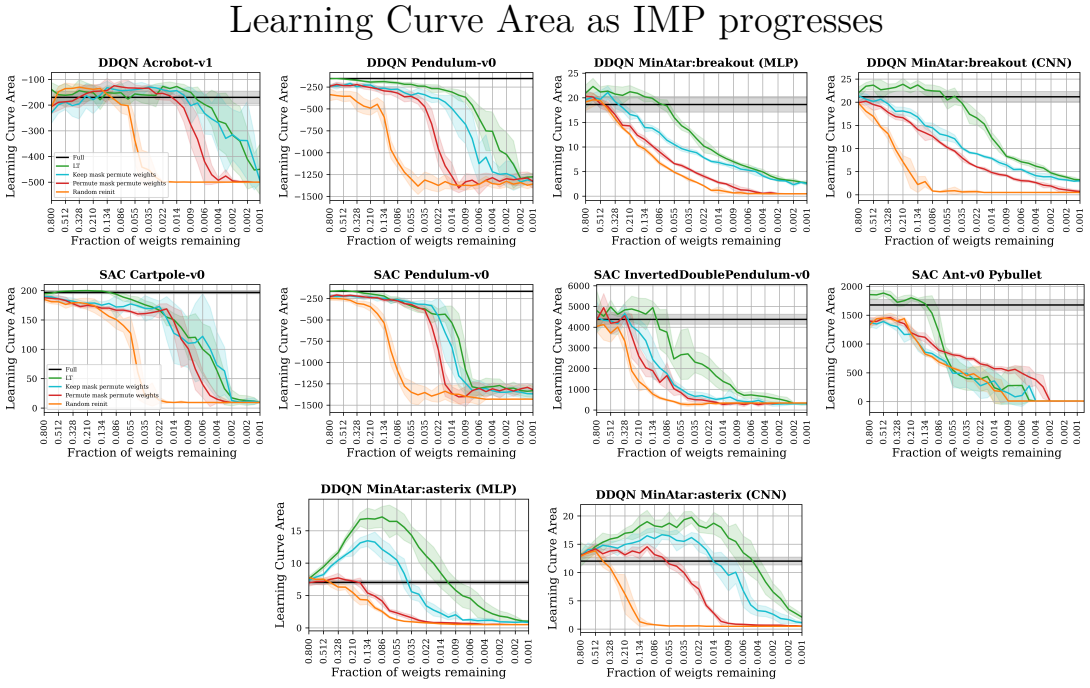

Figure 5.6: Description. LCA scores as pruning rate increases for DDQN and SAC on several tasks. More details about the curves calculation can be found in Appendix C.2. Green curve is the winning ticket from subnetworks obtained through IMP. Orange curve is for random network (random sub-network + random-reinit). Blue curve is for winning ticket whose mask is kept fixed and whose remaining weights are permuted layerwise. Red curve permutes mask and remaining weights from winning tickets. **Observations.** The winning tickets always have the largest sample efficiency measured in LCA score. For some environments, winning tickets are more sample efficient than the unpruned model.

Secondly, on some tasks, the winning tickets are able to learn faster than the unpruned model. This property also holds for some of the other IMP variants but to a much lower extent. On MinAtar-breakout, the winning tickets is able to obtain LCA scores up to 10-15% larger than the unpruned model. In the case of MinAtar-Asterix - as could be expected since the performance dramatically increased - the LCA score can increase by up to 100%. For SAC on the Ant environment this effect grows up to almost 30%.

Discussion We observe the increase in learning speed (measure in LCA score) to be the largest for the least trivial tasks with the largest state spaces (i.e Ant-v0, MinAtar environments). Importantly, these are also the environments whose maximum rewards were not reached even by the unpruned models (no performance saturation). Joining observations on the effective ranks from previous section, we suggest this increase in learning speed might be due to the implicit training induced by weight pruning.

Consequently, we suggest there might exist an *intermediary regime* during which implicit training helps to learn faster. This regime may last until the expressivity of the network is damaged to an extent preventing it to work properly.

As a conclusion, we may answer this section question by the negative. Within our experiments we did not observe a random subnetworks to be able to learn the task faster.

5.2.3 Layer-wise fraction of remaining weights

In this section we look into the amount of remaining weights per layer as IMP progresses. Here we mainly confirm global pruning is taken advantage of. In Figure 5.7, the different layers fraction of remaining weights are displayed as IMP increase the model sparsity (black line). These are averaged values computed from the different random seeds (usually 5). From these charts, it appears that IMP does exploit the global setting by pruning either one or both the input and output layer (layer #1 and #3 from Figure 5.9) much less than the hidden layer (layer #2). In other words, IMP removes a large part of the weights to be pruned on the middle layer (which is always the largest in our experiments). This discrepancy in fraction of remaining weights is often as large as one order of magnitude. On many tasks the gap even increased with IMP steps. Interestingly, in the case of SAC we may note a few particularities. For the soft-Q-networks, the two networks seem to have very close layer-wise pruning ratio. Furthermore, in the case of SAC, the input layers - in contrast with DDQN - often have layer-wise pruning ratio close or lower than the global pruning ratio of the model.

Discussion This result are observations. They suggest that - when it can - IMP prunes the different layers differently. However, it is not straightforward to understand why it happens to such an extent. Obviously, it is due to a great part of the p% lowest magnitude weights to be located in the middle layer. While seemingly fortunate, we do not know the cause of this. In our opinion, it does not imply much regarding the existence or not of winning tickets with equal pruning ratio for every layer. In this work, we observed the winning tickets obtained through IMP often required global pruning.

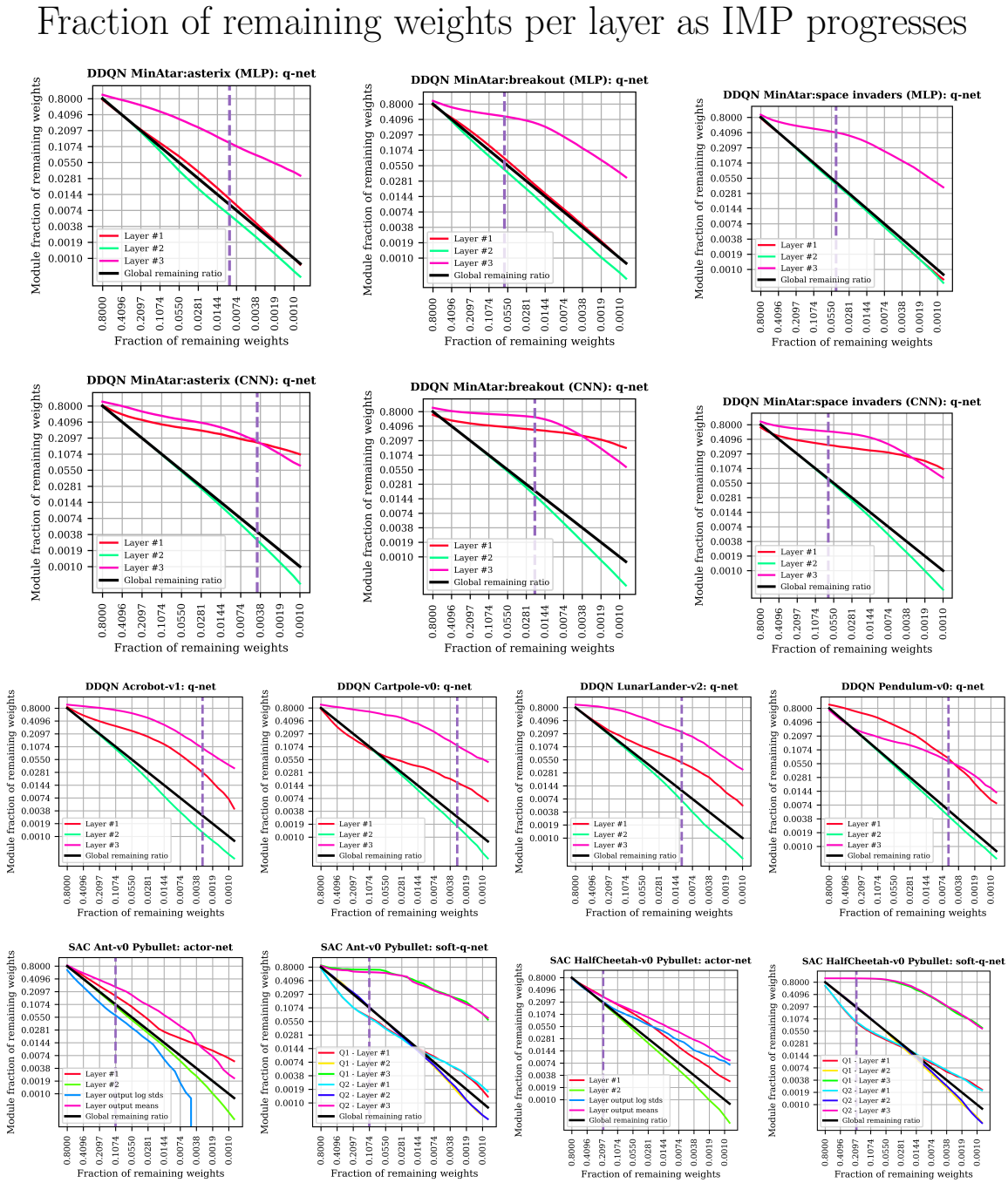

Figure 5.7: Description. Fraction of remaining weights per layer as IMP progresses. Curves are averaged estimated from the random seeds (usually 5). Both axis are log-scale. Purple dashed line is the performance collapse threshold. It is the pruning ratio at which 10% of the loss in performance from the unpruned model to the most pruned model (31 IMP steps) has been reached. It estimates the sparsity after which the model performance falls rapidly. **Observations.** Iterative magnitude pruning prunes the layers at very different rate. On some environments the last layer or the last and first layer are pruned much less than the middle layer which contains a large part of the parameters. Hence, IMP takes advantage of global pruning.

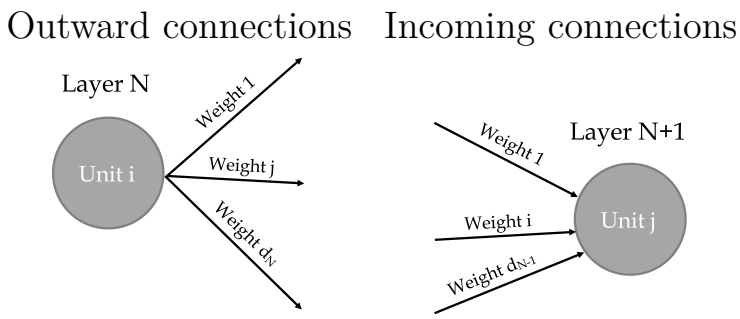

Figure 5.8: Depiction of the idea of incoming and outward connections in a neural network with linear layers. **(Left.)** The outward connections correspond to the weights multiplying the output of a unit (after the activation). There is one outward connection per unit in the following layer. **(Right.)** The incoming connections - for a given unit j - correspond to weights multiplying the outputs of the previous layer and summed before the activation.

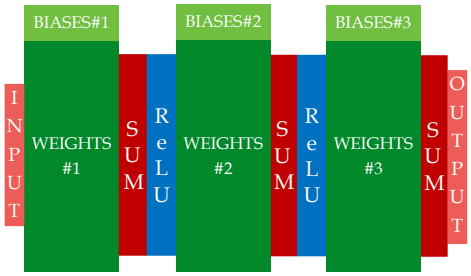

Figure 5.9: Schematic of a 3 layers neural network with ReLU activations. An input is processed twice with activations which creates an embedding that is linearly combined in the layer 3 to make the output. Most of the neural networks discussed in this work are variant of this schematic. Q-networks (DDQN, SAC-discrete) and actor network (SAC discrete) correspond exactly to this scheme.

5.3 Structure in unstructured pruning

In this section we argue that IMP is able to generate masks which exhibit more structure than their permuted counterparts. In other words, we found winning masks not to be *typical* random subnetworks (which are likely to be found randomly). In a previous section we discussed structured pruning and unstructured pruning. Iterative magnitude pruning is an algorithm which removes weights in an unstructured way such that connections are removed one by one. Here we show that - very often - IMP leans towards structured pruning in the sense that it tends to remove whole units and keep the other units more untouched. We think this discovery - while maybe not surprising - is insightful regarding how IMP spends what we call its *connection budget* - the remaining number of connection it must allocate given some global pruning rate and the mask of the previous iteration.

As an evidence we propose - for a given layer - to look at how many remaining incoming connections a unit receives. Then, we proceed by computing an empirical distribution of that quantity. More formally, we look at a layer N with weight matrix $\theta_{W_N} \in \mathbb{R}^{d_{N-1} \times d_N}$. This weight matrix distributes linearly the features from layer $N-1$ to the units of layer N. Every column of that matrix is associated with a unit of the layer N. We write the mask applied on matrix θ_{W_N} as $M_{\theta_{W_N}} \in \{0, 1\}^{d_{N-1} \times d_N}$. For a given unit $i \in \{1, ..., d_N\}$, the number of incoming connections is thus $R^C_{N,i} \triangleq \sum_{j=1}^{d_{N-1}} M_{\theta_{W_N}}[j\,;\,i]$. In this section we compute an empirical distribution of that quantity. In order to do so we use all the columns of a layer as well and group those obtained by all the random seeds (usually 5 in total). Furthermore, structured pruning can also be recovered by removing all the weights going outward from a unit. Hence, removing unit j can be done by setting row j of the weight mask to zeros. The corresponding quantity is written as $R^R_{N,j} \triangleq \sum_{i=1}^{d_{N+1}} M_{\theta_{W_{N+1}}}[j\,;\,i]$. In the idea of incoming and outward connections is depicted in Figure 5.8.

Interpretability For some layers R^R and R^C can be interpreted.

Input layer In the case of the input layer, R^C tells us how much the next layer tends to use the whole input space when R^R indicates how much a given input variable is used by the next layer. This last quantity will be studied separately in Section 5.4.

Output layer When the layer is the output layer, R^C quantifies how many embeddings from the penultimate layer an output relies on. Finally, R^R is indicative of how many times an embedding is used to compute the output.

Results The quantities we just introduced can be studied for any layer. As depicted in Figure 5.9, most of our networks only contain three set of weights and biases. Thus we can quickly visualize the empirical distributions for the whole network. It appears that structure may be found in the incoming connections (R^C) or the outgoing ones (R^R) or both. We first study the structure in the incoming connections of the first and second layers (input layer and penultimate layer in our case). Here we showcase three examples. The first and second have low-dimension input spaces (Pendulum-v0: 3 state variables and LunarLander-v0: 8 input variables). The third example has a much larger input space with 400 dimensions (MinAtar-breakout). Results for the second layer can be seen in Figure 5.10. The same experiments for the input layer are provided in Appendix C.2. From the results on the second layer we provide a few important observations.

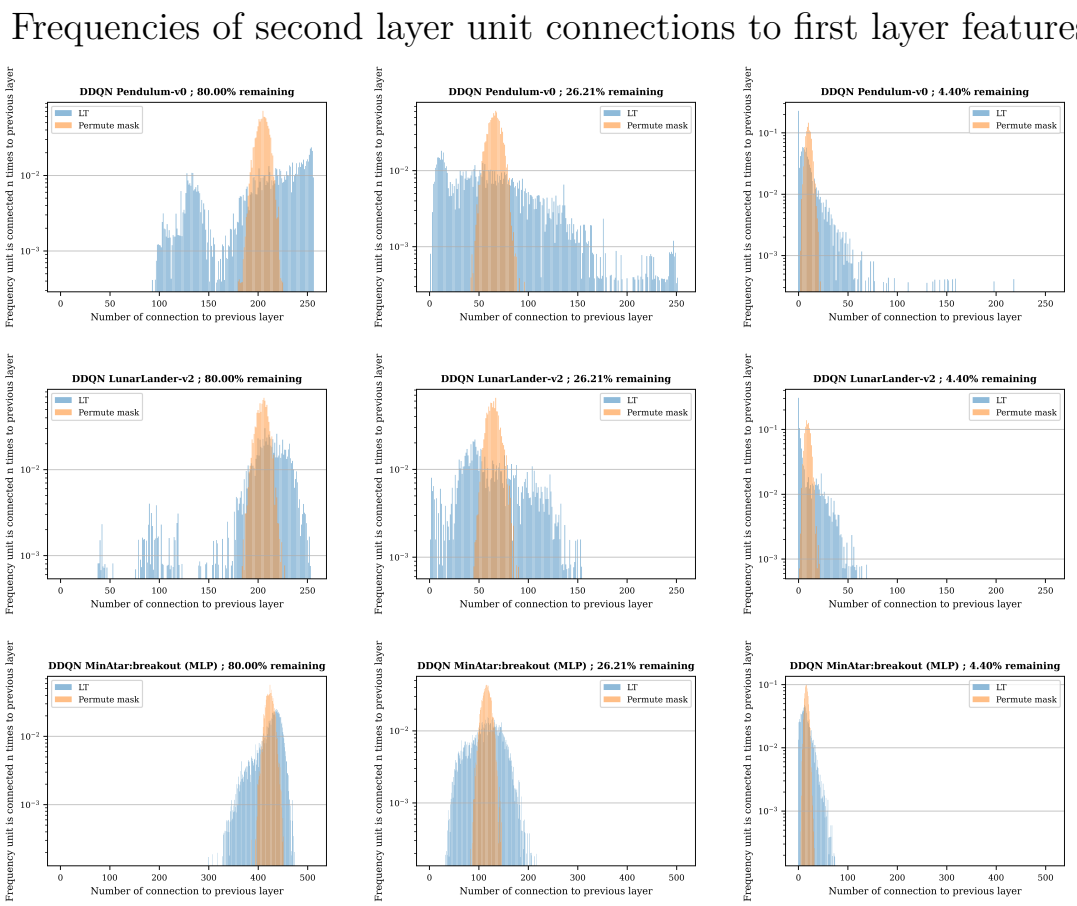

Figure 5.10: Description. Empirical frequencies of number of times a unit from second layer is connected to a feature outputed by the first layer. Comparison of aggregated frequencies computed from matching tickets (blue) with random permutation of the input mask (orange). Frequencies estimated by grouping the input masks from all the random seeds (usually 5). Y-axis is in log-scale. **Observations.** Mask found through Iterative Magnitude Pruning do not behave as truly random masks. They do split unit between low activity (few incoming connections) and high activity (many incoming connections) much more than a random mask would. For some sparsity many units are even made completely inactive (0 incoming connections). These observations are indicative of the strategy implicitly adopted by IMP.

Firstly, the random permutation of the masks (orange bars) follows a binomial distribution which has a quick decay and is unimodal. Obviously, this is normal and to be expected. The matching masks (blue bars) appears to be very different for every sparsity and environment. While the permutation bars fall quickly, the winning tickets leads to a distribution which is more heavy-tailed.

Secondly, matching tickets clearly splits activation between low activity (few incoming connections) and high activity (many incoming connections). For example, in the case of LunarLander-v2, units with few active connections emerge immediately (first pruning step with 80% remaining). With as much as 26.21% of the connection remaining (globally), IMP already removed some units when this does not happen for the random permutation even after more than 95% of the weights were removed. This observation is even more remarkable when one realize the further on the right of the x-axis, the larger the number of connections are involved. In other words, an increasing portion of the *active weights* is spent on units already having some active connections.

Finally, these observations hold for the input layer as well. However, they do so to a much lower extent such that the following discussion will only apply for results we got on the penultimate layer.

Discussion In this section we have shown that IMP extracts masks with larger groups of active connections per unit than what a random subnetwork would. We have also observed some units to be completely discarded since all their incoming connections got pruned. Consequently, we may say that IMP winning tickets indeed seem to lean towards structured pruning. We think it is an interesting observation. One might have expected winning masks to be especially lucky random networks subnetworks such that they do not exhibit any unlikely focus on some units or grouping as we observed. In our opinion, these observations are informative on the *strategy* applied by IMP to spend its *connection budget*.

Furthermore, we think these results may be related to previous experiments on the effective ranks of the penultimate layer. In the case of low ranks embedding environments such as Cartpole-v0 or Pendulum-v0, we observed winning tickets to be able to preserve low rank. This might be explained by a rapid discard of many units and the focus of remaining weights on a few units. Hence, dimensions of the penultimate layer embedding are simply inactive which preserves low rank. However, experiments provided in C.1 suggest that neither the mask nor the weights seems to explain alone the whole preservation of low rank. Our experiments lack low sranks environments and the two we showcased are not very conclusive. Thus we suggest future work could study that question in more details. Hence from this observed IMP strategy for spending weights and the previous reasoning, we ask the following question which will stay open: *Can Iterative Magnitude Pruning be considered as an implicit regularization on the sranks of the penultimate layer ?*

In a second time, we also observed sranks of winning tickets to decay more robustly than their random counterparts. Again, this might be explained by the difference in strategies. On the one hand, winning tickets group remaining connections such that removing some of them is less likely to make units inactive. On the other hand, remaining weights of random subnetworks are stretched thin such that removing a few may remove whole units and thus lower the effective rank more rapidly.

5.3.1 The output mask

In this section we follow from the previous observations and carry a similar study on the mask applied on the last layer which we call the *output mask*. This time will discuss both type of structure mentioned before. More specifically we will look at both the rows (outgoing connection) and the columns (incoming connections) of the last layer mask.

Incoming connections

In a first time we look at the connection from the perspective of the output units (incoming connections). In the previous section we observed IMP tended to favour units relying on larger number of features from the previous layer. In the case of the output layer, we observe it does not happen the same way.

In our experiments we chose to consider networks whose outputs correspond to action since we think it is more interesting. Hence, we looked at the actor network of SAC-continuous and the Q-network of DDQN.

Results Here the fraction of remaining weights connected per action is computed. The results are depicted for several algorithm-environments in Figure 5.11. As can be seen, it appears that actions tend to be treated equally in terms of *connection budget*. When it does not it is only after the performance collapse (pink vertical bar) where the performance of the agent starts to fall. In other words, IMP spreads the remaining connections evenly between the actions. It contrasts with the observations from the previous section. However, as discussed next, it does not mean the output mask does not exhibit structure.

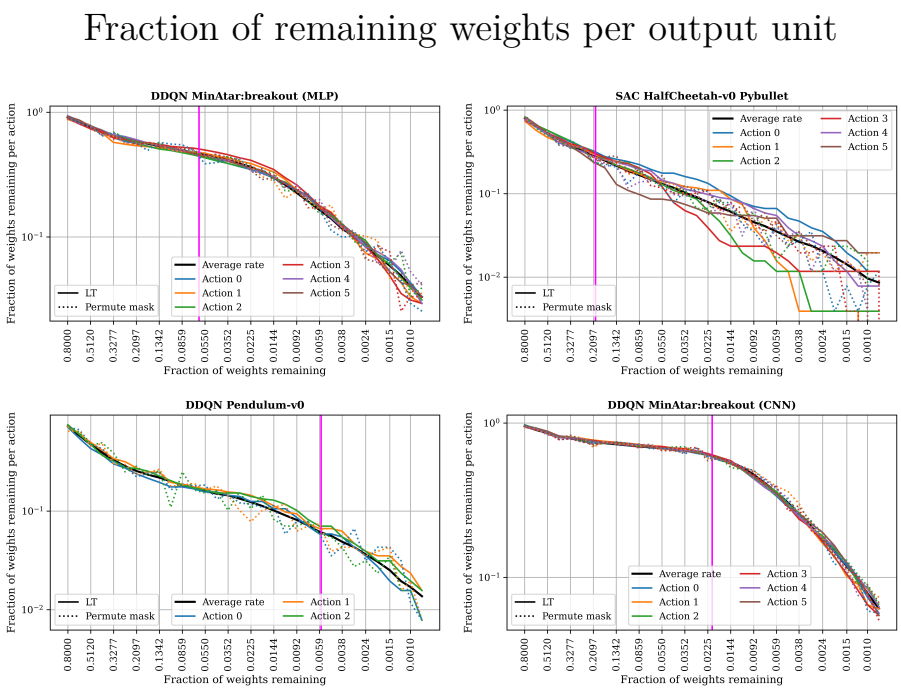

Figure 5.11: Fraction of weights connected per output as global pruning rate increases. Log scale on vertical axis. Log scale on horizontal axis. Solid line is the fraction of remaining connection per action on IMP tickets. Dotted line is the same quantity from tickets whose output mask is permuted. Curves are computed from 5 seeds. The output masks are merged and resampled using bootstraping. Pink vertical line is the performance collapse threshold compute as the pruning rate at which 10% of the performance decrease between unpruned model and most pruned model is reached. **Top left.** Actor network for SAC-continuous on InvertedDoublePendulum-v0 **Top right.** Actor network for SAC-continuous on HalfCheetah-v0 **Bottom left.** Q-network for DDQN on Pendulum-v0 **Bottom right.** Q-network for DDQN with CNN on MinAtar-breakout. **Observations.** Experiments suggest the mask of the output / last layer does not favour any action by giving more remaining connections. When a discrepancy appears, it is after the performance collapse threshold such that it is not significant.

Outward connections

In a second time we view the weight matrix and the mask of the last layer from the perspective of the penultimate layer (outward connections). In a previous section we asked ourselves the following question: *How likely is a unit to rely on n features from the previous layer ?* This time we ask the question the other way around: *What is the probability a feature from the penultimate layer is used n times by the output layer ?* On top of this we propose to consider only the features which have been used at least once. We called these the *active features*. Hence the final question we try to answer is *Knowing a feature is used at least once, how likely is it to be used n times by the output layer ?* As before we compute an empirical estimator by pooling all the masks from the different random seeds (usually 5).

Results Results for DDQN and SAC on two environments each can be found in Figure 5.12. The charts for the version which does not require the features to be active can be found in C.4.2. The first two rows are for DDQN Q-network. The last two rows are for the log std head of the gaussian actor network from SAC-continuous. Quite interestingly, the observation differs from one algorithm to the other. In the case of DDQN, the output tends to be much more structured than in the case of SAC-continous. For Pendulum-v0, IMP generated masks which tended to use features from the penultimate layer 3 times while a random permuted mask would almost not do so. In the case of SAC-continuous we observe a tendency to favour multiple usage of penultimate layer features but in a much lower extent. In the case of the mean head of the gaussian actor network, the effect is not observed at all (C.3). These observations are interesting because they may imply that - to some extent - the larger the number of outward connections the larger the weight magnitudes after training. Indeed, the only reason a group of weights can be kept systematically is that every time training ends, the magnitude of its weights do not belong to the $p\%$ lowest magnitude weights.

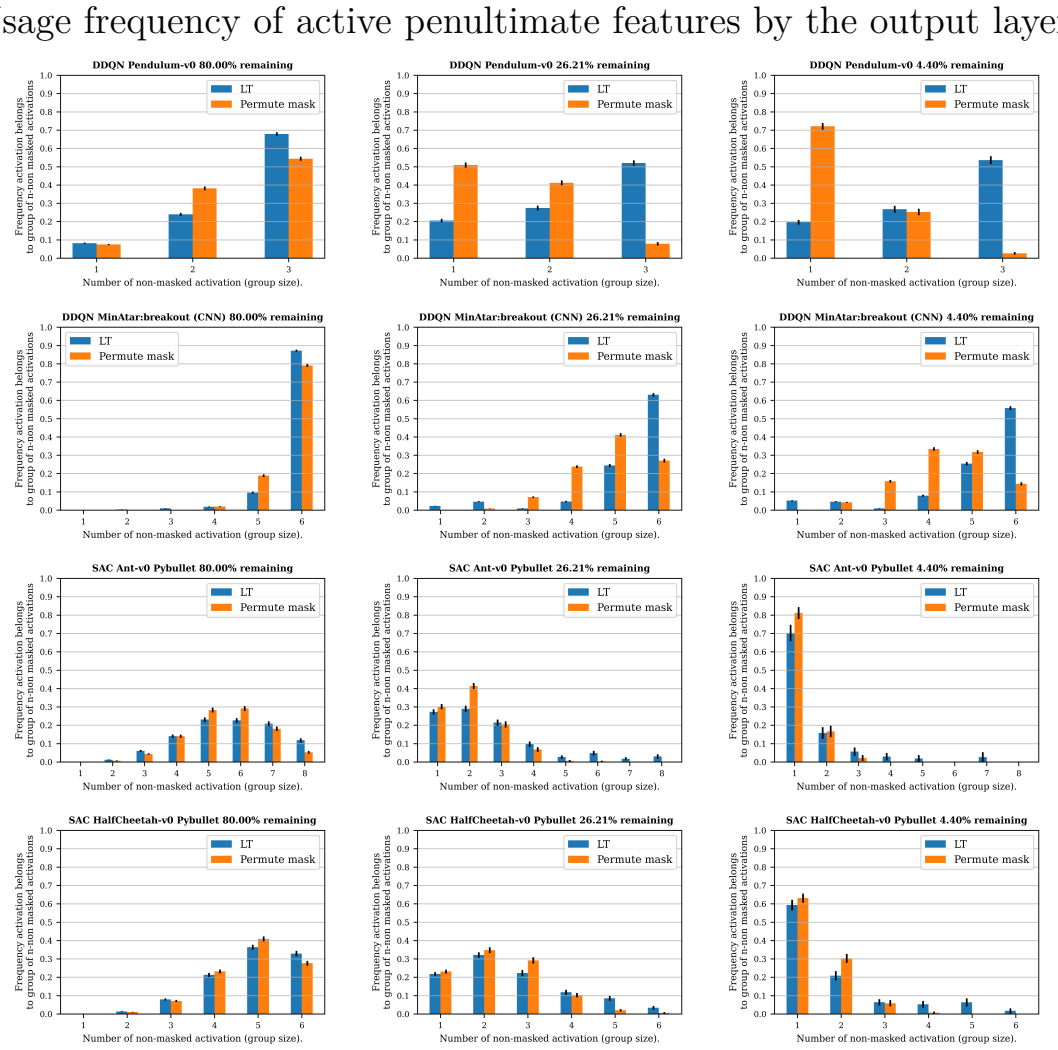

Figure 5.12: Description. Empirical frequencies for the number of times an active feature from the penultimate layer is used by the output layer. Histograms computed by pooling the masks obtained by the different random seeds and applying resampling (250 redraws). Bars are the mean frequencies across the samples. Error bars are ± 1 standard deviations. **Observations.** Output masks found through IMP quickly splits features of the penultimate layer between highly active (many usage) and less active units. Especially on DDQN, features of the penultimate layer - when they are used at least once - are used many more times than they would by a random mask. This is indicative of the strategy implicitly adopted by IMP.

Conclusion

In this section we studied the how connections tended to spent in the output mask. Our experiments suggest actions are considered equally in terms of connection budget. In other words, IMP spends a similar number of remaining weights per action. However, instead of spreading the remaining connections such that every feature from the penultimate layer is used evenly, IMP favours *strong features* which are used multiple times. This joins our observation on the penultimate layer in a previous section. Thus the reasoning on the effective rank and its robustness may hold as well. It is especially relevant in the case of Q-networks whose outputs can be seen as a Q-table sampled for some states and actions. It is known that some environments exhibit Q-table (sampled in the non-tabular scenario) which have lower ranks. For such environments, enforcing low rank in the Q-networks (sampled) Q-tables was shown to increase learning speed in [Yang et al., 2019]. From this knowledge and observations, we ask the following questions for future work: *For environments with low effective rank (sampled) Q-tables, is IMP able to generate tickets which preserve such low rank ? If yes, is it able to do so more robustly than a random subnetwork ?*

5.4 The input mask

In this section we study the so-called *input mask* which is the mask applied on the first layer processing the inputs. It is the layer #1 in Figure 5.9. As insightfully said in [Vischer et al., 2021], the input mask can be viewed as a set of googles which are applied on the inputs. The agent will perceive the environment through these googles which may completely remove some input variables. It has already been shown in [Vischer et al., 2021] that IMP is able to remove *redundant variables* - whose dynamics is redundant because it can be deduced from others. In this section we confirm their observations and try to go further in some areas.

5.4.1 Variable selection

Here we confirm the observations from [Vischer et al., 2021] on the ability of IMP to remove variables or at least a very large fraction of the weights connected to them thus while keeping agent performance almost unharmed. In Figure 5.13 the fraction of remaining weights connected for each input variable is depicted as global model pruning increases. Details on how these charts were computed can be found in Appendix C.5.1. The vertical bar indicates at what sparsity a given decrease in agent performance has be undergone. This decrease is computed as 10% of the gap between the performance of the unpruned model and the performance of the most pruned model. This threshold is designed to be an estimate of *when* the performance starts to decrease rapidly.

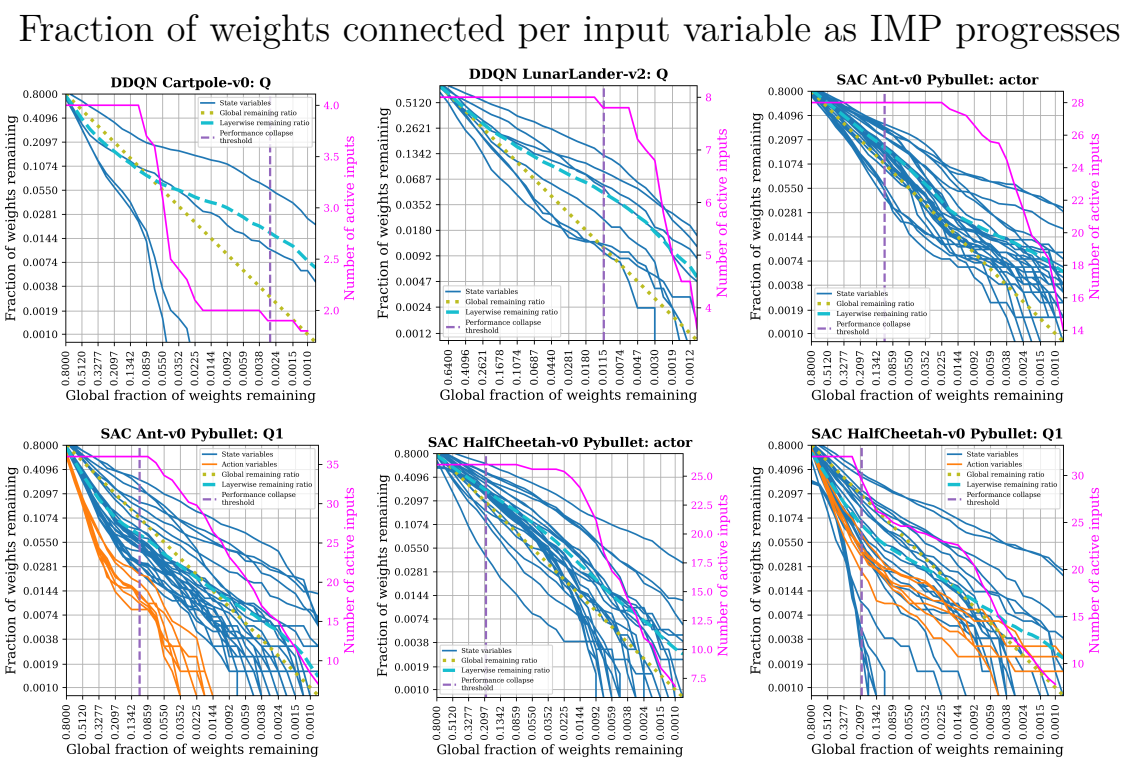

Figure 5.13: Description. Fraction of remaining weights connected per input as the global fraction of weights remaining decreases. Results for the Q-network of DDQN, the actor network of SAC and the soft-Q-network of SAC. Green dotted line is the global remaining ratio computed on the whole model. Layerwise remaining ratio is the fraction of weights remaining in the input layer. Purple vertical bar indicates at what sparsity a given decrease in agent performance has be undergone. This decrease is computed as 10% of the gap between the performance of the unpruned model and the performance of the most pruned model. Pink curve and right y-axis relates to the number of active inputs. An active input is an input having at least one weight connected. The curve (in pink) is a mean estimated from the different random seeds (usually 5). Left y-axis and x-axis are both in logscale. **Observations.** As IMP progresses and sparsity increases the input variables are treated very unequally such that the weights connected to some variables are pruned much more heavily than for other variables.

As can be seen, every chart suggests IMP prunes weights connected to some variables much more heavily than others. In the the case of DDQN Cartpole-V0, 2 of the 4 input variables are completely removed before performance collapse. Such that the final agent can perform the task with less variables. In the case of SAC, variables are often completely removed only after performance collapse. However, the differences between variables in remaining fraction of weights is even more dramatic. Interestingly, in the case of SAC for continuous action spaces (i.e Ant-v0, HalfCheetah-v0), weights connected to the actions seems to be pruned more heavily than states.

5.4.2 Ability to remove useless variables

In this section we show that IMP is able to remove useless variables quite effectively. In order to show this ability, we added to the normal state variables a combination of variables which are - by construction - useless. These are constant variables (all equal to 0) and random variables whose value is resampled at every environment iteration in $[-1, 1]$. We tested two settings. One adds 5 constant variables and 5 random variables (low noise scenario) and the second adds 100 of each (large noise scenario). Here we study the ability of the model to remove these useless variables as well as how harmful these were on training performance and final ticket performance. In order to account for the additional number of parameters in the input layer in the case where 200 variables are added, we display the curves with the number of remaining weights as x-axis instead of the usual fraction of remaining parameters.

Results Results are depicted in Figure 5.14 and 5.15. From these charts we observe that for both DDQN and SAC, IMP was effectively able to remove most of the useless variables before the performance collapse. In other words using IMP, we were able - for some configuration - to find sparse agents which removed the useless variables and performed well at the task. Hence, IMP may help to recover from the addition of some amount of useless variables.

Interestingly, the constant variables ($= 0$) seemed harder to remove compared to the uniformly sampled random variables. IMP on DDQN seems slightly more effective than on SAC at removing useless variables especially when compared to the actor networks as provided in C.5. We also note that the impact of adding useless variables on sample efficiency and final performance of the tickets is not clear.

Discussion We observed IMP to be effective at removing useless variables. However, it had sometimes a cost in both sample efficiency and final performance. If this cost is acceptable, IMP could be used as a tool to discard variables when the state space of some environment is poorly understood. Iterative magnitude pruning could be applied either on the full network or only on the input layer. Then the input winning mask with our without the winning weights could be used alone and be applied as a filter for a proper new learning of the task.

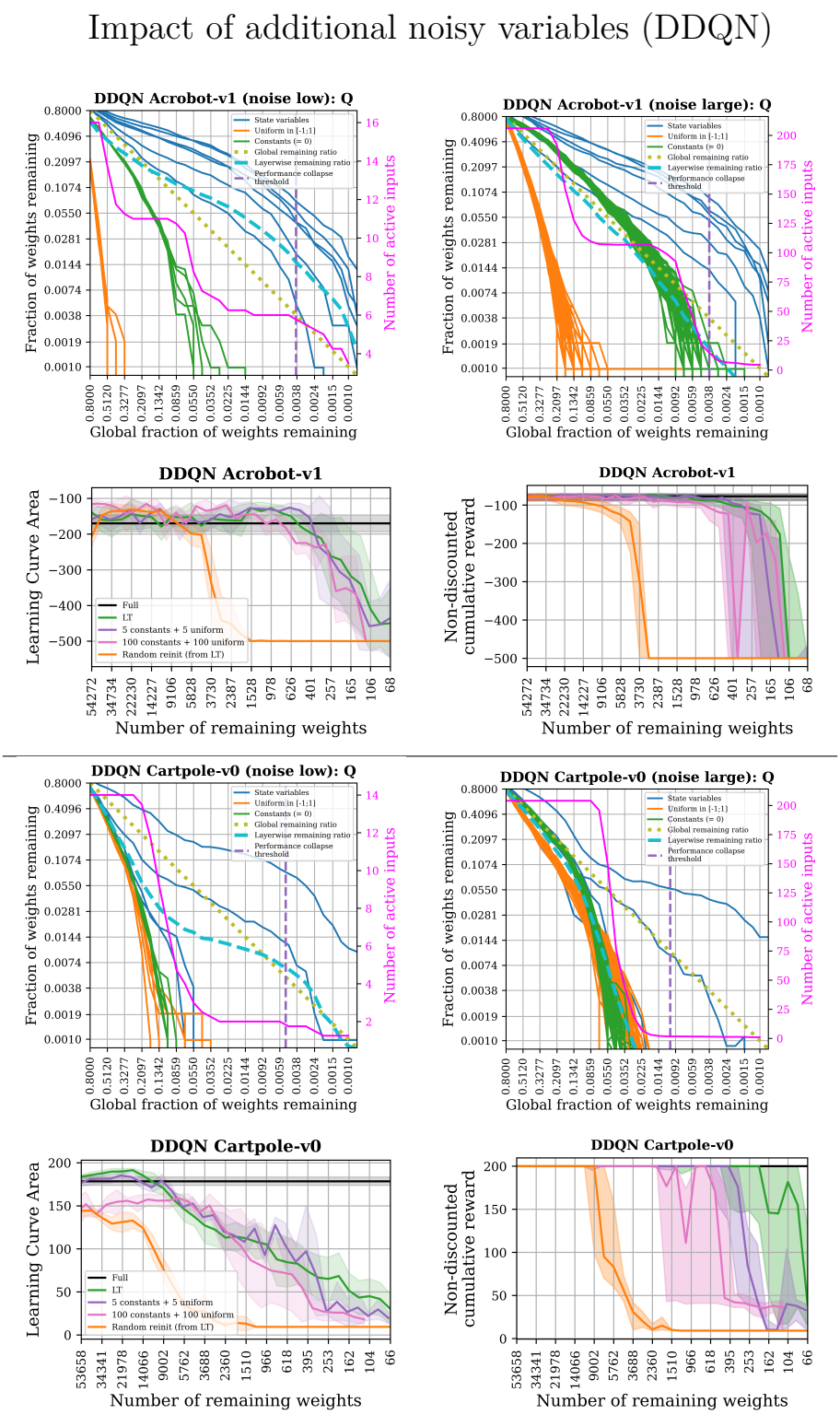

Figure 5.14: Experiments with DDQN and IMP on environments whose state space is added useless noisy variables. Two scenarios: the low noise scenario (5 constants variables = 0 + 5 variables uniformly sampled in $[-1\,;\,1]$) ; the large noise scenario (same as low noise but 100 new variables for each). **Top charts.** Acrobot-v1, **Bottom charts.** Cartpole-v0. For each group the top charts are the fraction of remaining weights connected per input (in the first layer). Pink curve and right y-axis relates to the number of active inputs. An active input is an input having at least one weight connected. The curve (in pink) is a mean estimated from the different random seeds (usually 5). The two bottom charts are the Learning Curve Areas and tickets evaluations. Comparison of winning tickets (LT in green), winning tickets in low noise scenario (in purple), winning tickets in large noise scenario (in pink) and random subnetwork (random subnetwork + random initialization, in orange). **Observations.** Iterative magnitude pruning is able to effectively remove completely almost every useless variable. It does so before collapsing in performance such that the resulting agent use only useful variables and performs well.

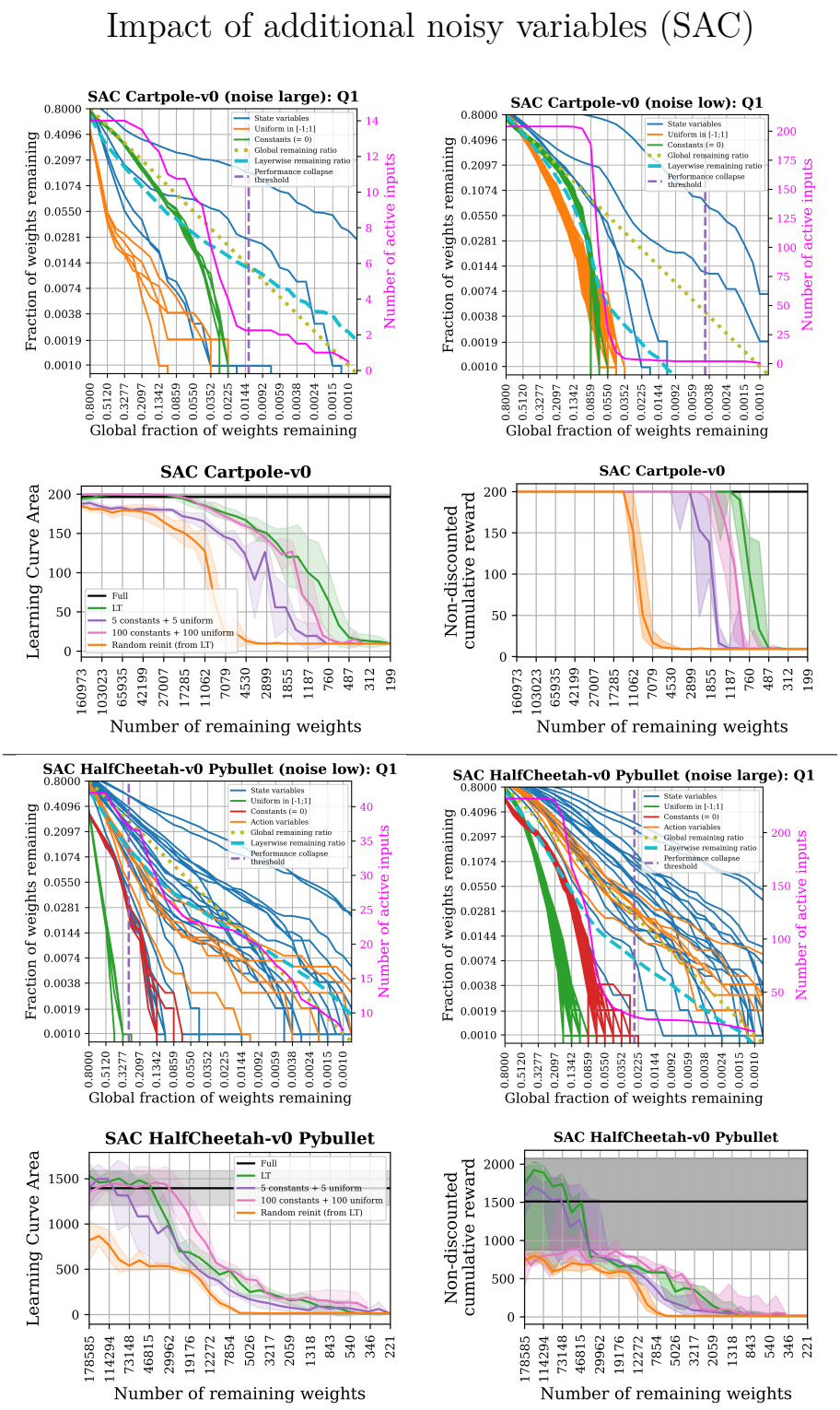

Figure 5.15: Experiments with SAC and IMP on environments whose state space is added useless noisy variables. Two scenarios: the low noise scenario (5 constants variables = 0 + 5 variables uniformly sampled in $[-1\,;\,1]$) ; the large noise scenario (same as low noise but 100 new variables for each). **Top charts.** Cartpole-v0, **Bottom charts.** HalfCheetah-v0 (Pybullet). For each group the top charts are the fraction of remaining weights connected per input (in the first layer). Pink curve and right y-axis relates to the number of active inputs. An active input is an input having at least one weight connected. The curve (in pink) is a mean estimated from the different random seeds (usually 5). The two bottom charts are the Learning Curve Areas and tickets evaluations. Comparison of winning tickets (LT in green), winning tickets in low noise scenario (in purple), winning tickets in large noise scenario (in pink) and random subnetwork (random subnetwork + random initialization, in orange). **Observations.** Iterative magnitude pruning is able to effectively remove completely almost every useless variable. It does so before collapsing in performance such that the resulting agent use only useful variables and performs well.

5.4.3 Recovers when the input is linearly transformed

In this section we investigate a scenario introduced in [Vischer et al., 2021]. It involves to linearly entangle the state space of the environment through a random matrix multiplication. Let's have $s \in \mathcal{S}$ with $\mathcal{S} = \mathbb{R}^{1 \times d_{\mathcal{S}}}$. Let's consider a random matrix B drawn uniformly in $[-1\,;\,1]^{d_{\mathcal{S}} \times d_E}$ with d_E the number of dimensions of the *entangled state space* ${\mathcal{S}'}_B = \{s' \,|\, s' = s\,B,\ s \in \mathcal{S}\}$. We experimented on two scenarios. The first one keeps the dimensions of the original state space ($d_E = d_{\mathcal{S}}$) and the second increases the dimensionality to 100 ($d_E = 100$). As in the previous section we observe how IMP will remove or not some of the new variables. We will also look at how harmful these changes are on the final performance of the agent and its sample efficiency. Once more, in order to account for the larger number of parameters in the input layer, results are provided with regards to the number of remaining weights instead of the fraction of remaining weights.

Results The charts are available in Figure 5.16 and 5.17. From these charts we make several observations.

Firstly, linear entangling prevents IMP from reducing feature usage as much as it could without state space transformation. Except in the case of Cartpole-V0, IMP is not able to find a subset of variables whose size is commensurate with the original state space before the performance collapse threshold (purple dashed line).

Nevertheless, we still observe a very large discrepancy in the number of weights connected for each input. Even though IMP is not able to completely remove features it still spends a large part of the *connection budget* on a few input variables indicating that some new features may be more useful than others.

Regarding sample efficiency and final performance, linearly entangling to 100 variables seems to be never harmful and quite the opposite. In the case of HalfCheetah-v0, the new state space lead to much more stable performance across the random seeds as well as a substantial delay in the performance collapse. However - in this same case - linearly mapping to a state space of same number of variables ($d_E = d_{\mathcal{S}}$) made IMP unable to perform better than a random subnetwork.

Discussion It seems on the one hand that keeping the dimensions ($d_E = d_{\mathcal{S}}$) can both improve or decrease the performance of the tickets. On the other hand, increasing the dimensionality to 100 new variables ($d_E = 100$) never harmed sample efficiency and final performance. It most often improved either one or both. Consequently, our observations agree with [Vischer et al., 2021] even though they did not correct for the addition of parameters in the input layer and used a much larger entangled state space ($d_E = 1000$).

One might wonder *why* we observe this phenomenon. We humbly suggest a possible explanation. Linearly entangling seems to slow down IMP into removing input variables. We thus suggest linear entangling preserves the flow of environment information for larger sparsties. This might be because, there is no subset of new variables which suffices or because the increase in dimensionality influences the weight distribution in the input layer (as discussed in Section 3.2.1). However, it seems orthogonal initialization (the scheme we used) does not scale differently weights for larger layers.

It appears that the linear transformation $s' = s\,B$ is equivalent to the addition of a linear layer between the input s and the rest of the network. This linear layer is held constant and is not pruned. As such, some might consider it as an additional set of parameters which has to be accounted for. Here we considered this change of variable as being part of the environment.

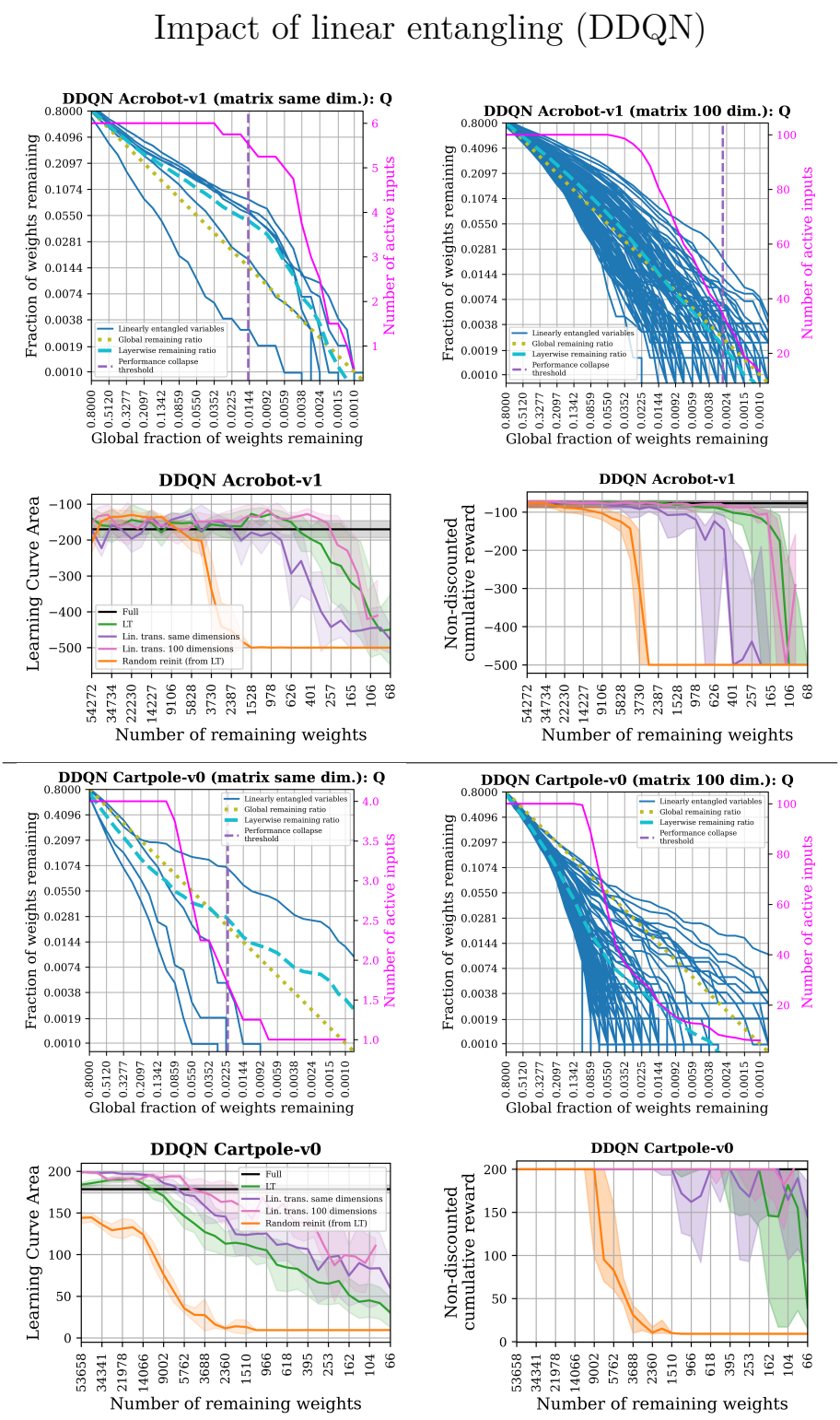

Figure 5.16: Experiments with DDQN and IMP on environments whose state space is linearly entangled by a random linear transformation. An original state vector is multiplied by a random matrix (drawn once) whose elements are sampled uniformly in $[-1\,;\,1]$. Two scenarios: the same dim. scenario where the original dimensions of the state space are preserved ; a large **Top charts.** Acrobot-v1, **Bottom charts.** Cartpole-v0. For each group the top charts are the fraction of remaining weights connected per input (in the first layer). The two bottom charts are the Learning Curve Areas and tickets evaluations. Comparison of winning tickets (LT in green), winning tickets in low noise scenario (in purple), winning tickets in large noise scenario (in pink) and random subnetwork (random subnetwork + random initialization, in orange). **Observations.** Entangling to a new state space of same dimension ($d_E = d_S$) has an unclear effect. However, increasing the dimensionality to 100 $d_E = 100$ does improve sample efficiency and final performance of the tickets (delays the collapse).

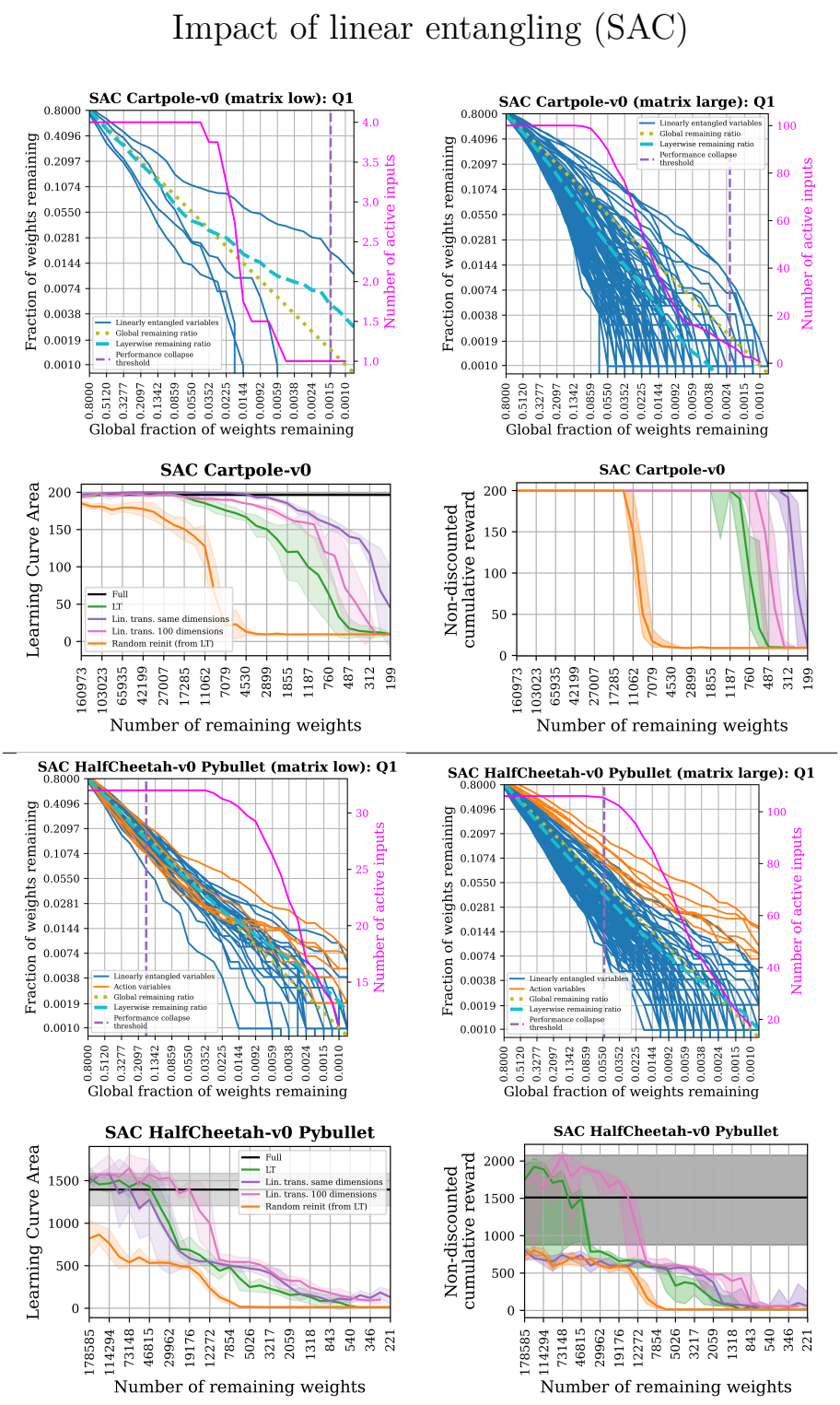

Figure 5.17: Experiments with SAC and IMP on environments whose state space is linearly entangled by a random linear transformation. An original state vector is multiplied by a random matrix (drawn once) whose elements are sampled uniformly in $[-1\,;\,1]$. Two scenarios: the same dim. scenario where the original dimensions of the state space are preserved ; a large **Top charts.** Cartpole-v0, **Bottom charts.** HalfCheetah-v0 (Pybullet). For each group the top charts are the fraction of remaining weights connected per input (in the first layer). Pink curve and right y-axis relates to the number of active inputs. An active input is an input having at least one weight connected. The curve (in pink) is a mean estimated from the different random seeds (usually 5). The two bottom charts are the Learning Curve Areas and tickets evaluations. Comparison of winning tickets (LT in green), winning tickets in low noise scenario (in purple), winning tickets in large noise scenario (in pink) and random subnetwork (random subnetwork + random initialization, in orange). **Observations.** Entangling to a new state space of same dimension ($d_E = d_S$) has an unclear effect. However, increasing the dimensionality to 100 $d_E = 100$ does improve sample efficiency and final performance of the tickets (delays the collapse).

5.4.4 Discussion: Using the input mask as variable filter

The previous sections have clearly shown that IMP could generate sparse input layers which can sometimes discard a great part of the input space. This ability could be especially useful when a large amount of variables is provided by the environment. Indeed, some variables could be useless or highly redundant. Removing variables could lead to faster processing (if fast implementations of sparse unstructured neural networks are available) and even a cheaper / faster sensoring of the environments (since some variables are completely discarded). Reducing the size of the *active* state space could also greatly reduce the memory footprint of replay buffer techniques. Now these observations were made it is natural to wonder whether sparse input layers could be transferred alone and be used for training in isolation. We also wonder whether the input layer could viably be pruned alone. In this scenario, the fraction of weights connected per state variable could maybe be considered as some measure of feature importance. We think these are interesting questions to pursue but they are outside the scope of this work.

5.5 IMP variant for soft-actor critic

As discussed earlier, throughout this work the pruning was always carried globally on a per model basis. However, in the case of SAC, there are three models whose parameters are pruned (DDQN only has one): the first and second soft-Q-networks and the actor network. In this section we propose to extend the idea of global pruning. We experiment pruning at a larger scale by *pooling* all the models parameters. It assumes the L1 magnitude heuristic can be generalized to several models which is not obvious. Indeed, the soft-Q-networks and the actor - while related by construction - optimizes different losses and may exhibit parameter distributions which vary. In this section, we provide a set of experiments showing that extending the L1 heuristic to several models seems reasonable in the case of soft actor critic. These experiments suggest a slight improvement in the tickets performance in general. To the best of our knowledge, it is the first time such a pooling of parameters from models with different losses and objectives is performed in the context of the LTH or at least in the context of the LTH and deep reinforcement learning.

Results Results are provided in Figure 5.18. As can be seen on all but one environment (Cartpole-v0), pooled pruning does not harm the perform nor does it decrease the maximum pruning the models can sustain. Looking closely, pooled pruning improves slightly the performance and clearly does in the case of InvertedDoublePendulum-v0.

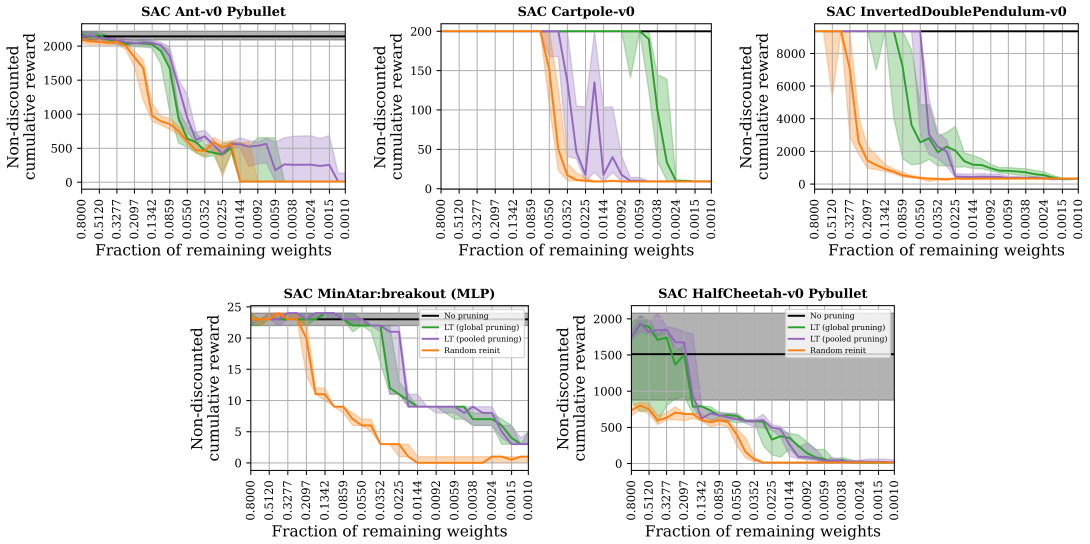

Figure 5.18: Description. Results for the *pooled* variant of IMP. In this variant, the p% lowest magnitude parameters are removed from the collection of parameters from all the models pooled together. Non-discounted cumulative rewards as sparsity increases for SAC on several environments. Comparison of random subnetwork (random mask + random weights) with same sparsity for every model (in orange), global pruning IMP (in green) and pooled pruning (in purple). Curves are median with 25% and 75 % quantiles estimated from 1000 episodes per random seed (usually 5). **Observations.** On four out of five, pooled pruning is able to keep or slightly increase the performance of tickets obtained with global pruning.

Discussion Here we introduced a variant of global pruning we called *pooled pruning*. It extends the space of possible masks by allowing for different sparsity per model. As can be seen in Appendix C.6, pooled pruning is exploited. It seems the actor networks tends to be slightly more pruned than the soft-Q-network. Even though comparing weight magnitude from models whose objective loss and purpose differ may not be a sound heuristic, experiments suggest it is at least able to compete with the classical global pruning. One of the main interests of pooled pruning could arise in situations where model sizes varies. It is actually the same argument for global pruning compared to layerwise pruning. This kind of scenario appears for more complex architectures such as *Soft-Actor-Critic with deterministic auto-encoders* introduced in [Yarats et al., 2019]. In their proposed architecture the relative size of models can vary dramatically. We argue that with these architectures, the use of pooled pruning could be critical. This is a scenario that - we think - deserves to be be experimented.

Chapter 6

Conclusion

This worked studied the combination of the Lottery Ticket Hypothesis and value-based Deep Reinforcement Learning. We have confirmed on two standards algorithms (DDQN and SAC) the existence of winning tickets found by Iterative Magnitude Pruning. Subsequently, we studied properties exhibited by winning tickets such as their sample efficiency and their ability to represent complex functions. Then, we introduced IMP as a method for variable selection and potentially feature importance. Lastly, we studied winning mask and found out that they tended to be more structured than what could have been expected.

6.1 Preliminaries

The first step of this work has been to provide an introduction to DDQN and SAC from first principles. We provided the different steps starting from the tabular scenario and progressively lifted or added constraints. We provided the losses as well as full-pseudo codes for both algorithms. It is also - to the best of our knowledge - the first time SAC-continuous and SAC-discrete were introduced concurrently.

The second step was to provide a formal definition of the Lottery Ticket Hypothesis. We provided a literature review of that notion which has undergone several changes and controversies following its introduction. We re-introduced the notion of mask criterion and mask operation. Finally, the Iterative Magnitude Pruning algorithm was uncovered along with a discussion on its most important hyperparameters.

Thirdly, we discussed the state of knowledge regarding the combination of LTH and DRL. We provided a summary of the experimental setups applied by the preceding works in that scenario. We also discussed some challenges which might prevent the finding of winning tickets.

Finally, we introduced a scalable experimental framework for value-based DRL which is able to withstand effectively the very large computational burden implied by the IMP algorithm. This method was designed to enforce reproducibility while leveraging a large amount of hardware.

6.2 Contributions

The first contribution of this work was the third confirmation of the existence of winning tickets in the context of DRL. It is also the first time winning tickets are found for the Soft-Actor-Critic algorithm.

The second contribution of this work was the confirmation of the importance and usefulness of late rewinding and global pruning. We also suggest a lower number of IMP steps than the standard value (31) can be applied for DDQN and SAC while still exhibiting winning tickets.

The third contribution of this work has been regarding the ability of winning tickets to represent complex function and to perform what we call *beneficial state aliasing*. Our results suggest that winning tickets

found through IMP are able to robustly preserve low rank state embeddings / descriptions before the last layer of the network.

Fourthly, we studied sample efficiency and suggested that under some sparsity pruned agents are able to learn tasks faster than their unpruned counterparts.

Subsequently, we provided some evidence suggesting that there is more structure in unstructured IMP pruning than what could be expected. We suggest IMP spends the *connection budget* in a way that tends to favour groups of connection pointing from or to a restricted set of units. We related this observation with the ability to robustly preserve low ranks in the penultimate layer.

Then we provided evidence that IMP is able to remove a large part of noisy useless variables as well as linearly entangled variables. We showed that linear entangling - a simple and fast mechanism modifying the input space - is able to stabilize tickets performance and reduce the minimum number of parameters required to complete a task.

Finally, we introduced a new variant of IMP that we call *pooled pruning* which could potentially be very beneficial for algorithms involving several networks of different sizes.

6.3 Limitations and open questions

As any research work, this study has a number of limitations and leaves many questions opened.

Firstly, in our opinion, this study could be improved with a better hyper-parameter exploration. The learning rates is known to be very impact-full for the uncovering of winning tickets. In the case of tasks whose rewards were not maximized, it might be interesting to see what would happen with more training time. The *sample regime*, induced by the number of exploration processes may be an important hyperparameter impacting the data distribution used for learning. Hence, we suggest to reproduce our experiments with the same performance target but different number of exploration processes.

Only two network sizes were tested without a proper study of the importance of the number of parameters of the original network. We suggest to study the possibility of increasing layer widths (which is known to be a sound idea) up to 2048-4096 units. The idea to explore would be to *go higher to go lower* - starting from a more overparametrized network to possibly reach a lower number of remaining parameters at the end.

Iterative Magnitude Pruning requires to prune weights after - ideally - convergence. In our experiments we often did not so but still observed a valuable *lottery effect*. Thus we wonder whether one could reduce the training time per IMP iteration but increase the number of IMP steps. This could introduce a trade-off between number of IMP steps and training time per iteration. A sweet-spot, potentially better than the standard *31 steps and training to convergence* could maybe be found.

This study is limited to two types of network architectures which were either sequence of dense layers or a CNN layer followed by a sequence of dense layers. To the best of our knowledge no previous experiments has been carried out on different architectures applied to DRL. Firstly, we suggest to study the combination of LTH and Recurrent Neural Networks for Partially Observable MDPs. Secondly, many state of the art architectures in value-based DRL use auto-encoders with reconstruction or contrastive losses to re-encode the state space (Yarats et al. 2019, Srinivas et al. 2020). We suggest to consider the pruning of these types of architectures especially since they tend to grow in size which may limit their wide-spread applicability.

Furthermore, this last type of architectures may be seen as a bridge between model-free methods and model-based methods. The model-based framework offers a large zoo of new algorithms and models whose pruning could be investigated.

In a quest to further reduce the memory footprint, we suggest to study the combination of IMP and weights quantization. Our results and previous ones suggest a larger importance of the mask over the winning weights. As such, maybe quantization could be applied before of after training. It could potentially work with less damage on performance than an unpruned model.

A question left open in this work was about the ability of IMP to preserve robustly preserve low ranks Q-tables. We suggest to investigate that question. However we think this might require to work with environments having larger number of actions than the ones we used. Indeed, to some extent all the environments we used in this work had low rank Q-tables (maximum of 6 since at most 6 actions).

Finally, an important component of Iterative Magnitude Pruning is the L1 heuristic. However, as suggested in [Zhou et al., 2019], other mask criterion may work just as well or even better. Hence, we think it might be interesting to investigate - in the context of DRL - whether another heuristic could allow to reach larger sparsities or have impact on the tickets properties (remaining weights per layer, structure, sranks).

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. and Zheng, X. [2015]. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.
URL: *https://www.tensorflow.org/*

Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z. D. and Blundell, C. [2020]. Agent57: Outperforming the atari human benchmark, *CoRR* **abs/2003.13350**.
URL: *https://arxiv.org/abs/2003.13350*

Bellemare, M. G., Naddaf, Y., Veness, J. and Bowling, M. [2012]. The arcade learning environment: An evaluation platform for general agents, *CoRR* **abs/1207.4708**.
URL: *http://arxiv.org/abs/1207.4708*

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W. [2016]. Openai gym, *CoRR* **abs/1606.01540**.
URL: *http://arxiv.org/abs/1606.01540*

Christodoulou, P. [2019]. Soft actor-critic for discrete action settings, *CoRR* **abs/1910.07207**.
URL: *http://arxiv.org/abs/1910.07207*

Ellenberger, B. [2018–2019]. Pybullet gymperium, https://github.com/benelot/pybullet-gym.

Ernst, D. [2020]. Info8003-1: Optimal sequential decision making for complex problems, http://blogs.ulg.ac.be/damien-ernst/wp-content/uploads/sites/9/2020/02/RL-1.pdf. Accessed: 2021–12-12.

Ernst, D., Geurts, P. and Wehenkel, L. [2005]. Tree-based batch mode reinforcement learning., *Journal of Machine Learning Research* **6**: 503–556.

Frankle, J. and Carbin, M. [2018]. The lottery ticket hypothesis: Training pruned neural networks, *CoRR* **abs/1803.03635**.
URL: *http://arxiv.org/abs/1803.03635*

Frankle, J., Dziugaite, G. K., Roy, D. M. and Carbin, M. [2019]. Linear mode connectivity and the lottery ticket hypothesis, *CoRR* **abs/1912.05671**.
URL: *http://arxiv.org/abs/1912.05671*

Fujimoto, S., van Hoof, H. and Meger, D. [2018]. Addressing function approximation error in actor-critic methods, *CoRR* **abs/1802.09477**.
URL: *http://arxiv.org/abs/1802.09477*

Gale, T., Elsen, E. and Hooker, S. [2019]. The state of sparsity in deep neural networks, *CoRR* **abs/1902.09574**.
URL: *http://arxiv.org/abs/1902.09574*

Glorot, X. and Bengio, Y. [2010]. Understanding the difficulty of training deep feedforward neural networks, *AISTATS*.

Haarnoja, T., Tang, H., Abbeel, P. and Levine, S. [2017]. Reinforcement learning with deep energy-based policies, *CoRR* **abs/1702.08165**.
URL: *http://arxiv.org/abs/1702.08165*

Haarnoja, T., Zhou, A., Abbeel, P. and Levine, S. [2018]. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, *CoRR* **abs/1801.01290**.
URL: *http://arxiv.org/abs/1801.01290*

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A. and Dally, W. J. [2016]. EIE: efficient inference engine on compressed deep neural network, *CoRR* **abs/1602.01528**.
URL: *http://arxiv.org/abs/1602.01528*

Han, S., Pool, J., Tran, J. and Dally, W. J. [2015]. Learning both weights and connections for efficient neural networks, *CoRR* **abs/1506.02626**.
URL: *http://arxiv.org/abs/1506.02626*

Hasselt, H. [2010]. Double q-learning, *in* J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel and A. Culotta (eds), *Advances in Neural Information Processing Systems*, Vol. 23, Curran Associates, Inc.
URL: *https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf*

He, K., Zhang, X., Ren, S. and Sun, J. [2015a]. Deep residual learning for image recognition, *CoRR* **abs/1512.03385**.
URL: *http://arxiv.org/abs/1512.03385*

He, K., Zhang, X., Ren, S. and Sun, J. [2015b]. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, *CoRR* **abs/1502.01852**.
URL: *http://arxiv.org/abs/1502.01852*

He, Y. and Han, S. [2018]. ADC: automated deep compression and acceleration with reinforcement learning, *CoRR* **abs/1802.03494**.
URL: *http://arxiv.org/abs/1802.03494*

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G. and Silver, D. [2017]. Rainbow: Combining improvements in deep reinforcement learning, *CoRR* **abs/1710.02298**.
URL: *http://arxiv.org/abs/1710.02298*

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H. and Silver, D. [2018]. Distributed prioritized experience replay, *CoRR* **abs/1803.00933**.
URL: *http://arxiv.org/abs/1803.00933*

Jang, E., Gu, S. S. and Poole, B. [2017]. Categorical reparameterization with gumbel-softmax, *ArXiv* **abs/1611.01144**.

Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Sallab, A. A. A., Yogamani, S. K. and Pérez, P. [2020]. Deep reinforcement learning for autonomous driving: A survey, *CoRR* **abs/2002.00444**.
URL: *https://arxiv.org/abs/2002.00444*

Krizhevsky, A. [2009]. Learning multiple layers of features from tiny images.

Kumar, A., Agarwal, R., Ghosh, D. and Levine, S. [2020]. Implicit under-parameterization inhibits data-efficient deep reinforcement learning, *CoRR* **abs/2010.14498**.
URL: *https://arxiv.org/abs/2010.14498*

Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. [1998]. Gradient-based learning applied to document recognition, *Proceedings of the IEEE* **86**(11): 2278–2324.

Levine, S. [2021a]. Cs 285 introduction to reinforcement learning, https://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-4.pdf. Accessed: 2021–12-12.

Levine, S. [2021b]. Cs 285 introduction to reinforcement learning, https://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-7.pdf. Accessed: 2021–12-12.

Li, Y. and Ablavatski, A. [2021]. Build fast, sparse on-device models with the new tf mot pruning api.
URL: *https://blog.tensorflow.org/2021/07/build-fast-sparse-on-device-models-with-tf-mot-pruning-api.html*

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

Liu, Z., Sun, M., Zhou, T., Huang, G. and Darrell, T. [2018]. Rethinking the value of network pruning, *CoRR* **abs/1810.05270**.
URL: *http://arxiv.org/abs/1810.05270*

Luong, N. C., Hoang, D. T., Gong, S., Niyato, D., Wang, P., Liang, Y.-C. and Kim, D. I. [2019]. Applications of deep reinforcement learning in communications and networking: A survey, *IEEE Communications Surveys Tutorials* **21**(4): 3133–3174.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D. and Kavukcuoglu, K. [2016]. Asynchronous methods for deep reinforcement learning, *CoRR* **abs/1602.01783**.
URL: *http://arxiv.org/abs/1602.01783*

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. A. [2013]. Playing atari with deep reinforcement learning, *CoRR* **abs/1312.5602**.
URL: *http://arxiv.org/abs/1312.5602*

Morcos, A. S., Yu, H., Paganini, M. and Tian, Y. [2019]. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers.

Munos, R., Stepleton, T., Harutyunyan, A. and Bellemare, M. G. [2016]. Safe and efficient off-policy reinforcement learning, *CoRR* **abs/1606.02647**.
URL: *http://arxiv.org/abs/1606.02647*

Neal, B. and Mitliagkas, I. [2019]. In support of over-parametrization in deep reinforcement learning: an empirical study.

Ota, K., Jha, D. K. and Kanezaki, A. [2021]. Training larger networks for deep reinforcement learning, *CoRR* **abs/2102.07920**.
URL: *https://arxiv.org/abs/2102.07920*

Paganini, M. and Forde, J. Z. [2020]. Bespoke vs. prêt-à-porter lottery tickets: Exploiting mask similarity for trainable sub-network finding, *CoRR* **abs/2007.04091**.
URL: *https://arxiv.org/abs/2007.04091*

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S. [2019]. Pytorch: An imperative style, high-performance deep learning library, *CoRR* **abs/1912.01703**.
URL: *http://arxiv.org/abs/1912.01703*

Poupart, P. [2020]. Cs885 reinforcement learning module 2, https://cs.uwaterloo.ca/~ppoupart/teaching/cs885-spring20/slides/cs885-module2.pdf. Accessed: 2021–12-12.

Riedmiller, M. [2005]. Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method, *in* J. Gama, R. Camacho, P. B. Brazdil, A. M. Jorge and L. Torgo (eds), *Machine Learning: ECML 2005*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 317–328.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M. S., Berg, A. C. and Fei-Fei, L. [2014]. Imagenet large scale visual recognition challenge, *CoRR* **abs/1409.0575**.
URL: *http://arxiv.org/abs/1409.0575*

Russell, S. J. and Norvig, P. [2009]. *Artificial Intelligence: a modern approach*, 3 edn, Pearson.

Schaul, T., Quan, J., Antonoglou, I. and Silver, D. [2016]. Prioritized experience replay.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T. P. and Silver, D. [2019]. Mastering atari, go, chess and

shogi by planning with a learned model, *CoRR* **abs/1911.08265**.
URL: *http://arxiv.org/abs/1911.08265*

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O. [2017]. Proximal policy optimization algorithms, *CoRR* **abs/1707.06347**.
URL: *http://arxiv.org/abs/1707.06347*

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D. [2016]. Mastering the game of go with deep neural networks and tree search, *Nature* **529**: 484–503.
URL: *http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html*

Simonyan, K. and Zisserman, A. [2015]. Very deep convolutional networks for large-scale image recognition.

Sokar, G., Mocanu, E., Mocanu, D. C., Pechenizkiy, M. and Stone, P. [2021]. Dynamic sparse training for deep reinforcement learning, *CoRR* **abs/2106.04217**.
URL: *https://arxiv.org/abs/2106.04217*

Srinivas, A., Laskin, M. and Abbeel, P. [2020]. CURL: contrastive unsupervised representations for reinforcement learning, *CoRR* **abs/2004.04136**.
URL: *https://arxiv.org/abs/2004.04136*

Sutton, R. S. and Barto, A. G. [2018]. *Reinforcement learning: An introduction*, MIT press.

van Hasselt, H., Guez, A. and Silver, D. [2015]. Deep reinforcement learning with double q-learning, *CoRR* **abs/1509.06461**.
URL: *http://arxiv.org/abs/1509.06461*

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. and Polosukhin, I. [2017]. Attention is all you need, *CoRR* **abs/1706.03762**.
URL: *http://arxiv.org/abs/1706.03762*

Vischer, M. A., Lange, R. T. and Sprekeler, H. [2021]. On lottery tickets and minimal task representations in deep reinforcement learning, *CoRR* **abs/2105.01648**.
URL: *https://arxiv.org/abs/2105.01648*

Watkins, C. J. C. H. and Dayan, P. [1992]. Q-learning, *Machine Learning*, pp. 279–292.

Xiao, L., Bahri, Y., Sohl-Dickstein, J., Schoenholz, S. S. and Pennington, J. [2018]. Dynamical isometry and a mean field theory of cnns: How to train 10,000-layer vanilla convolutional neural networks.

Yang, Y., Zhang, G., Xu, Z. and Katabi, D. [2019]. Harnessing structures for value-based planning and reinforcement learning, *CoRR* **abs/1909.12255**.
URL: *http://arxiv.org/abs/1909.12255*

Yarats, D., Zhang, A., Kostrikov, I., Amos, B., Pineau, J. and Fergus, R. [2019]. Improving sample efficiency in model-free reinforcement learning from images, *CoRR* **abs/1910.01741**.
URL: *http://arxiv.org/abs/1910.01741*

Young, K. and Tian, T. [2019]. Minatar: An atari-inspired testbed for more efficient reinforcement learning experiments, *CoRR* **abs/1903.03176**.
URL: *http://arxiv.org/abs/1903.03176*

Yu, H., Edunov, S., Tian, Y. and Morcos, A. S. [2020]. Playing the lottery with rewards and multiple languages: lottery tickets in rl and nlp.

Zhang, S., Boehmer, W. and Whiteson, S. [2019]. Deep residual reinforcement learning, *CoRR* **abs/1905.01072**.
URL: *http://arxiv.org/abs/1905.01072*

Zhou, A., Haarnoja, T., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P. and Levine, S. [2018]. Soft actor-critic algorithms and applications, *CoRR*

abs/1812.05905.
URL: *http://arxiv.org/abs/1812.05905*

Zhou, H., Lan, J., Liu, R. and Yosinski, J. [2019]. Deconstructing lottery tickets: Zeros, signs, and the supermask, *CoRR* **abs/1905.01067**.
URL: *http://arxiv.org/abs/1905.01067*

Zoph, B. and Le, Q. V. [2016]. Neural architecture search with reinforcement learning, *CoRR* **abs/1611.01578**.
URL: *http://arxiv.org/abs/1611.01578*

Appendix A

Background reinforcement learning

A.1 Enforcing action bounds

We denote $u \in \mathbb{R}^d$ a random vector with density $\mu(u \mid s)$. The final action is an element-wise application of the hyperbolic tangent such that $a = \tanh(u)$. Since this transform is invertible and differentiable we can use the change of variables of a vectored-density to a vectored-density. Writing the Jacobian of the transform $a = \tanh(u)$ as $\frac{da}{du}$ we can write

$$\pi(a \mid s) = \mu(a \mid s) \left| \det \left(\frac{da}{du} \right) \right|^{-1} \tag{A.1}$$

The Jacobian of the transform is simply the diagonal matrix $\text{diag}(1 - \tanh^2(u))$. Thus the final log-density is $\log(\pi(a \mid s)) = \log(\mu(u \mid s)) - \sum_{i=1}^{d} \log(1 - \tanh^2(u_i))$.

A.2 Exploration and exploitation

Epsilon-greedy exploration

As discussed in Section 2.6. Epsilon-greedy exploration may require to decrease the exploration parameter ϵ. In order to do so, one may simply define a decay parameter for epsilon written d_ϵ as well as an initial value ϵ_i and a final value ϵ_f. Interpolating in between can be done as $\epsilon(t) = \epsilon_f + (\epsilon_i - \epsilon_f) \exp\left(-\frac{t}{d_\epsilon}\right)$ with t the iteration number.

A.3 Discrete soft actor critic pseudo-code

Algorithm 5: Discrete soft actor critic

Inputs:

B: the batch size, integer ≥ 1
N: the replay buffer size, integer ≥ 1
τ, the target networks update rate, float $\in [0, 1]$
α_Q, α_ϕ, α_λ, the learning rate, float > 0
$\overline{\mathcal{H}}$, the entropy target, float < 0

Initialization:

Initialize $Q_{\theta_{1,0}}$ and $Q_{\theta_{2,0}}$ with random weights
Initialize $Q_{\bar{\theta}_{1,0}}$ and $Q_{\bar{\theta}_{2,0}}$ with $\bar{\theta}_{1,0} = \theta_{1,0}$ and $\bar{\theta}_{2,0} = \theta_{2,0}$
Initialize experience replay memory for size N

Iterations:

for episode $= 1, M$ **do**
  Initialize the environment and take $s_0 \sim P(s_0)$
  for $t = 0, T-1$ **do**
    Draw the action from the policy network and act
    $a_t \sim \pi_{\phi_k}(s_t)$
    Observe s'_t and r_t
    Store (s_t, a_t, s'_t, r_t) into $\mathcal{D}$

    Sample uniformly a minibatch of transitions (s_i, a_i, s'_i, r_i) of size B from $\mathcal{D}$

    Draw the next actions and compute the targets
    for *i = 1, B* **do**

$$y_i = \begin{cases} r_i & \text{if } s'_i \text{ is terminal} \\ r_i + \gamma\, \pi_{\phi_k}(s'_i)^T \left(\min_{j=1,2} \left[Q_{\bar{\theta}_{j,k}}(s'_i) \right] - \log\left(\pi_{\phi_k}(s'_i)\right) \right) & \text{otherwise} \end{cases}$$

    end

    Update $\theta_{1,k}$ and $\theta_{2,k}$ according to:
    $\theta_{i,k+1} = \theta_{i,k} - \alpha_Q \nabla_{\theta_{i,k}} \left[\frac{1}{2B} \sum_{j=1}^{B} \left(Q_{\theta_{i,k}}(s_j, a_j) - y_j \right)^2 \right]$ for i $\in \{1, 2\}$

    Update ϕ according to:
    $\phi_{k+1} = \phi_k - \alpha_\pi \nabla_{\phi_k} \left[\frac{1}{B} \sum_{j=1}^{B} \pi_{\phi_k}(s_j)^T \left[-\lambda \log\left(\pi_{\phi_k}(s_j)\right) - \min_{i=1,2} Q_{\theta_{i,k}}(s_j) \right] \right]$
    Update λ according to:
    $\lambda_{k+1} = \lambda_k - \alpha_k \nabla_{\lambda_k} \left[\frac{1}{B} \sum_{j=1}^{B} \pi_{\phi_k}(s_j)^T \left[-\lambda_k \log\left(\pi_{\phi_k}(s_j)\right) - \lambda_k \overline{\mathcal{H}} \right] \right]$

    Update the target networks:
    $\theta'_{i,k+1} = \tau\, \theta'_{i,k} + (1 - \tau)\, \theta_{i,k}$ for i $\in \{1, 2\}$

    Update state: $s_{t+1} \leftarrow s'_t$
    Update iteration: $k \leftarrow k + 1$
  end
end

Appendix B

Background the Lottery Ticket Hypothesis

B.1 Iterative magnitude pruning with late resetting

Algorithm 6: Iterative magnitude pruning with late resetting

Inputs:
$p\%$, the pruning rate, float $\in [0, 100]$
M, a masking criterion: $M : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$
$\mathcal{N}$, a neural network architecture which defines a mapping $f(x\,;\,\theta)$
$\mathcal{D}_0$ a distribution over initializations for $\mathcal{N}$
$\mathcal{L}$, a training algorithm with limit on iterations, $\mathcal{L} : \mathbb{R}^{|\theta|} \times \mathbb{Z}^{+} \to \mathbb{R}^{|\theta|}$
k, a late resetting iteration number, integer > 0

Outputs:
$m \odot \theta_k$, a lottery ticket, $m \odot \theta_k \in \mathbb{R}^{|\theta|}$

Initialization:
Draw the initial parameters with $\theta_0 \sim \mathcal{D}_0$
Initialize the pruning mask $m = \mathbf{1}^{|\theta_0|}$
Train for k iterations
$\theta_k \leftarrow \mathcal{L}(\theta_0, k)$

Iterations:
for $j = 1, ..., M$ **do**
    Train the neural network until convergence, keeping the mask frozen
    $m \odot \theta_f \leftarrow \mathcal{L}(m \odot \theta_k, \,.\,)$

    Compute the mask scores of non-pruned weights using
    $s^i \leftarrow M(\theta_k^i, \theta_f^i)$ for $0 \le i < |\theta|$

    For the $(100 - p)\%$ top scores of non-pruned weights (break ties randomly):
    $m^i \leftarrow 1$
    For the $p\%$ bottom scores of non-pruned weights:
    $m^i \leftarrow 0$
end

B.2 Hyperparameters

B.2.0.1 DDQN

Parameter	Value
Number of actors	4
Batch size	256
Parameter update (sync. APEX)	64
Ratio actor update (sync. APEX) MinAtar	8
Ratio actor update (sync. APEX) others	4
Replay max size	10^6
Learning rate α	5×10^4
Epsilon (Adam)	10^{-4}
N-step return	1
Gamma γ	0.98
Polyak-averaging τ	0.99
Noise	Yes

Table B.1: Main hyperparameters for DDQN in classic control and pixel control.

Environment	Epsilon start min	Epsilon start max	Epsilon final	Epsilon decay
CartPole-v0	0.1	0.75	0.05	100 000
Pendulum-v0	0.1	0.75	0.05	50 000
Acrobot-v1	0.1	0.75	0.05	50 000
InvertedPendulumPyBylletEnv-v0	0.1	0.75	0.05	50 000
LunarLander-v2	0.1	0.75	0.05	100 000
MinAtar	0.1	0.4	0.05	100 000

Table B.2: Exploration noise hyperparameters for DDQN in classic control and pixel control.

Environment	Late-rewinding steps	Max samples	Target discounted reward
CartPole-v0	37 500	750 000	195
Pendulum-v0	37 500	750 000	-200
Acrobot-v1	50 000	1 000 000	-85
InvertedPendulumPyBylletEnv-v0	125 000	2 500 000	900
LunarLander-v2	100 000	2 000 000	150
MinAtar: asterix	50 000	5 000 000	20
MinAtar: breakout	50 000	5 000 000	25
MinAtar: freeway	50 000	5 000 000	100
MinAtar: space_invaders	50 000	5 000 000	100

Table B.3: Stopping criterion hyperparameters for DDQN in classic control and pixel control.

B.2.0.2 SAC

Classic control and simulations

Parameter	Value
Number of actors	4
Batch size	256
Parameter update (sync. APEX)	64
Ratio actor update (sync. APEX)	4
Replay max size	10^6
Temperature-autotune	True
Temperature init λ_0	0.25
Learning rate Q-networks α_Q	10^{-3}
Learning rate actor α_π	10^{-3}
Learning rate temperature α_λ	10^{-4}
Epsilon (Adam) (same for the 3)	10^{-4}
N-step return	1
Gamma γ	0.98
Polyak-averaging τ	0.99
Noise	None

Table B.4: Hyperparameters for SAC in classic control and physics simulations.

Environment	Late-rewinding steps	Max samples	Target discounted reward
CartPole-v0	37 500	750 000	195
Pendulum-v0	37 500	750 000	-200
Acrobot-v1	50 000	1 000 000	-85
InvertedPendulumPyBylletEnv-v0	125 000	2 500 000	900
InvertedDoublePendulumPyBulletEnv	125 000	500 000	8000
HalfCheetahPyBulletEnv-v0	50 000	1 000 000	1750
AntPyBulletEnv-v0	50 000	1 000 000	2000

Table B.5: Stopping criterion hyperparameters for SAC in classic control and physics simulations.

Pixel control (with MLP)

Parameter	Value
Number of actors	8
Batch size	256
Parameter update (sync. APEX)	64
Ratio actor update (sync. APEX)	4
Replay max size	10^6
Temperature-autotune	False
Temperature value	0.5
Learning rate Q-networks α_Q	5×10^{-4}
Learning rate actor α_π	5×10^{-4}
Learning rate temperature α_λ	10^{-4}
Epsilon (Adam) (same for the 3)	10^{-4}
N-step return	5
Gamma γ	0.98
Polyak-averaging τ	0.99
Noise	Yes

Table B.6: Hyperparameters for SAC in pixel based settings with simple dense layers (MLP).

Environment	Epsilon start min	Epsilon start max	Epsilon final	Epsilon decay
MinAtar: asterix	0.1	0.4	0.05	100 000
MinAtar: breakout	0.1	0.4	0.05	100 000
MinAtar: freeway	0.1	0.4	0.05	100 000
MinAtar: space_invaders	0.1	0.4	0.05	100 000

Table B.7: Exploration noise hyperparameters for SAC in pixel control.

Environment	Late-rewinding steps	Max samples	Target discounted reward
MinAtar: asterix	50 000	5 000 000	20
MinAtar: breakout	50 000	5 000 000	25
MinAtar: freeway	50 000	5 000 000	100
MinAtar: space_invaders	50 000	5 000 000	100

Table B.8: Stopping criterion hyperparameters for SAC in pixel control.

B.3 Network architectures

B.3.1 DDQN

```
==========================================================================================================================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param #
==========================================================================================================================
Critic                                   --                        --                        --
├─DenseQNetwork: 1                       --                        --                        --
│    └─ModuleList: 2-1                   --                        --                        --
├─DenseQNetwork: 1-1                     [1, 6]                    [1, 3]                    --
│    └─ModuleList: 2-1                   --                        --                        --
│    │    └─Linear: 3-1                  [1, 6]                    [1, 256]                  1,792
│    │    └─Linear: 3-2                  [1, 256]                  [1, 256]                  65,792
│    │    └─Linear: 3-3                  [1, 256]                  [1, 3]                    771
==========================================================================================================================
Total params: 68,355
Trainable params: 68,355
Non-trainable params: 0
Total mult-adds (M): 0.07
==========================================================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.27
Estimated Total Size (MB): 0.28
==========================================================================================================================
```

Figure B.1: DDQN Q-network architecture, a sequence of linear layers. Environment is Acrobot-v1 with 6 state space dimensions and 3 discrete actions. The Q-network takes the state as input and outputs a Q-value for every action.

```
==========================================================================================================================================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param #                   Kernel Shape
==========================================================================================================================================
Critic                                   --                        --                        --                        --
├─CnnMiniAtarQNetwork: 1                 --                        --                        --                        --
│    └─ModuleList: 2-1                   --                        --                        --                        --
├─CnnMiniAtarQNetwork: 1-1               [1, 10, 10, 4]            [1, 6]                    --                        --
│    └─Conv2d: 2-2                       [1, 4, 10, 10]            [1, 16, 8, 8]             592                       [4, 16, 3, 3]
│    └─ModuleList: 2-1                   --                        --                        --                        --
│    │    └─Linear: 3-1                  [1, 1024]                 [1, 512]                  524,800                   [1024, 512]
│    │    └─Linear: 3-2                  [1, 512]                  [1, 6]                    3,078                     [512, 6]
==========================================================================================================================================
Total params: 528,470
Trainable params: 528,470
Non-trainable params: 0
Total mult-adds (M): 0.57
==========================================================================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 2.11
Estimated Total Size (MB): 2.13
==========================================================================================================================================
```

Figure B.2: DDQN Q-network architecture, a set of 16 $3 \times 3 \times N_{\text{channel}}$ CNN filters followed by two linear layers. Environment is MinAtar: breakout with state space of shape $4 \times 10 \times 10$ and 6 discrete actions. The Q-network takes the state as input and outputs a Q-value for every action.

B.3.2 SAC

B.3.2.1 Discrete actions

```
==========================================================================================================================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param #
==========================================================================================================================
Critic                                   --                        --                        --
├─DenseQNetwork: 1                       --                        --                        --
│    └─ModuleList: 2-1                   --                        --                        --
├─DenseQNetwork: 1                       --                        --                        --
│    └─ModuleList: 2-2                   --                        --                        --
├─DenseQNetwork: 1-1                     [1, 3]                    [1, 3]                    --
│    └─ModuleList: 2-1                   --                        --                        --
│    │    └─Linear: 3-1                  [1, 3]                    [1, 256]                  1,024
│    │    └─Linear: 3-2                  [1, 256]                  [1, 256]                  65,792
│    │    └─Linear: 3-3                  [1, 256]                  [1, 3]                    771
├─DenseQNetwork: 1-2                     [1, 3]                    [1, 3]                    --
│    └─ModuleList: 2-2                   --                        --                        --
│    │    └─Linear: 3-4                  [1, 3]                    [1, 256]                  1,024
│    │    └─Linear: 3-5                  [1, 256]                  [1, 256]                  65,792
│    │    └─Linear: 3-6                  [1, 256]                  [1, 3]                    771
==========================================================================================================================
Total params: 135,174
Trainable params: 135,174
Non-trainable params: 0
Total mult-adds (M): 0.14
==========================================================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.54
Estimated Total Size (MB): 0.55
==========================================================================================================================
```

Figure B.3: The two SAC Q-networks architecture, a sequence of linear layers. Environment is Pendulum-V0 with 3 state space dimensions and 3 discrete actions. Action space has been discretized evenly from the continuous version. The Q-networks take the state as input and output a Q-value for every action.

```
==========================================================================================================================
Layer (type:depth-idx)                             Input Shape               Output Shape              Param #
==========================================================================================================================
Actor                                              --                        --                        --
├─DenseDiscreteStochasticActorNetwork: 1           --                        --                        --
│    └─ModuleList: 2-1                             --                        --                        --
├─DenseDiscreteStochasticActorNetwork: 1-1         [1, 3]                    [1, 3]                    --
│    └─ModuleList: 2-1                             --                        --                        --
│    │    └─Linear: 3-1                            [1, 3]                    [1, 256]                  1,024
│    │    └─Linear: 3-2                            [1, 256]                  [1, 256]                  65,792
│    │    └─Linear: 3-3                            [1, 256]                  [1, 3]                    771
│    └─Softmax: 2-2                                [1, 3]                    [1, 3]                    --
==========================================================================================================================
Total params: 67,587
Trainable params: 67,587
Non-trainable params: 0
Total mult-adds (M): 0.07
==========================================================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.27
Estimated Total Size (MB): 0.27
==========================================================================================================================
```

Figure B.4: The SAC actor architecture, a sequence of linear layers. Environment is Pendulum-V0 with 3 state space dimensions and 3 discrete actions. Action space has been discretized evenly from the continuous version. The actor network takes the state as input and outputs a discrete distribution over actions.

B.3.2.2 Continuous actions

```
==========================================================================================================================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param #
==========================================================================================================================
Critic                                   --                        --                        --
├─DenseContinuousQnetwork: 1            --                        --                        --
│    └─ModuleList: 2-1                   --                        --                        --
├─DenseContinuousQnetwork: 1            --                        --                        --
│    └─ModuleList: 2-2                   --                        --                        --
├─DenseContinuousQnetwork: 1-1          [1, 28]                   [1, 1]                    --
│    └─ModuleList: 2-1                   --                        --                        --
│    │    └─Linear: 3-1                  [1, 36]                   [1, 256]                  9,472
│    │    └─Linear: 3-2                  [1, 256]                  [1, 256]                  65,792
│    │    └─Linear: 3-3                  [1, 256]                  [1, 1]                    257
├─DenseContinuousQnetwork: 1-2          [1, 28]                   [1, 1]                    --
│    └─ModuleList: 2-2                   --                        --                        --
│    │    └─Linear: 3-4                  [1, 36]                   [1, 256]                  9,472
│    │    └─Linear: 3-5                  [1, 256]                  [1, 256]                  65,792
│    │    └─Linear: 3-6                  [1, 256]                  [1, 1]                    257
==========================================================================================================================
Total params: 151,042
Trainable params: 151,042
Non-trainable params: 0
Total mult-adds (M): 0.15
==========================================================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.60
Estimated Total Size (MB): 0.61
==========================================================================================================================
```

Figure B.5: The two SAC Q-networks architecture, a sequence of linear layers. Environment is AntPyBulletEnv-v0 with 28 state space dimensions and 8 action dimensions. The Q-networks take the state and action as inputs and output a Q-value for that combination (one value).

```
==========================================================================================================================
Layer (type:depth-idx)                             Input Shape               Output Shape              Param #
==========================================================================================================================
Actor                                              --                        --                        --
├─DenseGaussianStochasticActorNetwork: 1          --                        --                        --
│    └─ModuleList: 2-1                             --                        --                        --
├─DenseGaussianStochasticActorNetwork: 1-1        [1, 28]                   [1, 8]                    --
│    └─ModuleList: 2-1                             --                        --                        --
│    │    └─Linear: 3-1                            [1, 28]                   [1, 256]                  7,424
│    │    └─Linear: 3-2                            [1, 256]                  [1, 256]                  65,792
│    └─Linear: 2-2                                 [1, 256]                  [1, 8]                    2,056
│    └─Linear: 2-3                                 [1, 256]                  [1, 8]                    2,056
==========================================================================================================================
Total params: 77,328
Trainable params: 77,328
Non-trainable params: 0
Total mult-adds (M): 0.08
==========================================================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.31
Estimated Total Size (MB): 0.31
==========================================================================================================================
```

Figure B.6: The SAC actor architecture, a sequence of linear layers. Environment is AntPyBulletEnv-v0 with 28 state space dimensions and 8 action dimensions. The actor network takes the state as input and outputs two heads: one for the mean and one for the standard deviation (one of each for every action dim).

Appendix C

Results

C.1 Effective ranks

C.1.1 Low sranks environments

Here we provide sranks curves for the two low sranks environments Pendulum-v0 and Cartpole-v0 (DDQN agent).

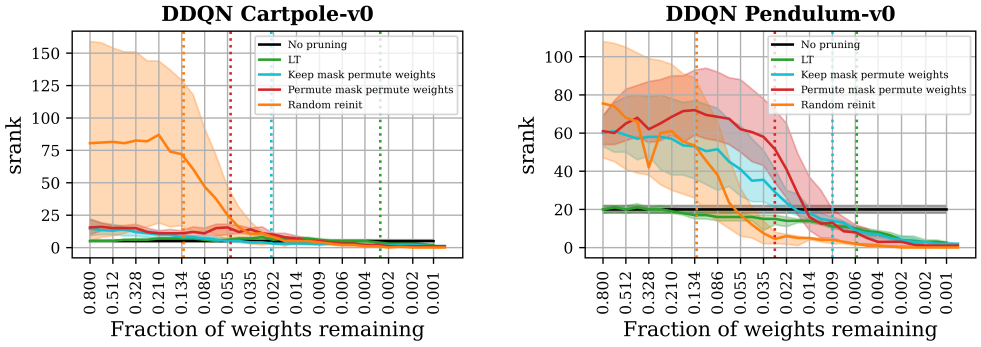

Figure C.1: Comparison of effective ranks of the penultimate layers output for tickets after training. Four variants: vanilla/winning ticket (green), permute mask permute mask (red), keep mask permute weights (cyan), random reinit which is a random subnetwork + random reinitialization (orange). Effective ranks are computed by drawing 500 000 samples from the unpruned trained models and sampling batches of states whose size is 1024 for classic control and 2048 for pixel-based control. These states are passed through the networks and embeddings are aggregated as matrices whose effective ranks are computed. Curves are medians surrounded by 25% and 75% quantiles estimated from 100 repetitions of the previously mentioned experiments.

C.2 LCA scores computation

The LCA scores were computed by going over the training curves which map training iteration to actual performance of the agent at that time. The performance was computed by the evaluator from synchronized APEX. Since the sampling rate was sometimes inconsistent - the evaluator taking longer (the evaluator is not synchronized in our implementation) we decided to linearly interpolate the training curves. The linear interpolation was carried every 100 training steps from the points available.

C.3 Input mask feature usage

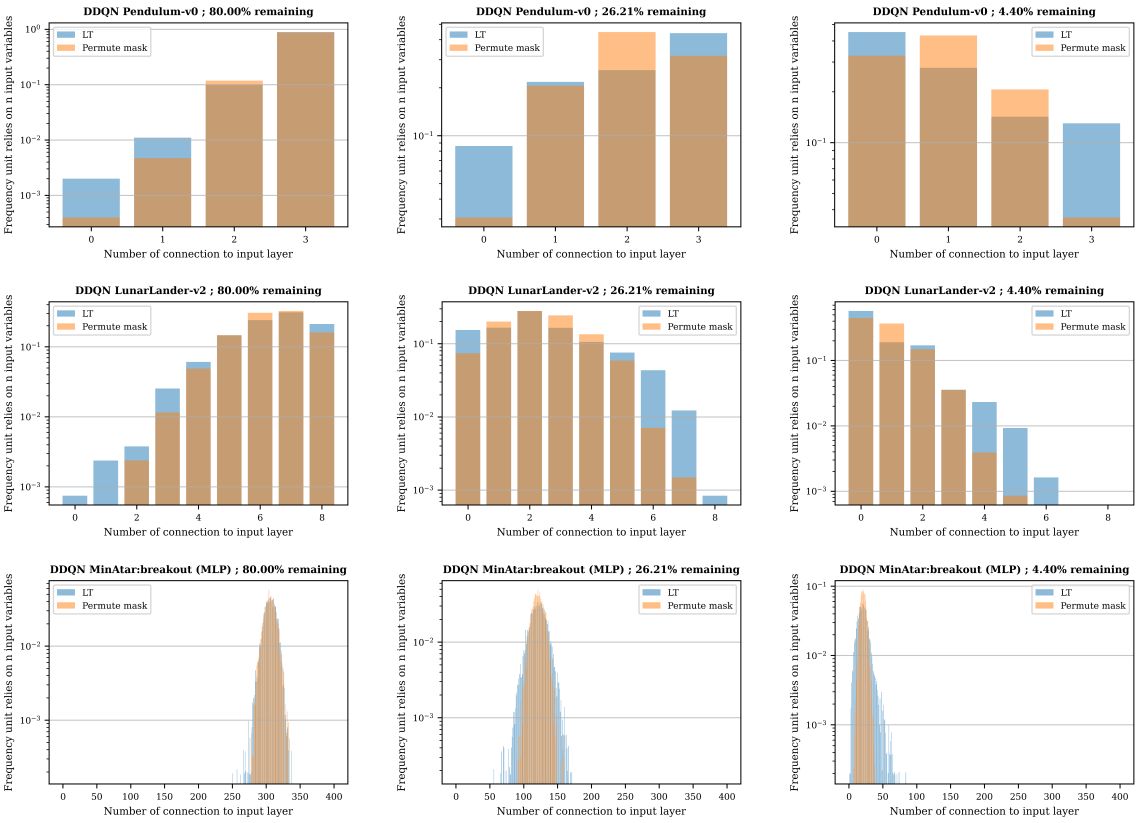

Figure C.2: Empirical frequency of the number of inputs a unit of the first processing layer (layer 1 in Figure 5.9) is connected to. Comparison of aggregated frequency computed from matching tickets (blue) with random permutation of the input mask (orange). Frequencies estimated by grouping the input masks from all the random seeds (usually 5). Y-axis is in log-scale.

C.4 Output mask

C.4.1 Mean heads of SAC

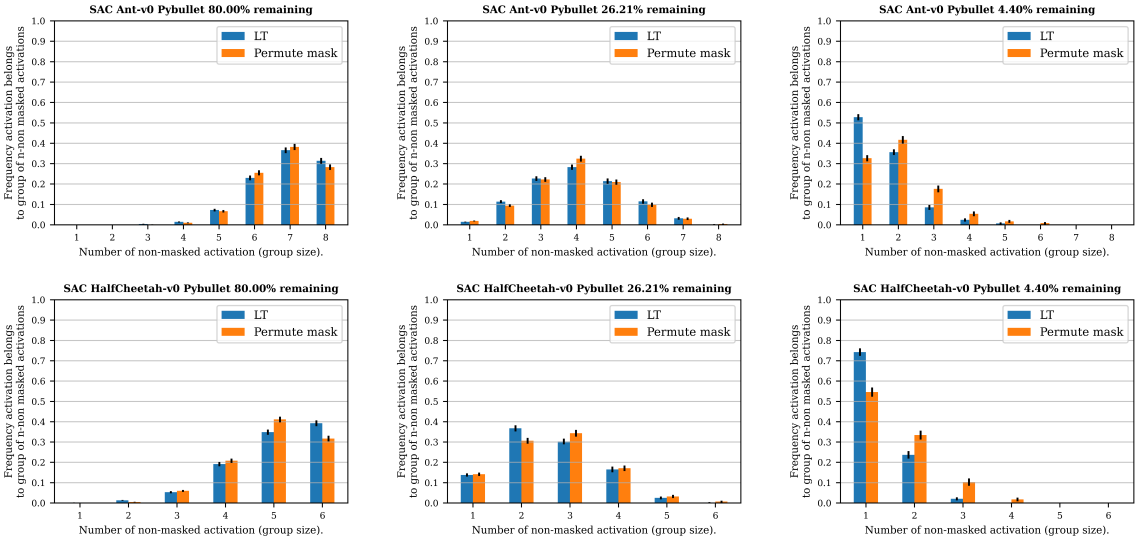

Figure C.3: Empirical frequencies for the number of times an active feature from the penultimate layer is used by the output layer. Histograms computed by pooling the masks obtained by the different random seeds and applying resampling (250 redraws). Bars are the mean frequencies across the samples. Error bars are standard deviations

C.4.2 Results without conditioning

In Figure C.4, the penultimate feature usage by the output layer without conditioning is depicted. These charts answer the question: *What is the probability of a feature from the penultimate layer to be used n times ?*.

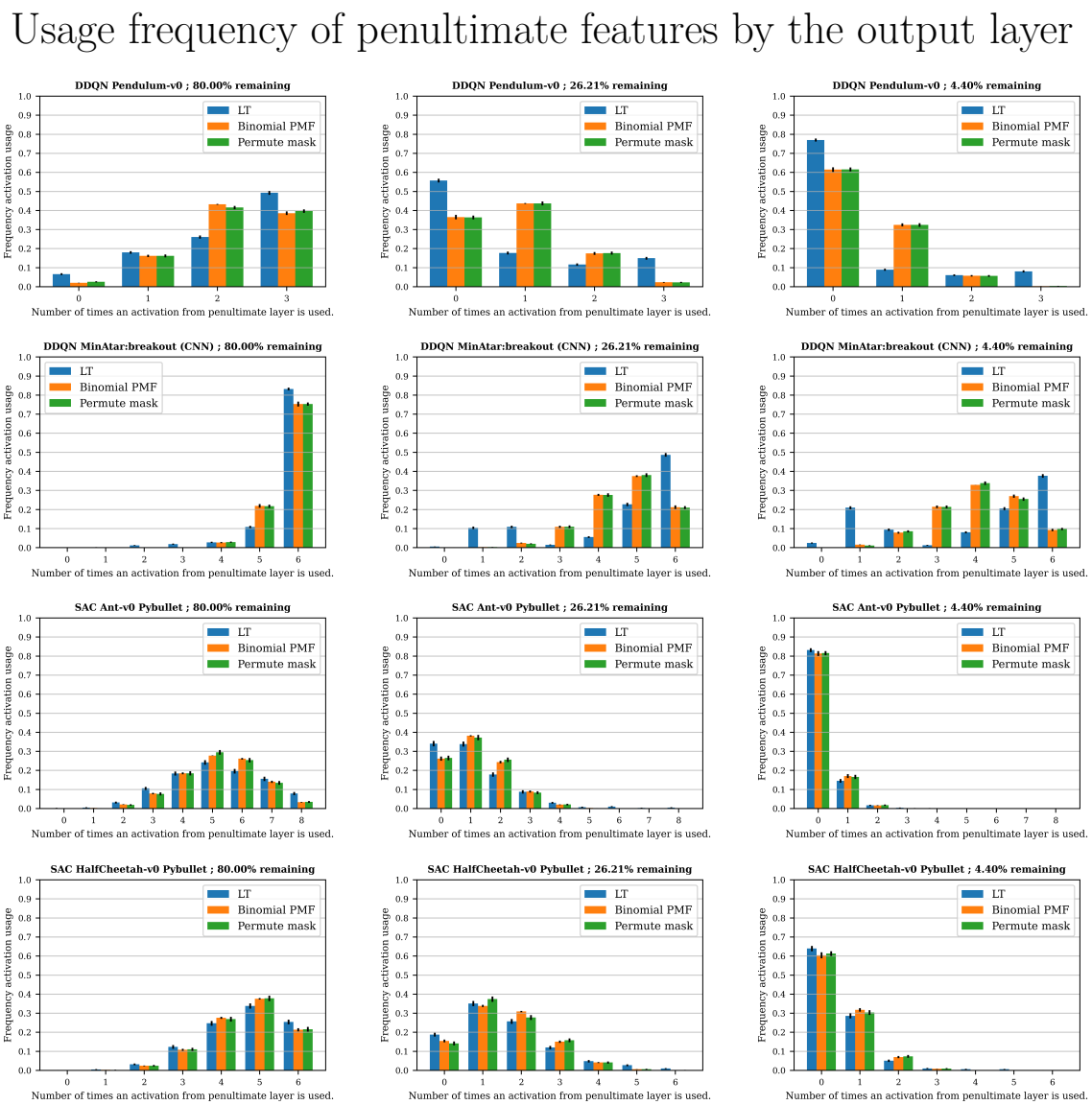

Figure C.4: Empirical frequencies for the number of times a feature from the penultimate layer is used by the output layer. Histograms computed by pooling the masks obtained by the different random seeds and applying resampling (250 redraws). Bars are the mean frequencies across the samples. Error bars are standard deviations.

C.5 The input mask

C.5.1 Fraction of weights connected per input

Here we give more details on the computation of the charts relating the fraction of remaining weights connected per input variable. The process involves to look at the input mask $M_{\theta_{N_1}} \in \{0, 1\}^{d_{\mathcal{S}} \times d_1}$. Every row corresponds to a different input variable. Summing the values of every row one by one gives the number of weights of the first layer spent per input variable. Then one simply divide these numbers by d_1. This gives the fraction of remaining weights per input variable. This process is repeated for every random seed (usually 5) and the results are averaged for the final charts.

C.5.2 Useless variables and SAC

Here we provide the figures for useless variables removal for SAC on the actor networks.

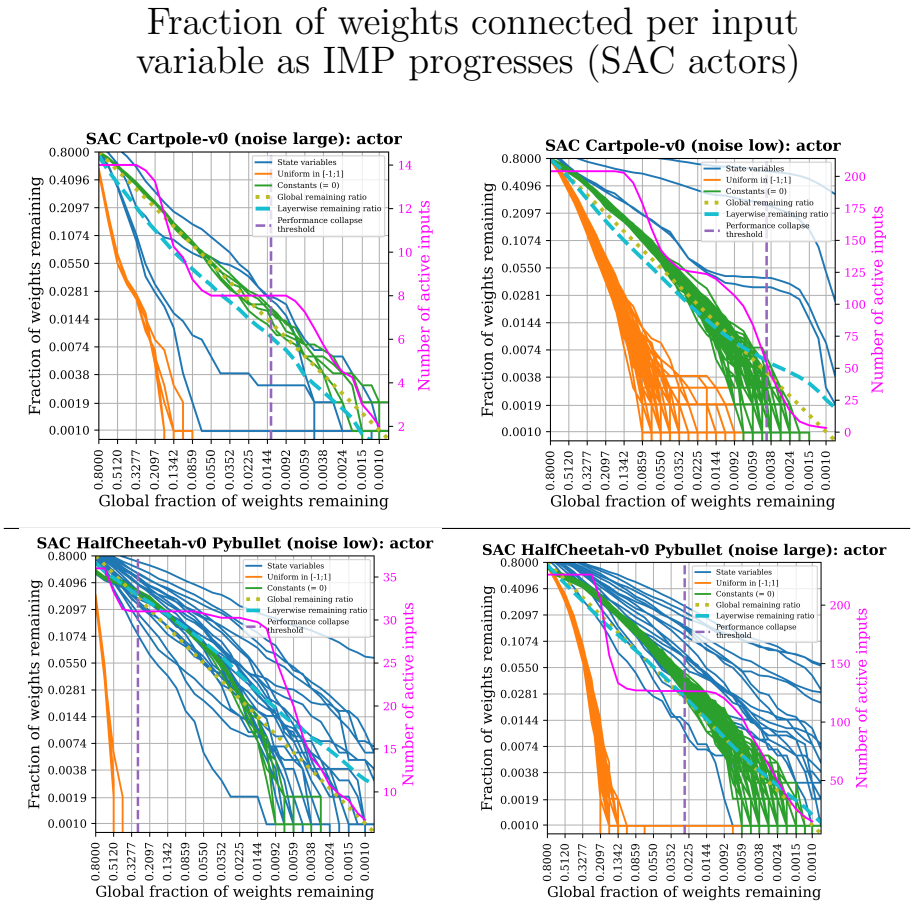

Figure C.5: Fraction of remaining weights connected per input as the global fraction of weights remaining decreases. Results for the actor-network of SAC, the actor network of SAC and the soft-Q-network of SAC. Green dotted line is the global remaining ratio computed on the whole model. Layerwise remaining ratio is the fraction of weights remaining in the input layer. Purple vertical bar indicates at what sparsity a given decrease in agent performance has be undergone. This decrease is computed as 10% of the gap between the performance of the unpruned model and the performance of the most pruned model. Pink curve and right y-axis relates to the number of active inputs. An active input is an input having at least one weight connected. The curve (in pink) is a mean estimated from the different random seeds (usually 5). Left y-axis and x-axis are both in logscale.

C.6 Pooled pruning

In Figure C.6, the remaining fraction of weights connected per layer for the actor and soft-Q-network is depicted as global sparsity increases.

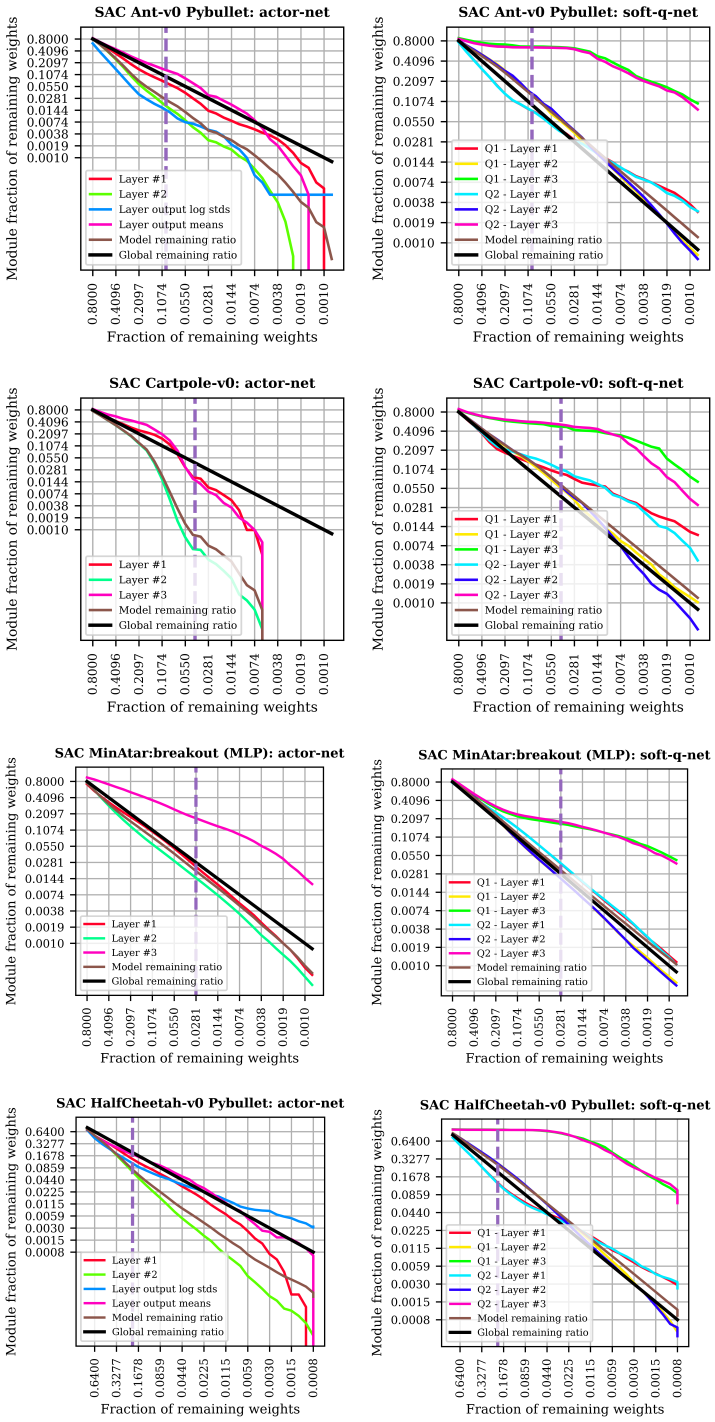

Figure C.6: Description. Fraction of remaining weights per layer as IMP progresses. Models are SAC actor-nets and soft-Q-nets. The parameters were pruned using the pooled variant instead of the global one. Curves are averaged estimated from the random seeds (usually 5). Both axis are log-scale. Purple dashed line is the performance collapse threshold. It is the pruning ratio at which 10% of the loss in performance from the unpruned model to the most pruned model (31 IMP steps) has been reached. It estimates the sparsity after which the model performance falls rapidly. **Observations.** Pooled pruning allows to prune the models differently. The actor network seems to be systematically pruned more heavily than the soft-Q-network.